

J. GLENN BROOKSHEAR

**CIÊNCIA DA
COMPUTAÇÃO**
UMA VISÃO ABRANGENTE

7a
EDIÇÃO



CIÊNCIA DA COMPUTAÇÃO



B873c Brookshear, J. Glenn.
Ciéncia da computação [recurso eletrônico] : uma visão
abrangente / J. Glenn Brookshear ; tradução Cheng Mei Lee. –
Dados eletrônicos. – 7 ed. – Porto Alegre : Bookman, 2008.

Editado também como livro impresso em 2005.
ISBN 978-85-7780-314-9

1. Ciéncia da computação – Introdução. I. Título.

CDU 004

Catalogação na publicação: Mônica Ballejo Canto – CRB 10/1023

J. GLENN BROOKSHEAR

CIÊNCIA DA COMPUTAÇÃO

UMA VISÃO ABRANGENTE

**7^a
EDIÇÃO**

Tradução:

CHENG MEI LEE

Consultoria, supervisão e revisão técnica desta edição:

JOÃO CARLOS DE ASSIS RIBEIRO DE OLIVEIRA

Mestre em Informática pela PUC/RJ

Professor do Departamento de Ciência da Computação da UFJF

**Versão impressa
desta obra: 2005**



2008

Obra originalmente publicada sob o título
Computer Science: An Overview, 7/ed;
BROOKSHEAR, J. GLENN
© 2003. Todos os direitos reservados.
Tradução autorizada a partir do original em língua inglesa,
publicado por Pearson Education, Inc., sob o selo Addison Wesley.
ISBN 0-201-78130-1

Capa:
GUSTAVO MACRI

Leitura final:
LETÍCIA V. ABREU DORNELLES

Supervisão editorial:
ARYSINHA JACQUES AFFONSO e DENISE WEBER NOWACZYK

Editoração e filmes:
WWW.GRAFLINE.COM.BR

Muitas das designações utilizadas pelos fabricantes e vendedores para distinguir seus produtos são consideradas marcas comerciais. Todas as designações das quais a Addison-Wesley tinha conhecimento foram impressas, neste livro, com a primeira letra (ou todas elas) em maiúscula.

Os programas e as aplicações neste livro foram incluídos por seu valor instrutivo. Embora cuidadosamente testados, não foram escritos com uma determinada finalidade. Os editores se isentam de qualquer responsabilidade em relação aos programas e às aplicações.

Reservados todos os direitos de publicação, em língua portuguesa, à
ARTMED® EDITORA S.A.
(BOOKMAN® COMPANHIA EDITORA é uma divisão da ARTMED® EDITORA S.A.)
Av. Jerônimo de Ornelas, 670 - Santana
90040-340 Porto Alegre RS
Fone (51) 3027-7000 Fax (51) 3027-7070

É proibida a duplicação ou reprodução deste volume, no todo ou em parte,
sob quaisquer formas ou por quaisquer meios (eletrônico, mecânico, gravação,
fotocópia, distribuição na Web e outros), sem permissão expressa da Editora.

SÃO PAULO
Av. Angélica, 1091 - Higienópolis
01227-100 São Paulo SP
Fone (11) 3665-1100 Fax (11) 3667-1333

SAC 0800-703-3444

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

*Aos meus pais,
Garland e Reba Brookshear*

Prefácio

Este livro proporciona uma introdução à Ciência da Computação. Ele cobre o assunto com bastante profundidade, de modo a transmitir uma apreciação justa sobre os tópicos envolvidos.

Audiência

Escrevi este texto para os estudantes da Ciência da Computação e de outras disciplinas. A maioria dos estudantes da área inicia os seus estudos com a ilusão de que a Ciência da Computação se resume em programação e navegação na Web, uma vez que isso é essencialmente o que ela viu. No entanto, a Ciência da Computação é muito mais que isso. Assim, os iniciantes precisam conhecer em extensão o assunto no qual desejam se especializar. Proporcionar essa exposição é o objetivo deste livro. Ele dá aos estudantes uma visão abrangente da Ciência da Computação — o fundamento para que eles possam apreciar a relevância e a interdependência das disciplinas que irão cursar.

O mesmo embasamento é necessário para os estudantes de outras áreas que devem se relacionar na sociedade tecnológica em que vivem. Um curso sobre Ciência da Computação para eles deve prover um entendimento fundamental que envolva todo o campo, em vez de uma mera introdução ao uso de pacotes de *software* populares. Essa abordagem de apresentação é usada em cursos introdutórios às ciências naturais e foi o modelo que tive em mente enquanto escrevia este texto. A facilidade de acesso a pessoas de outras áreas foi um de meus principais objetivos. O resultado é que edições anteriores deste livro têm sido usadas com sucesso em cursos para estudantes das mais variadas disciplinas. Esta edição foi projetada visando dar continuidade a essa tradição.

Organização

Este texto segue uma abordagem de baixo para cima que progride do concreto ao abstrato — uma ordem que resulta em uma apresentação pedagógica consistente, na qual cada tópico leva ao próximo. Ele inicia com os fundamentos da arquitetura de computadores (Parte 1), progride com o *software* e seu processo de desenvolvimento (Parte 2), explora os tópicos de organização de dados e seu armazenamento (Parte 3) e conclui considerando as aplicações correntes e futuras da tecnologia da computação (Parte 4).

Enquanto escrevia o texto, eu pensava em termos de desenvolvimento de um enredo. Em consequência, não me surpreende que muitos estudantes tenham afirmado que a leitura do texto foi parecida com a de uma novela. Ainda assim, o texto é dividido em capítulos bem independentes e seções que podem ser lidas como unidades isoladas (veja Figura 0.1 no Capítulo 0) ou rearranjadas para formar seqüências alternativas de estudo. De fato, o livro é freqüentemente usado em cursos que cobrem o material em diferentes ordens. As alternativas mais comuns iniciam com o material dos Capítulos 4 e 5 (“Algoritmos” e “Linguagens de Programação”) e retornam aos capítulos iniciais, se desejado. Em contraste, sei de um caso em que o início é com o material sobre computabilidade, do Capítulo 11. (Ainda em outra instância, o texto tem sido usado em cursos *senior capstone*^{*}, e serve como espinha dorsal a partir da qual os estu-

^{*}N. de R. Semelhante a um curso de especialização.

dantes partem em projetos que abrangem diversas áreas). Sugiro a seguinte seqüência para quem deseja simplesmente uma visão condensada da novela:

<u>Secção</u>	<u>Tópico</u>
1.1-1.4	Básico em codificação e armazenamento de dados
2.1-2.3	Arquitetura e linguagem de máquina
3.1-3.3, 3.5, 3.7	Sistemas operacionais e redes
4.1-4.4	Algoritmos e projeto de algoritmos
5.1-5.4	Linguagens de programação
6.1-6.2	Engenharia de <i>software</i>
7.1-7.2	Estruturas elementares de dados
8.1-8.2	Estruturas elementares de arquivos
9.1-9.2, 9.6	Introdução à tecnologia de banco de dados
10.1-10.3	Inteligência artificial
11.1-11.2	Computabilidade

Além do enredo geral, há vários temas costurados ao longo do texto. Um deles é que a Ciência da Computação é dinâmica. O texto reiteradamente apresenta os tópicos em uma perspectiva histórica, discute o estado da arte e indica direções de pesquisa corrente. Outro tema é o papel da abstração e a maneira como as ferramentas abstratas são usadas para controlar a complexidade.

Sítios da Web

Este texto é apoiado pelo sítio no endereço <http://www.aw.com/brooksheat>, sítio oficial mantido pela Addison-Wesley. Ali você encontrará materiais para estudantes e professores, tais como *software* de apoio (por exemplo, simuladores para a máquina usada como exemplo no Capítulo 2 e descrita no Apêndice C), manuais de laboratório, vínculos com tópicos adicionais de interesse, um guia do instrutor e transparências PowerPoint. Talvez você também queira conferir meu sítio pessoal na Web no endereço <http://mscs.mu.edu/~glennb>. Ele não é muito formal (e está sujeito a meus caprichos), mas procuro manter lá alguma informação que você possa achar útil.

Aos estudantes

Fui introduzido no campo da computação durante minha estadia na marinha americana em fins da década de 1960 e início de 1970. (Sim, isso me deixa velho — mas acontecerá com você também. Além disso, a velhice me torna sensato, portanto você deve ouvir o que tenho a dizer.) Passei a maior parte desse tempo fazendo manutenção do sistema de *software* na instalação de computação da marinha em Londres, Inglaterra. Completada a minha estadia, retornei à Universidade e terminei meu doutorado em 1975. Desde então, tenho ensinado Ciência da Computação e Matemática.

Muita coisa mudou na Ciência da Computação ao longo dos anos, mas outro tanto permaneceu. Em especial, a Ciência da Computação era e ainda é fascinante. Há muitas coisas impressionantes acontecendo nela. O desenvolvimento da Internet, o progresso da inteligência artificial e a capacidade de coletar e disseminar a informação em proporções surpreendentes são apenas algumas coisas que afetarão a sua vida. Você vive em um mundo excitante, em constante mutação e tem a oportunidade de ser parte da ação. Assuma!

Sou um pouco não-conformista (alguns amigos diriam *mais* que um pouco); por isso, quando decidi escrever este texto, nem sempre segui as recomendações recebidas. Em especial, muitos argumentaram que alguns materiais são muito avançados para iniciantes. Entretanto, acredito que se um tópico é relevante, ele não deixará de sê-lo pelo fato de a comunidade acadêmica considerá-lo “tópico avançado”. Você merece um texto que apresente o quadro completo da Ciência da Computação — não uma visão aguada com apresentações artificialmente simplificadas apenas dos tópicos considerados aceitáveis para iniciantes.

Portanto, não evitei tópicos. Em vez disso, tenho procurado melhores explicações. Tentei proporcionar bastante profundidade para dar a você um quadro honesto do que é a Ciéncia da Computação. (Há uma diferença entre o fato de o lançamento de uma nave espacial fazer muito barulho e a constatação de que o ruído ressoa em cada osso de seu corpo). Assim como nos temperos de uma receita, você pode escolher omitir alguns tópicos nas páginas a seguir, mas eles estão lá para você provar se desejar — e eu o encorajo a fazê-lo.

Finalmente, devo assinalar que, em qualquer curso sobre tecnologia, os detalhes que você aprende hoje podem não ser os que você necessitará amanhã. O campo é dinâmico — isso também é empolgante. Este livro dará a você um quadro atual do assunto, bem como uma perspectiva histórica. Com esse embasamento, você estará preparado para crescer acompanhando o desenvolvimento da tecnologia. Encorajo-o a iniciar o processo de crescimento explorando o campo além do apresentado aqui. Aprenda a aprender.

Agradeço a confiança depositada em mim ao decidir ler o meu livro. Como autor, tenho a obrigação de produzir um texto que mereça o seu tempo. Espero que você ache que cumpri com a obrigação.

Aos instrutores

Há mais material neste texto do que o que normalmente pode ser coberto em um semestre, por isso, não hesite em saltar os tópicos que não se enquadrem nos objetivos de seu curso, ou alterar a ordem de apresentação se julgar conveniente. Escrevi o livro para ser usado como recurso — não como uma definição de curso. Você verá que, embora o texto siga um roteiro, os tópicos são cobertos de maneira independente que lhe permite selecioná-los a gosto (veja na Figura 0.7 um resumo visual do texto).

Na página de abertura de cada capítulo, usei asteriscos para indicar as seções que julgo opcionais — elas examinam mais profundamente o material ou desviam para direções que você pode não desejar seguir. Todavia, são meras sugestões. Em especial, você verá que a versão condensada do texto esboçada anteriormente neste prefácio omite muito mais do que apenas as seções “opcionais”. Por exemplo, considere o Capítulo 7, sobre “Estruturas de Dados”. Dependendo dos objetivos de seu curso, você pode manipular esse capítulo em qualquer das maneiras a seguir — todas já foram experimentadas por mim. Primeiramente, em um curso de Fundamentos de Computação, você pode escolher omitir o capítulo integralmente. Se desejar apenas introduzir o assunto de estruturas de dados, você provavelmente cobrirá apenas as seções 7.1 e 7.2 (como sugerido na versão condensada). Se, além disso, você pretender apresentar as estruturas propriamente ditas, precisará usar as seções 7.1 até a 7.6. Finalmente, se quiser estender o estudo para incluir tipos de dados especializados ou ponteiros em linguagem de máquina, você vai querer incluir as seções “opcionais” 7.7 e/ou 7.8.

Também sugiro que você considere a cobertura de alguns tópicos por meio de leitura individual, ou encoraje os estudantes a ler o material não incluído em seu curso. Penso que nós subestimamos os estudantes quando consideramos que tudo deve ser explicado em aula. Devemos ajudá-los a aprender a aprender por conta própria.

Já expliquei que o texto segue uma organização de baixo para cima, do concreto ao abstrato, mas gostaria de expandir isso um pouco. Como acadêmicos, freqüentemente pressupomos que os estudantes apreciarão a nossa perspectiva do assunto — provavelmente aquela que desenvolvemos ao longo de anos de trabalho no campo. Como professores, fazemos melhor apresentando o material a partir da perspectiva do estudante. Isso explica por que o texto inicia com a representação e o armazenamento de dados, arquitetura e linguagem de máquina. Esses são tópicos diretamente relacionados aos estudantes — eles podem ver os componentes do computador, segurá-los, e a maioria já terá comprado e usado algum computador. Iniciando o curso com esses tópicos, vejo os estudantes descobrindo as respostas de muitos “porquês” que os acompanham durante anos, e aprendendo a ver o curso como prático em vez de teórico. A partir desse início, é natural mover a atenção a tópicos mais abstratos, como descoberta, projeto, representação e complexidade de algoritmos, que, para nós do campo, são os principais tópicos do curso.

Todos nós estamos cientes de que os estudantes aprendem mais do que lhes ensinamos diretamente, e as lições que aprendem indiretamente em geral são mais bem assimiladas do que as estudadas

explicitamente. Isso é significativo quando devemos “ensinar” a resolução de problemas. Os estudantes não aprendem a resolver problemas estudando metodologias de resolução de problemas como um assunto isolado. Eles aprendem a resolução de problemas resolvendo problemas. Portanto, inclui numerosos problemas no texto e encorajo você a usá-los e expandi-los.

Outro tópico que coloco na mesma categoria é o do profissionalismo, da ética e da responsabilidade social. Não acredito que esse material deva ser apresentado como assunto isolado, mas emergir quando for relevante. Essa é a abordagem usada neste texto. Você verá que as seções 0.5, 3.7, 6.1, 6.8, 9.6, 10.1 e 10.7 apresentam tópicos como segurança, privacidade, obrigação e consciência social no contexto de redes, sistemas de banco de dados, engenharia de software e inteligência artificial. Você também verá que cada capítulo inclui uma coleção de questões chamadas *Questões Sociais* que desafiam os estudantes a refletir sobre a relação entre o material do texto e a sociedade em que vivem.

Características pedagógicas

Este texto é o produto de muitos anos em atividade de ensino. Como resultado, mostra-se rico em recursos pedagógicos. É muito importante a abundância de problemas para aumentar a participação dos estudantes — mais de 1.000 nesta sétima edição (1.010, para ser preciso). Eles são classificados como “Questões/Exercícios”, “Problemas de Revisão do Capítulo” e “Questões Sociais”. As “Questões/Exercícios” aparecem no final de cada seção. São uma revisão do material ali discutido, estendem a discussão ou sugerem tópicos a serem cobertos mais tarde. Essas questões são respondidas no Apêndice F.

Os “Problemas de Revisão do Capítulo” aparecem no fim de cada capítulo (exceto no capítulo introdutório). Foram projetados para servir como “tema de casa”, uma vez que cobrem o material de todo o capítulo e não são resolvidos no texto.

Além disso, no fim de cada capítulo estão as “Questões Sociais”. Elas foram projetadas para promover reflexão e discussão. Muitas podem ser usadas em projetos de pesquisas que culminem com pequenos relatórios escritos ou orais.

Cada capítulo também termina com uma lista chamada “Leituras Adicionais” que contém referências a outros materiais relacionados ao assunto do capítulo. Os sítios da Web, identificados anteriormente neste prefácio, também são bons lugares para procurar os materiais relacionados.

Sétima edição

Embora esta sétima edição mantenha a mesma estrutura de capítulos das edições anteriores, alguns tópicos foram adicionados, outros eliminados e grande parte do material restante foi reescrita para proporcionar um quadro relevante e atualizado da Ciência da Computação.

As distinções mais significativas entre a sexta e a sétima edições são pedagógicas por natureza. A maioria do material foi reorganizada e reescrita a fim de melhorar a clareza e simplificar as explanações. Por exemplo, as Seções 2.1 e 2.2 (“Arquitetura de Computadores” e “Linguagem de Máquina”) foram reorganizadas, a introdução formal aos algoritmos na Seção 4.1 foi amaciada, a introdução às estruturas de dados (Seção 7.1), reorganizada, a Seção 7.7 (“Tipos de Dados Personalizados”) foi revisada minuciosamente, o material sobre arquivos seqüenciais e do tipo texto combinado em uma única seção (8.2) e o material relativo à computabilidade (Seções 11.1 a 11.3), reescrito. Ademais, inúmeras figuras foram adicionadas e a arte final melhorou como um todo.

Houve, evidentemente, numerosas adições de tópicos. Incluem-se as técnicas de codificação de som (Capítulo 1), a cobertura expandida de redes (Capítulo 3), o desenvolvimento do código-fonte aberto (Capítulo 6), material adicional em direito de cópia e patentes (Capítulo 6), XML (Capítulo 8) e memória associativa (Seção 10.4). Além disso, numerosos quadros laterais que expandem o material no texto foram adicionados em toda parte.

Você também perceberá que esta sétima edição tem uma nova diagramação e arte final que dá ao livro uma aparência mais “aberta” do que a de versões anteriores. O objetivo é tornar o material do texto mais acessível e menos assustador aos estudantes iniciantes. Espero que você o aprecie.

Agradecimentos

Agradeço em primeiro lugar a todos aqueles que prestigiaram este livro, lendo-o e usando-o em suas edições anteriores. Fico honrado.

A cada nova edição, cresce a lista de colaboradores, tanto revisores como consultores. Atualmente, ela inclui J. M. Adams, C. M. Allen, D. C. S. Allison, B. Auernheimer, P. Bankston, M. Barnard, P. Bender, K. Bowyer, P. W. Brashear, C. M. Brown, B. Calloni, M. Clancy, R. T. Close, D. H. Cooley, L. D. Cornell, M. J. Crowley, F. Deek, M. Dickerson, M. J. Duncan, S. Fox, N. E. Gibbs, J. D. Harris, D. Hascom, L. Heath, P. B. Henderson, L. Hunt, M. Hutchenreuther, L. A. Jehn, K. Korb, G. Krenz, J. Liu, T. J. Long, C. May, W. McCown, S. J. Merrill, K. Messersmith, J. C. Moyer, M. Murphy, J. P. Myers, Jr., D. S. Noonan, S. Olariu, G. Rice, N. Rickert, C. Riedesel, J. B. Rogers, G. Saito, W. Savitch, R. Schlaflay, J. C. Schlimmer, S. Sells, G. Sheppard, Z. Shen, J. C. Simms, M. C. Slattery, J. Slimick, J. A. Slomka, D. Smith, J. Solderitsch, R. Steigerwald, L. Steinberg, C. A. Struble, C. L. Struble, W. J. Taffe, J. Talburt, P. Tromovitch, E. D. Winter, E. Wright, M. Ziegler e um anônimo. A esses indivíduos, o meu muito obrigado.

Agradeço também ao pessoal da Addison-Wesley, da Argosy Publishing e da Theurer Briggs Design, cujos esforços estão refletidos nestas páginas. Um grupo que se envolve na publicação de um livro se torna uma família. Esta minha família cresceu com a inclusão de pessoas maravilhosas durante a produção desta sétima edição.

Minha esposa Earlene e minha filha Cheryl têm sido tremendas fontes de encorajamento ao longo dos anos. Agradeço-lhes por terem me imbuído no espírito de autor. Elas viram que “o livro” pode verdadeiramente fazer um professor desligado ficar ainda mais desligado. É confortante poder ser levado a ocupações acadêmicas como escrever um livro, sabendo que alguém está segurando as rédeas do mundo real. Em especial, na manhã do dia 11 de dezembro de 1998, sobrevivi a um ataque cardíaco porque Earlene me levou para o hospital a tempo. (Para vocês da nova geração, eu diria que sobreviver a um ataque de coração equivale a receber um acréscimo em seu dever de casa.)

Finalmente, agradeço a meus pais, a quem este livro é dedicado, por infundir em mim a importância da educação. Terminei com o seguinte endosso, cuja fonte deve permanecer anônima: “O livro do nosso filho é bom mesmo. Todos devem lê-lo”.

J. G. B.

Sumário

Capítulo 0	<i>Introdução</i>	17
0.1	O estudo de algoritmos	18
0.2	As origens das máquinas computacionais	21
0.3	A ciéncia dos algoritmos	24
0.4	O papel da abstração	24
0.5	Repercussões sociais	27
	Questões sociais	27
	Leituras adicionais	29
PARTE UM: ARQUITETURA DE MÁQUINA		31
Capítulo 1	<i>Armazenamento de dados</i>	33
1.1	Bits e seu armazenamento	34
1.2	Memória principal	40
1.3	Armazenamento em massa	42
1.4	Representação da informação como padrões de bits	47
1.5	O sistema binário	54
1.6	A representação de números inteiros	56
1.7	A representação de frações	61
1.8	Compressão de dados	65
1.9	Erros de comunicação	69
	Problemas de revisão de capítulo	72
	Questões sociais	77
	Leituras adicionais	78
Capítulo 2	<i>Manipulação de dados</i>	79
2.1	Arquitetura de computadores	80
2.2	Linguagem de máquina	82
2.3	Execução de programas	86
2.4	Instruções aritméticas e lógicas	92
2.5	Comunicação com outros dispositivos	96
2.6	Outras arquiteturas	100
	Problemas de revisão de capítulo	102
	Questões sociais	107
	Leituras adicionais	108

PARTE DOIS: SOFTWARE 109

Capítulo 3	<i>Sistemas operacionais e redes</i>	111
3.1	A evolução dos sistemas operacionais	112
3.2	Arquitetura dos sistemas operacionais	115
3.3	A coordenação das atividades da máquina	120
3.4	O tratamento da competição entre processos	123
3.5	Redes	127
3.6	Protocolos de redes	133
3.7	Segurança	140
	Problemas de revisão de capítulo	143
	Questões sociais	147
	Leituras adicionais	148
Capítulo 4	<i>Algoritmos</i>	149
4.1	O conceito de algoritmo	150
4.2	A representação de algoritmos	152
4.3	Descoberta de algoritmos	159
4.4	Estruturas iterativas	164
4.5	Estruturas recursivas	172
4.6	Eficiência e correção	178
	Problemas de revisão de capítulo	186
	Questões sociais	191
	Leituras adicionais	192
Capítulo 5	<i>Linguagens de programação</i>	193
5.1	Perspectiva histórica	194
5.2	Conceitos tradicionais de programação	202
5.3	Módulos de programas	211
5.4	Implementação de linguagens	218
5.5	Programação orientada a objeto	226
5.6	Programação de atividades concorrentes	231
5.7	Programação declarativa	233
	Problemas de revisão de capítulo	238
	Questões sociais	242
	Leituras adicionais	243
Capítulo 6	<i>Engenharia de software</i>	245
6.1	A disciplina da engenharia de <i>software</i>	246
6.2	O ciclo de vida do <i>software</i>	248
6.3	Modularidade	252
6.4	Metodologias de projeto	256
6.5	Ferramentas do ofício	259
6.6	Testes	263
6.7	Documentação	264
6.8	Propriedade e responsabilidade de <i>software</i>	265
	Problemas de revisão de capítulo	268
	Questões sociais	271
	Leituras adicionais	272

PARTE TRÊS: ORGANIZAÇÃO DE DADOS 273**Capítulo 7 *Estruturas de dados* 275**

7.1	Básico de estruturas de dados	276
7.2	Matrizes	277
7.3	Listas	280
7.4	Pilhas	284
7.5	Filas	288
7.6	Árvores	291
7.7	Tipos de dados personalizados	300
7.8	Ponteiros em linguagem de máquina	304
	Problemas de revisão de capítulo	306
	Questões sociais	311
	Leituras adicionais	312

Capítulo 8 *Estruturas de arquivo* 313

8.1	O papel do sistema operacional	314
8.2	Arquivos seqüenciais	316
8.3	Indexação	324
8.4	<i>Hashing</i>	327
	Problemas de revisão de capítulo	332
	Questões sociais	335
	Leituras adicionais	336

Capítulo 9 *Estruturas de banco de dados* 337

9.1	Tópicos gerais	338
9.2	A abordagem de implementação em níveis	340
9.3	O modelo relacional	342
9.4	Bancos de dados orientados a objeto	352
9.5	A preservação da integridade de bancos de dados	355
9.6	Impacto social da tecnologia de banco de dados	358
	Problemas de revisão de capítulo	360
	Questões sociais	364
	Leituras adicionais	364

PARTE QUATRO: O POTENCIAL DAS MÁQUINAS 365**Capítulo 10 *Inteligência artificial* 367**

10.1	Inteligência e máquinas	368
10.2	Compreensão das imagens	371
10.3	Raciocínio	373
10.4	Redes neurais artificiais	382
10.5	Algoritmos genéticos	390
10.6	Outras áreas de pesquisa	393
10.7	Considerando as consequências	400
	Problemas de revisão de capítulo	402
	Questões sociais	407
	Leituras adicionais	408

<i>Capítulo 11 Teoria da computação</i>	409
11.1 Funções e sua computação	410
11.2 Máquinas de Turing	411
11.3 Linguagens de programação universais	414
11.4 Uma função incomputável	419
11.5 Complexidade de problemas	424
11.6 Criptografia de chave pública	431
Problemas de revisão de capítulo	438
Questões sociais	441
Leituras adicionais	442
<i>Apêndices</i>	443
A. ASCII	445
B. Circuitos para manipular representações em complemento de dois	447
C. Uma linguagem de máquina simples	451
D. Exemplos de programas em linguagens de alto nível	455
E. A equivalência entre estruturas iterativas e recursivas	463
F. Respostas das questões e dos exercícios	465
<i>Índice</i>	503

O

C A P Í T U L O

Introdução

A Ciência da Computação é a disciplina que busca construir uma base científica para diversos tópicos, tais como a construção e a programação de computadores, o processamento de informações, as soluções algorítmicas de problemas e o processo algorítmico propriamente dito. Nesse sentido, estabelece os fundamentos para as aplicações computacionais existentes, assim como as bases para as futuras aplicações. Essa extensão significa que não podemos aprender a Ciência da Computação pelo simples estudo de alguns tópicos isolados, ou pela utilização das ferramentas computacionais atuais. Em vez disso, para entender a Ciência da Computação, é preciso compreender o escopo e a dinâmica da grande variedade de tópicos.

Este livro foi projetado para prover tais fundamentos. Apresenta uma introdução integrada aos tópicos que compõem os currículos universitários típicos desta área. O livro pode servir, portanto, como referência para estudantes que se iniciam na Ciência da Computação, além de ser uma fonte de consulta para outros que estejam em busca de uma introdução a esta ciência, que fundamenta a sociedade computadorizada dos dias de hoje.

- 0.1 O estudo de algoritmos
- 0.2 A origem das máquinas computacionais
- 0.3 A ciência dos algoritmos
- 0.4 O papel da abstração
- 0.5 Repercussões sociais

0.1 O estudo de algoritmos

Começamos com o conceito mais fundamental da Ciência da Computação — o de algoritmo. Informalmente, um algoritmo é um conjunto de passos que definem a forma como uma tarefa é executada.¹ Por exemplo, há algoritmos para a construção de aeromodelos (expressos na forma de folhas de instruções de montagem), para a operação de lavadoras de roupa (normalmente afixados no lado interno da tampa da máquina), para tocar música (expressos na forma de partituras) e para a execução de truques de mágica (Figura 0.1).

Antes que uma máquina possa executar uma tarefa, um algoritmo que a execute deve ser descoberto e representado em uma forma compatível com a máquina. Uma representação compatível com a máquina de um algoritmo é chamada **programa**. Os programas e os algoritmos que eles representam são coletivamente chamados **software**, em contraste com a máquina propriamente dita, que é conhecida como **hardware**.

O estudo dos algoritmos começou como um objeto da matemática. De fato, a procura por algoritmos era uma atividade significativa dos matemáticos muito antes do desenvolvimento dos computadores. O objetivo principal era descobrir um conjunto único de diretrizes que descrevessem como todos os

Efeito: O artista coloca algumas cartas de baralho sobre uma mesa, com a face voltada para baixo, e as embaralha bem enquanto as distribui sobre a mesa. Então, à medida que a audiência solicita cartas, especificando a sua cor (vermelha ou preta), o mágico as vira, exatamente da cor solicitada.

Segredo e Truque:

Passo 1. Selecionar dez cartas vermelhas e dez pretas. Fazer duas pilhas com essas cartas, abertas, separando-as de acordo com a sua cor.

Passo 2. Anunciar que você selecionou algumas cartas vermelhas e algumas cartas pretas.

Passo 3. Apanhar as cartas vermelhas. Com o pretexto de formar com elas uma pilha pequena, segurá-las, voltadas para baixo, com a mão esquerda, e, com o polegar e o indicador da mão direita, empurrar para trás cada extremidade da pilha, de modo que cada carta fique ligeiramente encurvada para trás. Então, colocar sobre a mesa a pilha de cartas vermelhas, voltadas para baixo, e dizer: “Aqui, nesta pilha, estão as cartas vermelhas”.

Passo 4. Apanhar as cartas pretas. De forma análoga ao que foi feito no passo 3,

imprimir a essas cartas uma leve curvatura para a frente. Então, devolvê-las à mesa, com a face voltada para baixo, e dizer: “E aqui estão as cartas pretas”.

Passo 5. Imediatamente após devolver as cartas pretas à mesa, usar as duas mãos para misturar as cartas vermelhas e pretas (ainda voltadas para baixo, da mesma forma como foram anteriormente espalhadas na mesa). Explicar que isso está sendo feito para que as cartas fiquem bem embaralhadas.

Passo 6. Enquanto houver cartas na mesa, executar repetidamente os seguintes passos:

6.1 Pedir que a audiência solicite uma carta de uma cor específica (vermelha ou preta).

6.2 Se a cor solicitada for vermelha e houver uma carta voltada para baixo, com formato côncavo, abrir essa carta e dizer: “Aqui está uma carta vermelha”.

6.3 Se a cor solicitada for preta e houver uma carta que tenha aparência convexa, abrir a carta e dizer: “Aqui está uma carta preta”.

6.4 Caso contrário, declarar que não há mais cartas da cor solicitada e abrir as cartas restantes para provar sua afirmação.

Figura 0.1 Um algoritmo para executar um truque de mágica.

¹Mais precisamente, um algoritmo é um conjunto ordenado e não-ambíguo de passos executáveis que definem uma atividade finita. Esses detalhes serão discutidos no Capítulo 4.

problemas de um determinado tipo poderiam ser resolvidos. Um dos mais conhecidos exemplos dessa antiga pesquisa é o algoritmo de divisão, para encontrar o quociente de dois números inteiros compostos de vários dígitos. Outro exemplo é o algoritmo de Euclides, descoberto pelo matemático grego de mesmo nome, que permite calcular o máximo divisor comum de dois inteiros positivos (Figura 0.2).

Uma vez descoberto um algoritmo que execute uma dada tarefa, sua execução já não dependerá do conhecimento dos princípios nos quais se baseia, restringindo-se apenas a seguir as instruções estabelecidas. (Pode-se seguir o algoritmo da divisão longa, para calcular um quociente, ou o algoritmo de Euclides, para obter o máximo divisor comum, sem necessitar compreender os princípios do seu funcionamento.) Em outras palavras, o algoritmo constitui uma codificação do raciocínio necessário à resolução do problema.

É por meio desta capacidade de captar e transferir inteligência mediante os algoritmos que são construídas máquinas que exibem comportamento inteligente. Por conseguinte, o nível de inteligência demonstrado pelas máquinas fica limitado pela inteligência que um algoritmo é capaz de transportar. Somente quando for possível obter um algoritmo que possa controlar a operação de uma tarefa será viável construir alguma máquina capaz de executá-la. Por outro lado, se não houver algoritmo capaz de executar tal tarefa, então a sua realização excederá as capacidades da máquina.

A tarefa do desenvolvimento de algoritmos torna-se, dessa forma, vital no campo da computação e, por isso, boa parte da Ciência da Computação se ocupa de assuntos a ela relacionados. No entanto, mediante estudo de alguns desses tópicos, adquire-se uma melhor compreensão da abrangência da Ciência da Computação. Um deles estuda como projetar novos algoritmos, uma questão muito semelhante ao problema geral de como solucionar problemas genéricos. Descobrir um algoritmo para solucionar um problema corresponde, essencialmente, a descobrir uma solução para esse problema. Logo, estudos nesta área da Ciência da Computação têm forte relação com outras áreas, tais como a da psicologia de resolução de problemas humanos e a das teorias de educação. Algumas dessas idéias serão consideradas no Capítulo 4.

Uma vez descoberto um algoritmo para solucionar um problema, o passo seguinte consiste em representá-lo de forma apropriada para que seja transmitido a alguma máquina, ou para que seja lido por outros seres humanos. Isto significa que se torna necessário transformar o algoritmo conceitual em um conjunto de instruções fácil de compreender e que elas sejam representadas sem ambigüidade. Sob esta perspectiva, estudos fundamentados em um conhecimento lingüístico e gramatical conduziram a uma grande diversidade de esquemas para a representação de algoritmos conhecidos como **linguagens de programação**, baseados em várias abordagens ao processo de programação conhecidas como paradigmas de programação. Algumas dessas linguagens e os paradigmas em que se baseiam serão tratados no Capítulo 5.

Uma vez que a tecnologia dos computadores tem sido aplicada a problemas cada vez mais complexos, os cientistas da computação descobriram que o projeto de grandes sistemas de software envolve mais do que o desenvolvimento de algoritmos individuais que executem as atividades requeridas. Acarreta também o projeto das interações entre os componentes. Para lidar com tais complexidades, os cientistas da computação se voltaram ao campo bem-estabelecido da engenharia, com o intuito de encontrar as ferramentas adequadas a esses problemas. O resultado é o ramo da Ciência da Computação chamado engenharia de software, que hoje se abastece em diversos campos, como engenharia, gerência de projetos, gerência de pessoal e projetos de linguagens de programação.

Descrição: Este algoritmo pressupõe que sejam fornecidos dois inteiros positivos e calcula o seu máximo divisor comum.

Procedimento:

- Passo 1.** Atribuir, inicialmente, a M e N os valores correspondentes ao maior e menor dos dois números inteiros positivos fornecidos, respectivamente.
- Passo 2.** Dividir M por N, e chamar de R o resto da divisão.
- Passo 3.** Se R não for 0, atribua a M o valor de N, a N o valor de R e retorne ao passo 2; caso contrário, o máximo divisor comum será o valor corrente de N.

Figura 0.2 O algoritmo de Euclides, que calcula o máximo divisor comum de dois inteiros positivos.

Outra área importante de Ciência da Computação é a que se ocupa do projeto e da construção de máquinas para executar algoritmos. Esses assuntos são tratados nos Capítulos 1 e 2. Embora nosso estudo sobre arquitetura de computadores incorpore algumas discussões tecnológicas, o objetivo não é o de dominar minuciosamente a forma como tal arquitetura é implementada por meio de circuitos eletrônicos. Isso desviaria demasiadamente para o campo da engenharia elétrica. Além disso, da mesma maneira que as antigas calculadoras, constituídas de engrenagens, deram lugar a dispositivos eletrônicos, a atual eletrônica pode vir a ser em breve substituída por outras tecnologias, entre as quais a ótica se apresenta como uma forte candidata. Nossa objetivo é ter uma compreensão da atual tecnologia que seja suficiente para que possamos observar suas ramificações nas máquinas de hoje, bem como a sua influência no desenvolvimento da própria Ciência da Computação.

O ideal seria que a arquitetura de computadores fosse produto exclusivo do nosso conhecimento dos processos algorítmicos, e que não fosse limitada pelas capacidades tecnológicas: em vez de permitir que a tecnologia forneça diretrizes para o projeto das máquinas e, portanto, para o modo como serão representados os algoritmos, gostaríamos que o nosso conhecimento de algoritmos funcionasse como força motriz por trás da arquitetura moderna das máquinas. Com os avanços da tecnologia, este sonho vai se tornando cada vez mais uma realidade. Hoje, é possível construir máquinas que permitem representar algoritmos como seqüências múltiplas de instruções, simultaneamente executadas, ou como padrões de conexões entre numerosas unidades de processamento, funcionando de forma parecida com o modo como nossas mentes representam a informação pelas conexões existentes entre os neurônios (Capítulo 10).

Outro contexto em que estudamos a arquitetura de computadores se refere ao armazenamento de dados e seu acesso. Neste ponto, as características internas de uma máquina freqüentemente se refletem nas suas características externas, as quais, bem como as formas de evitar seus efeitos indesejáveis, são objeto de estudo dos Capítulos 1, 7, 8 e 9.

Outro ponto fortemente relacionado com a concepção de máquinas de computação é o projeto da interface entre o computador e o mundo externo. Por exemplo, de que forma os algoritmos deverão ser inseridos na máquina, e de que maneira se pode informar a esta qual algoritmo deve ser executado? Solucionar tais problemas para ambientes nos quais se espera que a máquina preste serviços variados requer que sejam solucionados muitos outros problemas, que envolvem a coordenação de atividades e a alocação de recursos. Algumas dessas soluções são discutidas em nosso estudo sobre sistemas operacionais, no Capítulo 3.

À medida que se exigiu das máquinas a execução de tarefas cada vez mais inteligentes, a Ciência da Computação passou a buscar inspiração no estudo da inteligência humana. Acredita-se que, compreendendo o raciocínio da mente, poderíamos projetar algoritmos que imitassem tais processos, transferindo esta capacidade para as máquinas. Como resultado, surge a Inteligência Artificial, disciplina que se apóia fortemente em pesquisas das áreas da psicologia, biologia e lingüística. Discutimos no Capítulo 10 alguns desses tópicos da Inteligência Artificial.

A busca de algoritmos para controlar tarefas cada vez mais complexas conduz a questões relativas às suas limitações. A inexistência de algoritmos que resolvam um dado problema implica que ele não pode ser resolvido por uma máquina. Uma tarefa será considerada algorítmica se puder ser descrita por um algoritmo. Isto é, as máquinas são capazes de resolver apenas os problemas passíveis de solução algorítmica.

A constatação de que existem problemas sem solução algorítmica emergiu como um assunto na matemática na década de 1930, com a publicação do teorema da incompletude de Kurt Gödel. Este teorema declara essencialmente que em qualquer teoria matemática baseada no sistema de aritmética tradicional existem afirmações que não podem ser provadas nem refutadas. Em suma, um estudo completo de nosso sistema aritmético está além da capacidade das atividades algorítmicas.

O desejo de estudar as limitações dos métodos algorítmicos, decorrente da descoberta de Gödel, levou os matemáticos a projetar máquinas abstratas para executar algoritmos (isso ocorreu antes que a tecnologia fosse capaz de prover as máquinas reais para a investigação) e a estudar as potencialidades teóricas de tais máquinas hipotéticas. Atualmente, o estudo de algoritmos e máquinas constitui a espinha

dorsal da Ciência da Computação. Serão discutidos alguns dos tópicos relacionados a esta área no Capítulo 11.

0.2 As origens das máquinas computacionais

As máquinas abstratas idealizadas pelos matemáticos nos primeiros anos do século XX constituem importante parcela da árvore genealógica dos computadores modernos. Outros ramos desta árvore adêm de épocas mais remotas. De fato, é longa a história da busca pela máquina que executasse tarefas algorítmicas.

Um dos primeiros dispositivos de computação foi o ábaco. Sua história remonta às antigas civilizações grega e romana. Essa máquina é bastante simples e consiste em um conjunto de contas que correm em barras fixadas em uma armação retangular. As contas são movidas de um lado para outro sobre as barras, e suas posições representam valores armazenados. É por meio das posições das contas que esse “computador” representa e armazena dados. A execução do algoritmo é comandada por um operador humano. Assim, o ábaco constitui apenas um sistema de armazenamento de dados e deve ser combinado com uma pessoa para constituir uma máquina computacional completa.

Mais recentemente, o projeto das máquinas computacionais passou a se basear na tecnologia de engrenagens. Entre os inventores, estavam Blaise Pascal (1623-1662) da França, Gottfried Wilhelm Leibniz (1646-1716) da Alemanha e Charles Babbage (1792-1871) da Inglaterra. Essas máquinas representavam os dados por meio do posicionamento de engrenagens, e a introdução dos dados era mecânica, consistindo no estabelecimento conjunto das posições das diversas engrenagens. Os dados de saída das máquinas de Pascal e Leibniz eram obtidos observando-se as posições finais das engrenagens, da mesma forma como são lidos números nos hodômetros dos automóveis. Entretanto, Babbage projetou uma máquina que ele denominou máquina analítica, a qual podia imprimir em papel os dados de saída, eliminando assim erros de transcrição.

Tornou-se necessário um aumento da flexibilidade das máquinas para facilitar a execução dos algoritmos. A máquina de Pascal foi construída para efetuar somente o algoritmo de adição. Para tanto, uma sequência apropriada de instruções foi embutida na própria estrutura da máquina. De modo semelhante, a máquina de Leibniz tinha seus algoritmos firmemente incorporados à sua arquitetura, embora apresentasse uma série de operações aritméticas que podiam ser selecionadas pelo operador. A máquina das diferenças de Babbage (só foi construído um modelo de demonstração), ao contrário, podia ser modificada de modo a realizar diversos cálculos, mas a sua máquina analítica (ele nunca obteve os recursos necessários para sua construção) foi projetada para ler as instruções sob a forma de furos em cartões de papel. Assim, a máquina analítica de Babbage era programável. Com efeito, sua assistente Augusta Ada Byron é conhecida como a primeira programadora do mundo.

Augusta Ada Byron

Desde que o Departamento de Defesa dos EUA a homenageou usando seu nome em uma linguagem de programação, Augusta Ada Byron, a Condessa de Lovelace, tem sido muito comentada pela comunidade da computação. Ada Byron teve uma vida um tanto trágica em menos de 37 anos (1815-1852), complicada por saúde débil e o fato de ela não ter sido conformista em uma sociedade que limitava o papel profissional das mulheres. Ela ficou fascinada com as máquinas de Charles Babbage quando presenciou a demonstração de um protótipo de sua máquina das diferenças, em 1833. Sua contribuição à Ciência da Computação envolveu a tradução do francês para o inglês de um artigo que discutia os projetos de Babbage para a Máquina Analítica. Nessa tradução, Babbage a encorajou a anexar um adendo que descrevesse as aplicações da máquina e contivesse exemplos de como poderia ser programada para executar diversas tarefas. O entusiasmo de Babbage pelo trabalho de Ada Byron (ele fez numerosas sugestões e aparentemente contribuiu com alguns dos exemplos) era motivado pela esperança de que, com a publicação, teria o suporte financeiro para a construção da Máquina Analítica. (Sendo filha de Lord Byron, Ada Byron mantinha *status* de celebridade e tinha contatos potencialmente significativos em termos financeiros.) Esse suporte jamais se concretizou, mas o adendo de Ada Byron sobreviveu e é considerado um repositório dos primeiros programas de computador. Assim, ela é reconhecida atualmente como a primeira programadora do mundo.



Figura 0.3 O tear de Jacquard. (Cortesia de International Business Machines Corporation. Proibido o uso sem permissão.)

no Bell Laboratories, e o Mark I, concluído em 1944 na Harvard University por Howard Aiken e um grupo de engenheiros da IBM (Figura 0.4). Essas máquinas fizeram intenso uso de relés mecânicos eletronicamente controlados. Por essa razão, tornaram-se obsoletas logo depois de construídas, pois outros pesquisadores estavam então aplicando a tecnologia de válvulas na construção de computadores digitais totalmente eletrônicos. Aparentemente, a primeira dessas máquinas foi a de Atanasoff-Berry, construída de 1937 a 1941 no Iowa State College (atual Iowa State University) por John Atanasoff e seu assistente Clifford Berry. Outra pioneira foi a máquina conhecida como Colossus, construída na Inglaterra sob a direção de Tommy Flowers para decodificar mensagens alemãs durante o final da Segunda Guerra Mundial. (De fato, foram construídas dez dessas máquinas, mas os segredos militares, bem como as questões de segurança nacional, impediram que elas constassem na “árvore genealógica dos computadores”.) Logo surgiram outras máquinas mais flexíveis, como o ENIAC (*electronic numerical integrator and calculator*, ou integrador e calculador numérico eletrônico).

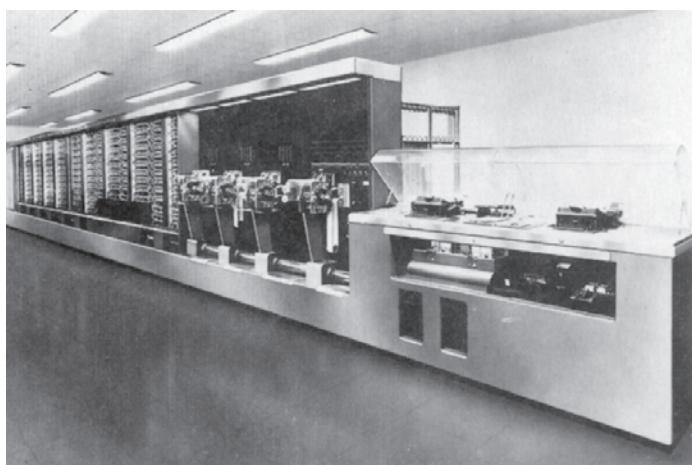


Figura 0.4 O computador Mark I.

A idéia de comunicar à máquina um algoritmo por intermédio de orifícios perfurados em papel não foi de Babbage. Em 1801, na França, Joseph Jacquard (1752-1834) aplicara uma técnica semelhante para controlar teares (Figura 0.3). Ele desenvolveu um tear no qual os passos a executar durante o processo de tecelagem eram determinados pelos padrões de orifícios em cartões de papel. Desta maneira, o algoritmo executado pela máquina poderia ser facilmente alterado, para produzir diferentes padronagens nos tecidos. Posteriormente, Herman Hollerith (1860-1929) aplicou a idéia de representar informação por meio de cartões perfurados para acelerar o processo de tabulação do censo americano de 1890. (Esse trabalho levou à criação da IBM.) Esses cartões passaram a ser conhecidos como cartões perfurados e sobreviveram até a década de 1970, como um meio popular de comunicação com os computadores. De fato, a técnica vive até os dias atuais, como atestam as questões de legitimidade de votos levantadas nas eleições presidenciais de 2000 nos EUA.

Com a tecnologia disponível na época, era financeiramente inviável a produção das complexas máquinas de engrenagens de Pascal, Leibniz e Babbage, e elas não se popularizaram. Somente com o advento da eletrônica, no início do século XX, a barreira foi ultrapassada. Como exemplos desse avanço, citam-se a máquina eletromecânica de George Stibitz, concluída em 1940

Berry, construída de 1937 a 1941 no Iowa State College (atual Iowa State University) por John Atanasoff e seu assistente Clifford Berry. Outra pioneira foi a máquina conhecida como Colossus, construída na Inglaterra sob a direção de Tommy Flowers para decodificar mensagens alemãs durante o final da Segunda Guerra Mundial. (De fato, foram construídas dez dessas máquinas, mas os segredos militares, bem como as questões de segurança nacional, impediram que elas constassem na “árvore genealógica dos computadores”.) Logo surgiram outras máquinas mais flexíveis, como o ENIAC (*electronic numerical integrator and calculator*, ou integrador e calculador numérico eletrônico).

co), desenvolvido por John Mauchly e J. Presper Eckert na Moore School of Electrical Engineering, University of Pennsylvania.

A partir daí, a história das máquinas computacionais foi a de uma tecnologia emergente em franco avanço, tendo sido seus marcos mais significativos a invenção dos transistores, o desenvolvimento subsequente dos circuitos integrados, o estabelecimento da comunicação por satélite e os avanços na tecnologia ótica. Hoje em dia, os computadores de mesa (bem como seus primos menores e portáteis conhecidos como *laptops*) possuem mais capacidade computacional do que as máquinas da década de 1940, que ocupavam uma sala inteira. Além disso, podem trocar informações rapidamente via sistemas globais de comunicação.

As origens dessas pequenas máquinas estão ligadas àqueles que tinham o computador como passatempo e começaram a fazer experiências com esses computadores feitos em casa logo após o desenvolvimento das grandes máquinas de pesquisa da década de 1940. Foi nessa atividade *underground* que Steve Jobs e Stephen Wosniak construíram um computador caseiro comercialmente viável e em 1976 fundaram a Apple Computer, Inc. para produzir e comercializar os seus produtos. Embora os produtos da Apple fossem populares, sua aceitação não foi grande na comunidade de negócios, que continuava procurando a bem-estabelecida IBM para suprir a maioria de suas necessidades computacionais.

Em 1981, a IBM introduziu o seu primeiro computador de mesa, chamado computador pessoal, ou simplesmente PC, cujo software subjacente foi desenvolvido por uma nova e esforçada companhia chamada Microsoft. O PC teve sucesso instantâneo e legitimou nas mentes da comunidade de negócios o computador de mesa como mercadoria estabelecida. Hoje em dia, o termo PC é largamente utilizado para se referir a todas essas máquinas (de vários fabricantes) cujos projetos são desdobramentos do computador pessoal da IBM, e cuja maioria é comercializada com o software da Microsoft. Às vezes, contudo, o termo PC é usado como sinônimo de *computador de mesa*.

A disponibilidade dos computadores de mesa colocou em destaque a tecnologia de computação na sociedade atual. Com efeito, a tecnologia da computação é tão predominante agora que saber como utilizá-la é fundamental para ser membro da sociedade moderna. É por meio dessa tecnologia que milhões de indivíduos têm acesso ao sistema de conexão global, conhecido como Internet, sistema esse que vem influenciando o estilo de vida e de comércio em base mundial.

A máquina das diferenças de Babbage

Os projetos das máquinas de Charles Babbage foram verdadeiramente os precursores dos projetos dos computadores modernos. Se tivesse a tecnologia conseguido produzir essas máquinas de uma maneira viável economicamente e as demandas por processamento de dados fossem comparáveis às atuais, as idéias de Babbage poderiam ter levado à revolução da computação ainda no século XIX. Na realidade, apenas um modelo de demonstração de sua Máquina das Diferenças foi construído enquanto ele viveu. Essa máquina determinava valores numéricos computando diferenças sucessivas. Podemos entender essa técnica considerando o cálculo dos quadrados de números inteiros. Comecemos com o conhecimento de que o quadrado de 0 é 0; o de 1, 1; o de 2, 4 e o de 3, 9. Com isso, podemos determinar o quadrado de 4 da seguinte maneira: primeiro calculamos as diferenças dos quadrados já conhecidos: $1^2 - 0^2 = 1$, $2^2 - 1^2 = 3$, $3^2 - 2^2 = 5$. Agora, calculamos as diferenças entre esses resultados: $3 - 1 = 2$ e $5 - 3 = 2$. Note que essas diferenças são iguais a 2. Pressupondo que essa consistência continue (a matemática pode provar isso), concluiremos que a diferença entre $(4^2 - 3^2)$ e $(3^2 - 2^2)$ também é 2. Assim, $(4^2 - 3^2)$ deve ser 2 mais do que $(3^2 - 2^2)$, logo $4^2 - 3^2 = 7$ e, assim, $4^2 = 7 + 3^2 = 16$. Agora que já possuímos o quadrado de 4, podemos continuar o procedimento e calcular o quadrado de 5 baseados nos valores de 1^2 , 2^2 , 3^2 e 4^2 . (Embora uma discussão mais aprofundada de diferenças sucessivas esteja além do escopo de nosso estudo, os estudantes de cálculo podem observar que o exemplo precedente se baseia no fato de que a derivada segunda de $y = x^2$ é uma linha reta.)

x	x^2	Primeira diferença	Segunda diferença
0	0		
1	1	1	
2	4	3	2
3	9	5	
4	16	7	2
5		2	

0.3 A ciência dos algoritmos

Dada a capacidade limitada de armazenamento de dados, o nível de detalhamento e os demorados procedimentos de programação, foi necessário restringir a complexidade dos algoritmos aplicados nas primeiras máquinas. Contudo, à medida que essas limitações foram superadas, as máquinas começaram a executar tarefas cada vez mais extensas e complexas. Conforme as tentativas de expressar a composição destas tarefas na forma algorítmica exigiam mais das habilidades da mente humana, crescentes esforços de pesquisa foram direcionados ao estudo dos algoritmos e do processo de programação.

Foi nesse contexto que começaram a dar frutos os trabalhos teóricos dos matemáticos. De fato, como decorrência do teorema da incompleta de Gödel, os matemáticos já haviam começado a investigar as questões relativas aos processos algorítmicos, que eram levantadas pela nova tecnologia. Com isso, ficou estabelecido o surgimento da disciplina conhecida como *Ciência da Computação*.

A disciplina que então se criava se estabeleceu com o nome de Ciência dos Algoritmos. Como vimos, seu escopo é amplo pois originou-se a partir de matérias diversas, como matemática, engenharia, psicologia, biologia, administração empresarial e lingüística. Muitos dos tópicos dessa ciência serão discutidos nos capítulos subsequentes. Em cada situação, o objetivo é introduzir as idéias principais do tópico, os mais recentes tópicos de pesquisa e algumas das técnicas empregadas para aumentar o conhecimento da área. É importante distinguir a Ciência da Computação das suas aplicações — distinção análoga à diferenciação entre a física e a engenharia mecânica. A física é a ciência que procura explicar a relação entre força, massa e aceleração e a engenharia mecânica, a aplicação dessa ciência. É necessário que o engenheiro mecânico entenda de física, mas, dependendo de sua especificidade, também deve saber as várias graduações do aço, a composição do concreto e a disponibilidade de posições no mercado para diversos produtos. Entretanto, o físico está interessado nas discrepâncias das teorias de Newton e como elas foram corrigidas nas considerações de Einstein. Portanto, não se deve esperar que um estudo de física inclua discussões sobre as limitações de carga em roldanas disponíveis no mercado, ou sobre as regulamentações governamentais a respeito dos cintos de segurança de automóveis.

De modo semelhante, nosso estudo não incluirá instruções para usar um programa particular de planilha, ou para instalar um navegador para a Internet. Isso não significa que esses tópicos não sejam importantes, apenas não fazem parte do nosso assunto. Nossa discussão a respeito da programação também não é focalizada no desenvolvimento das habilidades de programação em uma linguagem específica. Em vez disso, ela se concentra nos princípios que estão por trás das ferramentas atuais de programação, em como elas se desenvolveram e nos problemas que a pesquisa atual tenta solucionar.

À medida que se avança neste estudo de tópicos, é fácil perder a visão geral. Portanto, reunimos nossas idéias identificando algumas perguntas que definem a Ciência da Computação e estabelecendo o enfoque a ser dado para o seu estudo.

- Quais problemas podem ser solucionados por meio de processos algorítmicos?
- De que modo o processo de descoberta de algoritmos pode ser facilitado?
- Como se pode melhorar as técnicas de representação e comunicação de algoritmos?
- Como o nosso conhecimento de algoritmos e da tecnologia pode ser aplicado na obtenção de máquinas algorítmicas melhores?
- De que maneira as características de diferentes algoritmos podem ser analisadas e comparadas?

Note-se que o tema comum a todas essas perguntas é o estudo de algoritmos (Figura 0.5).

0.4 O papel da abstração

O conceito da abstração permeia de tal forma o estudo da Ciência da Computação e do projeto de sistemas de computadores que se torna necessário apresentá-lo neste capítulo introdutório. O termo **abstração**, como usado aqui, refere-se à distinção entre as propriedades externas de uma entida-

de e os detalhes de sua composição interna. É a abstração que nos permite ignorar os detalhes internos de um dispositivo complexo, como um computador, um automóvel ou um forno de microondas, e usá-lo como uma unidade compreensível. Além disso, é por meio da abstração que esses sistemas complexos são projetados. Um automóvel, por exemplo, é projetado de maneira hierárquica: no nível superior, é visto como uma coleção de grandes componentes, como o motor, o sistema de suspensão e direção, sem se levar em conta os detalhes internos de cada um desses sistemas. Cada componente, por sua vez, é construído a partir de outros componentes.

O *hardware* de um computador pessoal típico possui uma decomposição similar (Figura 0.6). No nível superior, ele pode ser visto como uma coleção de componentes, como o computador propriamente dito, um teclado, um monitor, um mouse e uma impressora. Se focalizarmos a impressora, veremos que ela consiste em componentes menores, como um mecanismo de alimentação de papel, um sistema de controle lógico e um sistema de injeção de tinta. Além disso, o mecanismo de alimentação de papel consiste em unidades ainda menores, como uma bandeja para o papel, um motor elétrico e um caminho para a alimentação.

Vemos então que a abstração nos permite construir, analisar e manejear sistemas grandes e complexos de computação que nos confundiriam se fossem vistos em sua totalidade em nível de detalhes. Com efeito, aplicando a abstração, abordamos tais sistemas em vários níveis de detalhe. Em cada um, vemos o sistema em termos de componentes chamados **ferramentas abstratas**, cuja composição interna podemos ignorar. Isso nos permite concentrar a nossa atenção no modo como cada componente interage com os outros componentes de mesmo nível e a coleção como um todo forma um componente no nível superior. Assim, compreendemos a parte do

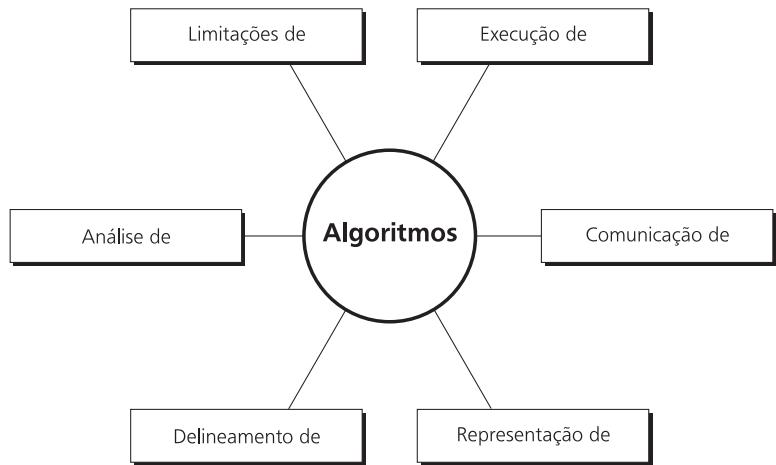


Figura 0.5 O papel central dos algoritmos na Ciência da Computação.

sistema que é relevante à tarefa desejada, sem nos perdermos em um mar de detalhes. Essa é a beleza da abstração.

Devemos notar que a abstração não se limita à ciência e à tecnologia. Ela é uma técnica de simplificação importante com a qual a nossa sociedade criou um estilo de vida que de outra forma seria impossível. Poucos são os que entendem como as várias conveniências do dia-a-dia são implementadas. Ingerimos alimentos e nos vestimos com roupas que sozinhos não poderíamos produzir. Usamos dispositivos elétricos sem entender a tecnologia subjacente. Usamos os serviços de outros sem conhecer os detalhes de suas profissões. Com cada novo avanço, uma pequena parte da sociedade escolhe se especializar em sua implementação enquanto o restante aprende a usar os resultados como ferramentas abstratas. Dessa maneira, o estoque de ferramentas abstratas da sociedade se expande e crescem as condições da sociedade para avançar ainda mais.

Este texto, por sua vez, aplica a abstração para obter capítulos (e mesmos seções dentro dos capítulos) que são surpreendentemente independentes. Os tópicos cobertos nos capítulos iniciais também servem como ferramentas abstratas nos capítulos posteriores. Assim, um entendimento detalhado dos capítulos iniciais não é necessário para compreender os demais. Você pode, por exemplo, iniciar o seu estudo lendo o Capítulo 10 (Inteligência Artificial) e mesmo assim compreender a maior parte do material. (Simplesmente use o índice para encontrar o significado de qualquer termo técnico que lhe tenha escapado.) No mesmo espírito da Figura 0.6, a Figura 0.7 apresenta a composição hierárquica deste texto, que consiste em quatro partes que podem ser estudadas de modo independente. Dentro

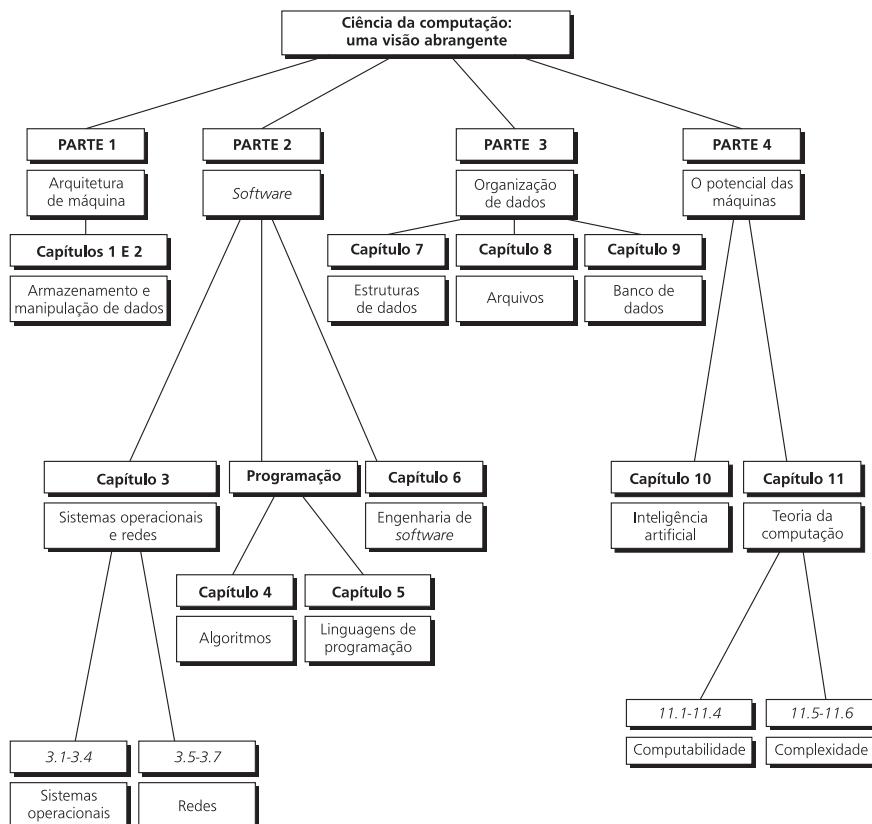


Figura 0.7 Visão do texto como uma hierarquia de ferramentas abstratas.

de cada parte, por sua vez, estão capítulos e agrupamentos de seções a serem estudados como unidades ainda menores de abstração. Esta decomposição não é mero artefato do texto — ela reflete a estrutura da ciência. De fato, a Ciência da Computação consiste em numerosos assuntos separados, ainda que relacionados.

0.5 Repercussões sociais

Os avanços na ciência e na tecnologia estão obscurecendo muitas distinções nas quais a nossa sociedade se baseou para as tomadas de decisões no passado e estão mesmo desafiando muitos princípios da sociedade. Qual é a diferença entre a presença de um comportamento inteligente e a inteligência propriamente dita? Quando a vida começa e quando termina? Qual é a diferença entre uma planta e um animal? Existe vida em outros sistemas estelares? Tais perguntas desafiam o indivíduo a reavaliar seus conceitos e, muitas vezes, a reconstruir os fundamentos de suas convicções.

A Ciência da Computação suscita essas questões em inúmeras situações. No Direito, questiona-se até onde vai o direito de posse de um *software* e os deveres que devem acompanhar tais direitos. Do ponto de vista ético, os indivíduos enfrentam numerosas opções, que desafiam conceitos antigos e tradicionais nos quais se baseia a sua conduta. No governo, questiona-se até que ponto devem ser regulamentadas a tecnologia e as aplicações da computação. Na sociedade como um todo, se pergunta se as aplicações dos computadores representam novas liberdades ou novos controles.

Solucionar esses dilemas de uma maneira racional requer um conhecimento dos elementos relevantes da ciência e da tecnologia. Por exemplo, para a sociedade tomar decisões racionais quanto ao armazenamento e à disposição de lixo nuclear, os seus membros precisam conhecer os efeitos da radiação, saber como proteger-se dos seus riscos e ter uma perspectiva realista do período de permanência do risco de radiação. De modo similar, para determinar se governos ou companhias têm ou não o direito de desenvolver grandes bancos de dados integrados que contenham informações sobre seus cidadãos ou clientes, os membros dessa sociedade devem possuir um conhecimento básico das capacidades, limitações e ramificações da tecnologia de banco de dados.

Este texto proporciona um embasamento fundamental, a partir do qual você pode abordar esses tópicos de uma maneira informada. Algumas seções são dedicadas aos aspectos sociais, éticos e legais. Por exemplo, discutimos matérias de privacidade em relação à Internet e à tecnologia de banco de dados e tópicos como direito de propriedade em relação ao desenvolvimento de *software*. Embora não sejam parte da Ciência da Computação, esses tópicos são importantes para os leigos e para quem pretende fazer carreira em campos relacionados à computação.

É claro que o conhecimento factual sozinho não necessariamente provê soluções às questões geradas pelos avanços recentes na Ciência da Computação. Raramente existe uma única resposta correta, e muitas soluções válidas são compromissos entre visões antagônicas. Assim, para encontrá-las, normalmente é necessária habilidade de ouvir, reconhecer outros pontos de vista, promover debates racionais e alterar a sua própria opinião quando novos conhecimentos são adquiridos. Com isso em mente, cada capítulo deste texto é finalizado com uma coleção de questões chamada Questões Sociais. Elas não são necessariamente questões para serem respondidas e sim, consideradas. Em muitos casos, uma resposta que inicialmente parece óbvia deixará de satisfazê-lo quando você explorar as alternativas. Encerramos esta Introdução com uma coleção dessas questões relacionadas a tópicos da computação em geral.

Questões sociais

As seguintes questões procuram auxiliar o leitor no entendimento de alguns assuntos éticos, sociais e legais no campo da computação. O objetivo não é meramente que sejam dadas respostas a tais questões. O leitor também deve justificá-las e verificar se as justificativas apresentadas preservam sua consistência de uma questão para outra.

1. Em geral, aceita-se a premissa de que a nossa sociedade é *diferente* do que teria sido se a revolução da computação não tivesse ocorrido. Nossa sociedade é *melhor* do que teria sido sem essa revolução? É *pior*? Sua resposta seria diferente se sua posição fosse outra nessa sociedade?
2. É aceitável participar da atual sociedade tecnológica sem fazer um esforço para conhecer os fundamentos dessa tecnologia? Por exemplo, os membros de uma democracia cujos votos geralmente determinam como a tecnologia será conduzida e utilizada têm a obrigação de conhecer essa tecnologia? A sua resposta depende de qual tecnologia está sendo considerada? Por exemplo, sua resposta será a mesma para a tecnologia nuclear e para a tecnologia da computação?
3. Usando dinheiro vivo, os indivíduos sempre tiveram a opção de administrar os seus negócios financeiros sem taxa de serviço. No entanto, como a nossa economia está cada vez mais automatizada, as instituições financeiras cobram taxas pelo acesso aos seus sistemas automáticos. Há momentos em que essa cobrança restringe, injustamente, o acesso de um indivíduo à economia? Por exemplo, suponha que um empregador remunere seus funcionários exclusivamente com cheques, e que todas as instituições financeiras cobrem uma taxa de serviço para cada cheque compensado ou depositado. Os funcionários estão sendo tratados injustamente? O que aconteceria se o empregador insistisse em pagar somente por meio de depósito direto?
4. No contexto da televisão interativa, até que ponto uma concessionária terá direito de extrair de crianças (talvez utilizando jogos interativos) informações sobre sua família? Por exemplo, deve ser permitido à companhia obter de uma criança informação sobre o perfil de consumo de seus pais? E quanto à informação relativa à própria criança?
5. Até que ponto um governo pode regulamentar a tecnologia da computação e suas aplicações? Por exemplo, consideremos os assuntos mencionados nas Questões 3 e 4. O que justificaria uma intervenção governamental?
6. Até que ponto as nossas decisões, relativas à tecnologia em geral e à tecnologia da computação em particular, afetarão as futuras gerações?
7. À medida que a tecnologia avança, nosso sistema educacional vai sendo constantemente desafiado a reconsiderar o nível de abstração em que são apresentados os diversos assuntos estudados. Questiona-se muitas vezes se uma certa habilidade do aluno continua sendo necessária, ou se deveria ser permitido aos estudantes apoiarem-se em uma ferramenta abstrata. Não se ensina mais os estudantes de trigonometria a encontrar os valores das funções trigonométricas com o uso de tabelas. Em vez disso, eles empregam calculadoras eletrônicas como ferramentas abstratas para encontrar esses valores. Alguns argumentam que a divisão de números longos também deveria ceder lugar à abstração. Que outros assuntos apresentam controvérsias semelhantes? Algum dia, o uso da tecnologia de vídeo eliminará a necessidade da leitura? Verificadores automáticos de ortografia eliminarão a necessidade de habilidades ortográficas?
8. Pressupõe que o conceito de biblioteca pública seja fortemente baseado na premissa de que todos cidadãos em uma democracia devem ter acesso à informação. À medida que a maioria da informação vai sendo guardada e disseminada por meio da tecnologia da computação, o acesso a essa tecnologia passa a ser direito de cada indivíduo? Em caso afirmativo, as bibliotecas públicas devem se constituir no canal pelo qual esse acesso é garantido?
9. Que considerações éticas aparecem em uma sociedade que se baseia no uso de ferramentas abstratas? Existem casos em que é antiético usar um produto ou serviço sem conhecer como ele é oferecido? Ou sem entender os efeitos colaterais de seu uso?

10. À medida que nossa economia vai se tornando cada vez mais automatizada, fica mais fácil para o governo monitorar as atividades financeiras dos cidadãos. Isso é bom ou ruim?
11. Quais tecnologias imaginadas por George Orwell (Eric Blair) em seu romance *1984* tornaram-se realidade? Elas são usadas da maneira que Orwell previu?
12. Os pesquisadores no campo da ética desenvolveram várias teorias com o intuito de analisar as decisões éticas. Uma dessas teorias é que as decisões devem ser baseadas nas consequências. Por exemplo, dentro dessa esfera está o utilitarismo que considera “correta” a decisão que produz o maior benefício para o maior número de indivíduos. Uma outra teoria se baseia nos direitos em vez das consequências. Ela considera “correta” a decisão que respeite os direitos dos indivíduos. As respostas às questões precedentes indicam que você tende a seguir a ética baseada nos direitos ou nas consequências?

Leituras adicionais

- Dejoie, D., G. Fowler, and D. Paradice. *Ethical Issues in Information Systems*. Boston: Boyd & Fraser, 1991.
- Edgar, S. L. *Morality and Machines*. Sudbury, MA: Jones and Bartlett, 1997.
- Forester, T., and P. Forrison. *Computer Ethics: Cautionary Tales and Ethical Dilemmas*. Cambridge, MA.: MIT Press, 1990.
- Goldstine, J. J. *The Computer from Pascal to von Neumann*. Princeton: Princeton University Press, 1972.
- Johnson, D. G. *Computer Ethics*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1994.
- Johnson, D. G. *Ethical Issues in Engineering*. Englewood Cliffs, NJ: Prentice-Hall, 1991.
- Kizza, J. M. *Ethical and Social Issues in the Information Age*. New York, Springer-Verlag, 1998.
- Mollenhoff, C. R. *Atanasoff: Forgotten Father of the Computer*. Ames: Iowa State University Press, 1988.
- Neumann, P. G. *Computer Related Risks*, Reading, MA: Addison-Wesley, 1995.
- Randell, B. *The Origins of Digital Computers*. New York: Springer-Verlag, 1973.
- Rosenoer, J. *CyberLaw: The Law of the Internet*, New York, Springer-Verlag, 1997.
- Shurkin, J. *Engines of the Mind*. New York: Norton, 1984.
- Spinello, R. A. and H. T. Tavani, *Readings in CyberEthics*, Sudbury, MA: Jones and Bartlett, 2001.
- Woolley, B. *The Bride of Science, Romance, Reason and Byron's Daughter*, New York: McGraw-Hill, 1999.

Arquitetura de máquina

A etapa principal no desenvolvimento de uma ciência é a construção de teorias, a serem confirmadas ou rejeitadas pela experimentação. Em alguns casos, essas teorias permanecem adormecidas por longos períodos, à espera do aparecimento de alguma tecnologia capaz de pô-las à prova. Em outros casos, as capacidades da tecnologia atual interferem nos assuntos da alçada da ciência.

O desenvolvimento da Ciência da Computação apresenta as duas características. Vimos que a ciência progrediu a partir de teorias originadas muito antes que a tecnologia pudesse produzir as máquinas preconizadas pelos antigos pesquisadores. Mesmo hoje, nosso crescente conhecimento dos processos algorítmicos está conduzindo ao projeto de novas máquinas, que desafiam os limites da tecnologia. Outros assuntos científicos, pelo contrário, têm suas raízes na aplicação da tecnologia moderna. A Ciência da Computação, em resumo, é a integração da pesquisa teórica com o desenvolvimento tecnológico, em uma harmoniosa simbiose com benefícios gerais.

Decorre que, para apreciar o papel dos vários assuntos da Ciência da Computação, é preciso compreender os fundamentos da atual tecnologia e a forma como ela influi no projeto e na implementação dos modernos computadores. Fornecer estes fundamentos é o propósito dos dois capítulos seguintes. No Capítulo 1, são discutidas as técnicas mediante as quais a informação é representada e armazenada nos computadores. No Capítulo 2, estudam-se os modos como as máquinas de hoje manipulam os seus dados.

CAPÍTULO 1

Armazenamento de dados

Neste capítulo, são tratados assuntos relacionados à representação e ao armazenamento de dados no computador. Às vezes, consideraremos tópicos de tecnologia, já que esses assuntos freqüentemente se refletem nas características externas das máquinas modernas. Entretanto, a maior parte da nossa discussão tratará de tópicos que só se ligarão ao projeto de computadores muito depois que a atual tecnologia vier a ser substituída, por se ter tornado obsoleta.

- 1.1 Bits e seu armazenamento**
Portas lógicas (*gates*) e *flip-flops*
Outras técnicas de armazenamento
A notação hexadecimal
- 1.2 Memória principal**
Organização da memória
Como medir a capacidade da memória
- 1.3 Armazenamento em massa**
Discos magnéticos
Discos óticos (*compact disks*)
Fita magnética
Armazenamento e recuperação de arquivos
- 1.4 Representação da informação como padrões de bits**
Representação de texto
Representação de valores numéricos
Representação de imagens
Representação de som
- * 1.5 O sistema binário**
Adição binária
Frações de números binários
- * 1.6 A representação de números inteiros**
A notação de complemento de dois
A notação de excesso
- * 1.7 A representação de frações**
A notação de vírgula flutuante
Erros de truncamento
- * 1.8 Compressão de dados**
Técnicas genéricas de compressão
Compressão de imagens
- * 1.9 Erros de comunicação**
Bits de paridade
Códigos de correção de erros

*Os asteriscos indicam sugestões de seções consideradas opcionais.

1.1 Bits e seu armazenamento

Os computadores representam a informação por meio de padrões de *bits*. Um **bit** (dígito binário) pode assumir os valores 0 e 1, os quais, por enquanto, consideraremos como meros símbolos, sem significado numérico. Na verdade, veremos que o significado de um *bit* varia de uma aplicação para outra. Algumas vezes, os padrões de *bits* são usados para representar valores numéricos e, em outras, para representar caracteres ou outros símbolos; também podem representar imagens ou sons. Armazenar um *bit* em um computador exige a presença de um dispositivo que possa assumir dois estados, como, por exemplo, um interruptor (ligado ou desligado), um relé (aberto ou fechado), ou um sinalizador de bandeira (erguida ou abaixada). Um dos estados representa 0 e o outro, 1. Nossa objetivo é estudar as formas como os *bits* são armazenados nos computadores modernos.

Portas lógicas (*gates*) e *flip-flops*

Começamos introduzindo as operações AND (e), OR (ou) e XOR (ou exclusivo), conforme esquematizado na Figura 1.1. Estas operações são semelhantes às operações aritméticas de produto e de soma, por operarem sobre um par de valores (as entradas da operação) e produzirem um terceiro valor, o resultado (saída) da operação. Entretanto, note-se que os únicos dígitos manipulados pelas operações AND, OR e XOR são 0 e 1; estas operações atuam conceitualmente sobre os valores TRUE (verdadeiro) e FALSE (falso) — 1 para verdadeiro, 0 para falso. Operações que manipulam valores verdadeiro/falso são chamadas **operações booleanas**, em homenagem ao matemático George Boole (1815-1864).

A operação booleana AND reflete a verdade ou falsidade de uma expressão formada combinando duas expressões menores por meio da conjunção *and*. Estas expressões apresentam a forma genérica

$$P \text{ AND } Q$$

onde *P* representa uma expressão e *Q*, outra. Por exemplo,

Caco é um sapo AND Senhorita Piggy é uma atriz.

Os dados de entrada para a operação AND representam a verdade ou falsidade dos componentes da expressão composta e a saída, a verdade ou falsidade da expressão composta. Como uma expressão da forma *P AND Q* só é verdadeira quando os seus dois componentes são verdadeiros, concluímos que 1 AND 1 resulta em 1; as demais situações resultariam em 0, de acordo com a Figura 1.1.

De modo semelhante, a operação OR está baseada em expressões compostas da forma

$$P \text{ OR } Q$$

onde, novamente, *P* representa uma expressão e *Q*, outra. Tais expressões são verdadeiras quando pelo menos um dos componentes é verdadeiro, o que concorda com o conceito da operação OR, descrito na Figura 1.1.

A operação AND

$\begin{array}{r} 0 \\ \text{AND } 0 \\ \hline 0 \end{array}$	$\begin{array}{r} 0 \\ \text{AND } 1 \\ \hline 0 \end{array}$	$\begin{array}{r} 1 \\ \text{AND } 0 \\ \hline 0 \end{array}$	$\begin{array}{r} 1 \\ \text{AND } 1 \\ \hline 1 \end{array}$
---	---	---	---

A operação OR

$\begin{array}{r} 0 \\ \text{OR } 0 \\ \hline 0 \end{array}$	$\begin{array}{r} 0 \\ \text{OR } 1 \\ \hline 1 \end{array}$	$\begin{array}{r} 1 \\ \text{OR } 0 \\ \hline 1 \end{array}$	$\begin{array}{r} 1 \\ \text{OR } 1 \\ \hline 1 \end{array}$
--	--	--	--

A operação XOR

$\begin{array}{r} 0 \\ \text{XOR } 0 \\ \hline 0 \end{array}$	$\begin{array}{r} 0 \\ \text{XOR } 1 \\ \hline 1 \end{array}$	$\begin{array}{r} 1 \\ \text{XOR } 0 \\ \hline 1 \end{array}$	$\begin{array}{r} 1 \\ \text{XOR } 1 \\ \hline 0 \end{array}$
---	---	---	---

Figura 1.1 As operações booleanas AND, OR e XOR (ou exclusivo).

Não há uma conjunção no idioma inglês* que expresse o significado da operação XOR. O resultado, ou seja, a saída da operação XOR será 1 quando uma de suas entradas for verdadeira e a outra, falsa. Por exemplo, uma declaração da forma $P \text{ XOR } Q$ significa “ P ou Q , mas não ambos”.

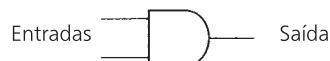
A operação NOT é outra operação booleana. Difere das operações AND, OR e XOR por apresentar uma única entrada. O valor da sua saída é o oposto da entrada: se o dado de entrada da operação NOT for verdadeiro, a saída será falsa, e vice-versa. Assim, se a entrada da operação NOT representar a veracidade** da afirmação

Fozzie é um urso,

então sua saída representará a veracidade** da afirmação oposta:

Fozzie não é um urso.

AND



Entradas	Saída
0 0	0
0 1	0
1 0	0
1 1	1

OR



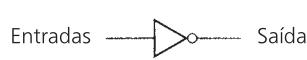
Entradas	Saída
0 0	0
0 1	1
1 0	1
1 1	1

XOR



Entradas	Saída
0 0	0
0 1	1
1 0	1
1 1	0

NOT



Entradas	Saída
0	1
1	0

Figura 1.2 Representação gráfica das portas lógicas AND, OR, XOR e NOT, com os valores de suas entradas e saídas.

*N. de T. Em português, também não. Aqui cabe uma observação similar para a saída da operação NOT.

**N. de T. Neste ponto, o termo veracidade está sendo empregado no sentido de que, se a afirmação em questão for verdadeira, o valor da entrada da operação NOT será considerado TRUE; caso contrário, FALSE.

Um dispositivo que fornece a saída de uma operação booleana, a partir de suas entradas, é denominado **porta lógica**. As portas lógicas podem ser implementadas por meio de diversas tecnologias, tais como engrenagens, relés e dispositivos óticos. As portas lógicas dos computadores modernos geralmente são constituídas de pequenos circuitos eletrônicos, em que os dígitos 0 e 1 são representados por níveis de tensão elétrica. No presente estudo, todavia, não haverá necessidade de nos aprofundarmos nesses pormenores. Para os nossos propósitos, basta representar as portas lógicas na forma simbólica, conforme mostrado na Figura 1.2. Note-se que as portas lógicas das operações AND, OR, XOR e NOT são representadas por meio de diagramas diferentes, com os dados de entrada posicionados em um dos lados do diagrama, enquanto os de saída são representados no lado oposto.

Tais portas lógicas permitem a realização de blocos, com os quais são construídos os computadores. Um passo importante nesta direção está descrito no circuito representado na Figura 1.3. Este é um exemplo de uma série de circuitos do tipo *flip-flop*. Um ***flip-flop*** é um circuito cuja saída apresenta um dos dois valores binários, permanecendo assim até que um pulso temporário em sua entrada, proveniente de outro circuito, venha a迫使-lo a modificar sua saída. Em outras palavras, o valor da saída alternará entre dois valores, de acordo com a ocorrência de estímulos externos. Enquanto as duas entradas do circuito da Figura 1.3 permanecerem com seus valores em 0, a sua saída (seja ela 0 ou 1) não se alterará. Todavia, introduzindo-se temporariamente o valor 1 na entrada superior, a saída será levada a assumir o valor 1. Se, no entanto, for introduzido temporariamente o valor 1 na entrada inferior, a saída será levada a assumir o valor 0.

Analisemos com maior cuidado estes resultados. Sem conhecer a saída do circuito descrito na Figura 1.3, suponhamos que o valor nele introduzido pela sua entrada superior seja 1, enquanto o da entrada inferior permanece com o valor 0 (Figura 1.4a). Isto levará o circuito a atribuir à saída da porta lógica OR o valor 1, independentemente do valor presente em sua outra entrada. Por outro lado, os valores das duas entradas da porta lógica AND agora serão iguais a 1, já que o valor da outra entrada desta porta já se encontra em 1 (valor este obtido passando-se o dado de entrada inferior do *flip-flop* através da porta lógica NOT). O valor da saída da porta lógica AND agora será igual a 1, o que significa que o valor da segunda entrada da porta lógica OR agora será igual a 1 (Figura 1.4b). Isto garante que o valor da saída da porta lógica OR permanecerá em 1, mesmo quando o valor da entrada superior do *flip-flop* retornar para 0 (Figura 1.4c). Em suma, o valor da saída do *flip-flop* passará a ser 1, permanecendo assim mesmo depois que o valor da entrada superior retornar a 0.

De forma semelhante, colocando-se temporariamente o valor 1 na entrada inferior, o valor da saída do *flip-flop* será forçado para 0, permanecendo assim mesmo que o valor da entrada inferior volte para 0.

A importância do *flip-flop*, do nosso ponto de vista, é que ele se mostra ideal para o armazenamento de um *bit* no interior de um computador. O valor armazenado é o valor de saída do *flip-flop*. Outros circuitos podem facilmente ajustar este valor enviando pulsos para as entradas do *flip-flop*, e outros ainda podem responder ao valor armazenado usando a saída do *flip-flop* como suas entradas.

Existem, é claro, outras maneiras de se construir um *flip-flop*. Uma alternativa é mostrada na

Figura 1.5. Se você experimentar com esse circuito, verá que embora apresente uma estrutura interna diferente, suas propriedades externas são as mesmas daquelas da Figura 1.3. Isto nos leva ao primeiro exemplo do papel das ferramentas abstratas. Quando projeta um *flip-flop*, um engenheiro considera as maneiras alternativas nas quais ele pode ser construído usando portas lógicas como módulos. Então, uma vez projetados os *flip-flops* e outros circuitos lógicos, o engenheiro pode usá-los como módulos para construir circuitos mais complexos. Assim, o projeto dos circuitos de um computador se faz em uma estrutura hierárquica, onde cada nível usa os componentes do nível abaixo como ferramentas abstratas.

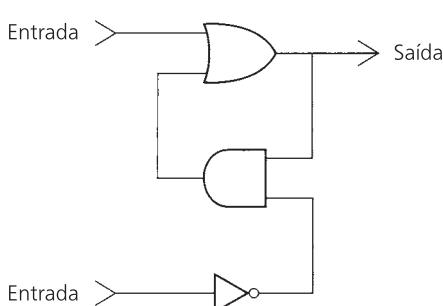


Figura 1.3 Um circuito simples de *flip-flop*.

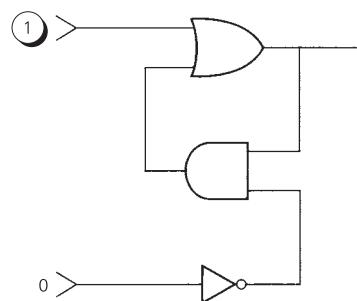
Outras técnicas de armazenamento

Na década de 1960, era comum os computadores conterem pequenos anéis ou argolas feitos de material magnético chamados **núcleos**, atravessados por fios elétricos muito finos. Passando uma corrente elétrica pelos fios, cada núcleo poderia ser magnetizado em uma de duas polaridades possíveis. Posteriormente, a polaridade desse campo magnético poderia ser consultada observando-se seu efeito sobre uma corrente elétrica forçada pelos fios que atravessavam o núcleo. Deste modo, os núcleos magnéticos realizavam uma forma de armazenamento de *bits* — o 1 era representado por um campo magnético em uma direção e o 0, por um campo magnético na outra. Tais sistemas são obsoletos hoje, devido ao seu tamanho e ao seu consumo de energia.

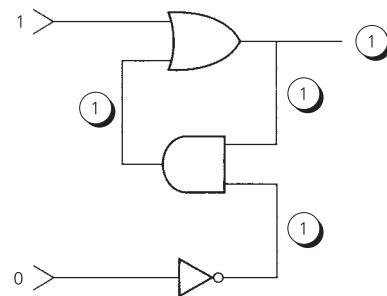
Um método mais recente para armazenar um *bit* é o capacitor, que consiste em duas pequenas placas metálicas posicionadas paralelamente mantendo uma pequena distância de separação. Se uma fonte de tensão for conectada às placas — positivo em uma placa e negativo na outra — as cargas da fonte se distribuirão nas placas. Então, quando a fonte de tensão for removida, essas cargas ficarão nas placas. Se estas forem conectadas mais tarde, uma corrente elétrica fluirá pela conexão e as cargas serão neutralizadas. Assim, um capacitor está em um de dois possíveis estados: carregado ou descarregado, um deles pode ser usado para representar o 0 e o outro, para representar o 1. A tecnologia atual é capaz de construir milhões de pequenos capacitores, bem como os seus circuitos, tudo em uma única pastilha (chamada **chip**). Assim, os capacitores vêm se tornando uma tecnologia popular para armazenamento de *bits* dentro das máquinas.

Flip-flops, núcleos e capacitores são exemplos de sistemas de armazenamento com diferentes graus de volatilidade. Um núcleo retém seu campo magnético após o desligamento da máquina. Um *flip-flop* perde o dado armazenado quando a fonte de energia é desligada. Por sua vez, as cargas nesses pequenos capacitores são tão frágeis que tendem a se dissipar neles próprios, mesmo enquanto a máquina está funcionando. Assim, a carga em um capacitor deve ser restaurada regularmente por um circuito conhecido como *refresh*. Considerando essa volatilidade, a memória do computador (Seção 1.2) construída com essa tecnologia, freqüentemente é chamada de **memória dinâmica**.

- a. 1 é forçado na entrada superior.



- b. Isto faz com que a saída da porta lógica OR apresente o símbolo 1 e, assim, a saída da porta lógica AND passe a ser 1.



- c. O 1 proveniente da porta lógica AND evita que a porta lógica OR altere sua saída depois que a entrada superior retornar ao valor 0.

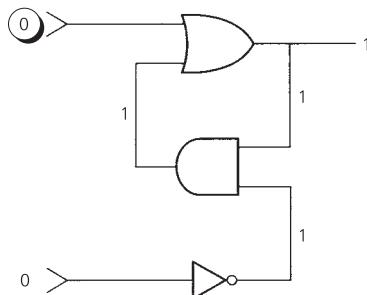


Figura 1.4 Como forçar o valor 1 na saída do *flip-flop*.

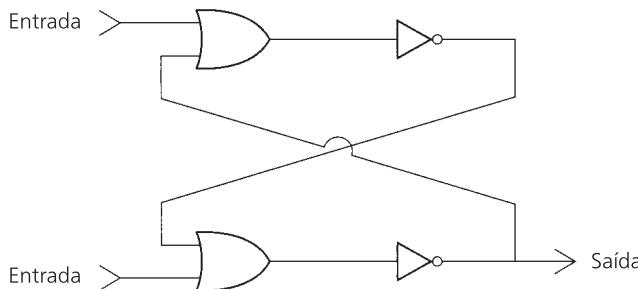


Figura 1.5 Outra forma de construção de um *flip-flop*.

A notação hexadecimal

Quando consideramos as atividades internas de um computador, devemos lidar com cadeias de *bits*, algumas das quais podem ser muito longas. Essa cadeia é freqüentemente chamada corrente (**stream**). Infelizmente, a mente humana tem dificuldades para gerenciar esse nível de detalhe. A simples transcrição do padrão 101101010011 se mostra não apenas tediosa, como muito propensa a erros. Por essa razão, para simplificar a representação de padrões de *bits*, normalmente utilizamos uma notação mais compacta, conhecida como *notação hexadecimal*. Esta notação aproveita o fato de que os padrões de *bits* dentro de um computador apresentam sempre comprimentos múltiplos de quatro. Em particular, a notação hexadecimal utiliza um único símbolo para representar quatro *bits*, ou seja, uma seqüência de doze *bits* pode ser denotada com apenas três símbolos hexadecimais.

A Figura 1.6 apresenta o sistema de codificação hexadecimal. A coluna da esquerda mostra todos os possíveis padrões de *bits* de comprimento quatro, enquanto a da direita apresenta o símbolo utilizado na notação hexadecimal para representar o padrão de *bits* correspondente. Nesta notação, o padrão 10110101 é representado como B5. Isto é obtido dividindo o padrão de *bits* em subcadeias de comprimento quatro e novamente representando cada cadeia pela notação hexadecimal equivalente. Assim, 1011 é representado por B e 0101, por 5. Desta maneira, o padrão 1010010011001000, de 16 *bits*, pode ser reduzido à forma A4C8, bem mais confortável.

Utilizaremos extensivamente a notação hexadecimal no próximo capítulo, no qual se poderá constatar a sua eficácia.

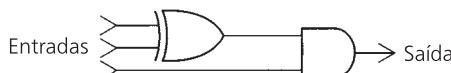
Padrão de bits	Representação hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Figura 1.6 O sistema de código hexadecimal.



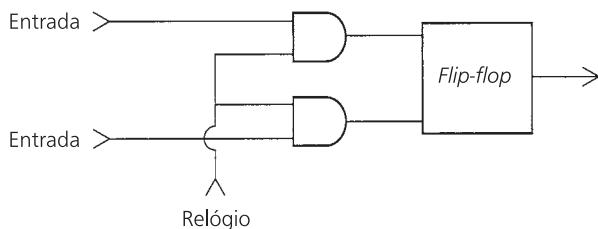
QUESTÕES/EXERCÍCIOS

- Que configurações de *bits* de entrada devem ser fornecidas ao circuito abaixo para que seja produzida uma saída 1?

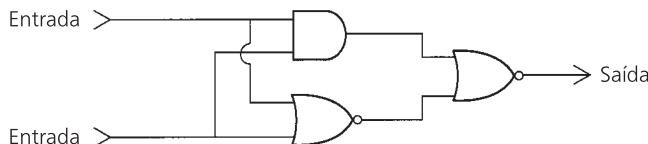


- No texto, afirmamos que a introdução do 1 na entrada inferior do *flip-flop* da Figura 1.3 (mantendo fixo o 0 para a entrada superior) forçará a saída do *flip-flop* a ser 0. Descreva a sucessão de eventos que ocorreram neste caso, dentro do *flip-flop*.

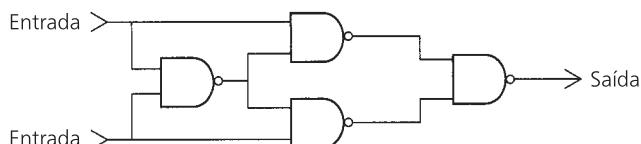
3. Admitindo que as duas entradas do *flip-flop* da Figura 1.5 sejam 0, descreva a sucessão de eventos que ocorrerão quando a entrada superior for temporariamente igualada a 1.
4. Muitas vezes é necessário coordenar atividades em várias partes de um circuito. Isso é feito acoplando um sinal pulsante (chamado *relógio*) às partes do circuito que requerem coordenação. Uma vez que o relógio alterna entre os valores 0 e 1, ele ativa os vários componentes do circuito. A seguir, está um exemplo de uma parte do circuito que envolve o *flip-flop* mostrado na Figura 1.3. Para quais valores do relógio o *flip-flop* ficará isolado dos valores de entrada do circuito? Para quais valores ele responderá aos valores de entrada do circuito?



5. a. Se a saída de uma porta lógica OR passar por uma porta NOT, a combinação executará a operação booleana chamada NOR, cuja saída só será 1 quando as duas entradas forem 0. O símbolo para essa porta lógica é o mesmo da porta OR, exceto em que ele possui um círculo em sua saída. Abaixo, está um circuito que contém uma porta lógica AND e duas portas NOR. Que função booleana ele computa?



- b. Se a saída de uma porta lógica AND passar por uma porta NOT, a combinação executará a operação booleana chamada NAND, cuja saída só será 0 quando as duas entradas forem 1. O símbolo para essa porta lógica é o mesmo da porta AND, exceto em que ele possui um círculo em sua saída. Abaixo, está um circuito que contém portas lógicas NAND. Que função booleana ele computa?



6. Utilize a notação hexadecimal para representar os seguintes padrões de bits:
- 0110101011110010
 - 111010000101010100010111
 - 01001000
7. Que padrões de bits são representados pelos seguintes padrões hexadecimais?
- 5FD97
 - 610A
 - ABCD
 - 0100

1.2 Memória principal

Com a finalidade de armazenar seus dados, um computador contém um conjunto grande de circuitos, cada qual capaz de armazenar um *bit*. A este conjunto denominamos **memória principal** da máquina.

Organização da memória

Os circuitos de armazenamento da memória principal de um computador são organizados em unidades manipuláveis denominadas **células** (ou *palavras*), geralmente com um tamanho de oito *bits* cada uma. De fato, os conjuntos de *bits* de tamanho oito ficaram tão populares que o termo **byte** é amplamente utilizado, nos dias de hoje, para denotá-lo. Computadores pequenos, utilizados em dispositivos domésticos, como fornos de microondas, podem ter suas memórias principais medidas em centenas de células, enquanto computadores de grande porte, utilizados para armazenar e manipular grandes quantidades de dados, podem ter bilhões de células em suas memórias principais.

Embora dentro do computador não tenham sentido os conceitos de lado esquerdo ou direito, normalmente imaginamos os *bits* de uma posição de memória organizados em linha, cuja extremidade esquerda é chamada **extremidade de alta ordem** e a direita, **extremidade de baixa ordem**. O último *bit* da extremidade de alta ordem é conhecido como o *bit* de ordem mais elevada, ou **bit mais significativo** em referência ao fato de que se o conteúdo da célula fosse interpretado como a representação de um valor numérico, este *bit* seria o dígito mais significativo do número. Do mesmo modo, o *bit* da extremidade direita é conhecido como o *bit* de ordem mais baixa, ou **bit menos significativo**. Dessa forma, podemos representar o conteúdo de uma célula de memória de um byte conforme mostra a Figura 1.7.

Para distinguir, na memória principal de um computador, cada célula individual, esta é identificada por um nome único, denominado **endereço**. Esse sistema de endereçamento é similar e emprega a mesma terminologia da técnica de identificação de edificações urbanas, com o auxílio de endereços. No caso das células de memória, contudo, os endereços utilizados são totalmente numéricos. Para ser mais precisos, podemos visualizar as células de memória como que dispostas em uma fila única, cujos elementos são numerados em ordem crescente, a partir do endereço com valor zero. Tal sistema de endereçamento não apenas nos fornece um meio de identificar univocamente cada célula, mas também associa a elas uma ordem (Figura 1.8), o que nos permite utilizar expressões como “a próxima célula” ou “a célula anterior”.

Uma consequência importante da ordenação simultânea das posições da memória principal e dos *bits* dentro de cada célula é que todo o conjunto de *bits* dentro da memória principal de uma máquina fica ordenado, essencialmente, em uma fila única longa. Assim, partes dessa longa fila podem ser utilizadas para armazenar padrões de *bits* que sejam mais longos do que o comprimento de uma única célula. Em particular, se a memória estiver dividida em células com tamanho de um byte cada, ainda assim será possível armazenar uma cadeia de 16 *bits*, simplesmente utilizando duas células consecutivas de memória para isso.

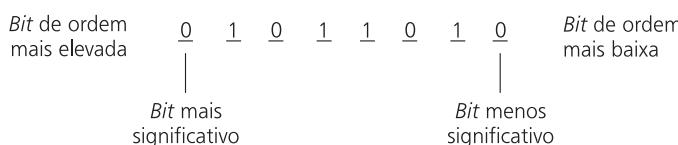


Figura 1.7 A organização de uma célula de memória de um byte de comprimento.

Outra consequência de organizar a memória principal de uma máquina com pequenas células endereçadas é que cada uma pode ser individualmente referenciada. Por conseguinte, dados armazenados na memória principal de uma máquina podem ser processados em qualquer ordem, razão pela qual este tipo de memória é conhecido como **memória de acesso aleatório** (*random access memory — RAM*).

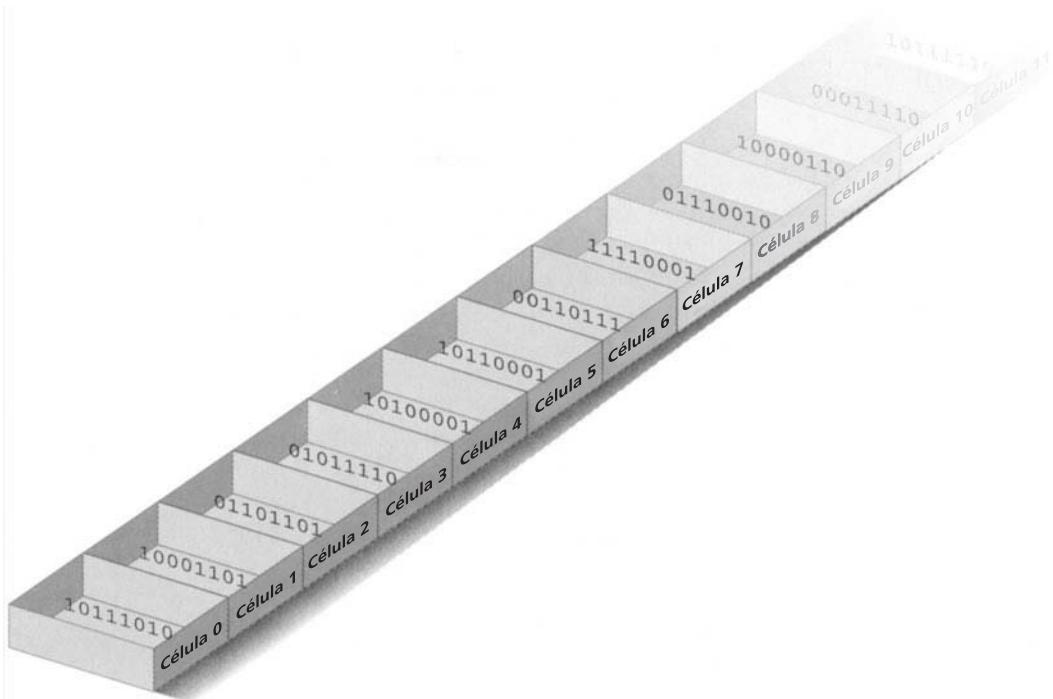


Figura 1.8 Células de memória, ordenadas em ordem crescente de endereços.

Esse acesso aleatório a pequenas porções de dados contrasta significativamente com os sistemas de armazenamento em massa, discutidos na próxima seção, nos quais longas cadeias de *bits* devem ser manipuladas em bloco. Quando a RAM é construída com a tecnologia de memória dinâmica, freqüentemente é chamada de DRAM (“Dynamic RAM”).

Para completar a memória principal de uma máquina, o sistema de circuitos que realmente armazena os *bits* é conectado com outro, encarregado de prover meios para que os demais circuitos possam armazenar dados nas células de memória e, posteriormente, reavê-los. Dessa maneira, outros circuitos ficam habilitados a consultar eletronicamente os dados da memória, recuperando, assim, o conteúdo de um determinado endereço (*read* — operação de leitura), ou então gravar informação na memória, ao solicitar que um padrão de *bits* desejado seja inserido na célula correspondente a um certo endereço (*write* — operação de gravação).

Como medir a capacidade da memória

Como aprenderemos nos próximos capítulos, é conveniente projetar sistemas de memória principal nos quais o número total de células seja uma potência de dois. A propósito, o tamanho da memória dos primeiros computadores era comumente medido em unidades de 1024 (que é 2^{10}) células. Como 1024 fica próximo de 1000, muitos na comunidade da computação adotaram o prefixo *kilo* em referência a essa unidade. Assim, o termo *kilobyte* (abreviado com KB) era usado para se referir a 1024 *bytes*, e dizia-se que uma máquina que tivesse 4096 células de memória tinha 4 KB de memória ($4096 = 4 \times 1024$). À medida que as memórias foram se tornando maiores, essa terminologia cresceu para incluir os prefixos

mega, para 1.048.576 (que é 2^{20}), e *giga*, para 1.073.741.824 (que é 2^{30}), e unidades como MB (*megabyte*) e GB (*gigabyte*) tornaram-se comuns.

Infelizmente, essa aplicação de prefixos representa um mau uso da terminologia, porque os prefixos já eram utilizados em outros campos para referir-se a unidades que são potências de dez. Por exemplo, quando medimos distância, *quilômetro* refere-se a 1.000 metros, e quando medimos rádio-freqüência, *megahertz* refere-se a 1.000.000 de hertz. Além disso, alguns fabricantes de equipamento de computação decidiram adotar o prefixo *mega* para referir-se a unidades de 1.024.000 (logo, um *megabyte* seria 1000 KB). Desnecessário dizer que essas discrepâncias têm levado à confusão e ao desentendimento ao longo dos anos.

Para esclarecer a matéria, uma proposta foi feita a fim de reservar os prefixos *kilo*, *mega* e *giga* para unidades que são potências de dez e introduzir os novos prefixos *kibi*. (abreviatura de *kilobinary*, abreviado com Ki), *mebi* (abreviatura de *megabinary*, abreviada com Mi), e *gibi* (abreviatura de *gigabinary*, abreviado com Gi), em referência às unidades correspondentes que são potências de dois. Nesse sistema, o termo *Kibibyte* se refere a 1024 bytes e *Kilobyte*, a 1000 bytes. Se esses prefixos se tornarão parte do vernáculo popular, só vendo. Por hora, o mau uso dos prefixos *kilo*, *mega* e *giga* continua impregnado na comunidade da computação, de modo que seguiremos essa tradição em nosso estudo. Contudo, os prefixos propostos *kibi*, *mebi* e *gibi* representam uma tentativa de resolver o problema crescente, e devemos interpretar cautelosamente no futuro termos como *kilobyte* e *megabyte*.



QUESTÕES/EXERCÍCIOS

1. Se a posição de memória cujo endereço é 5 contém o valor 8, qual a diferença entre escrever o valor 5 na célula de número 6 e passar o conteúdo da célula de número 5 para a de número 6?
2. Suponha que se queira intercambiar os valores armazenados nas células de memória de endereços 2 e 3. O que há de errado na seqüência de passos descrita a seguir?
Passo 1. Transfira o conteúdo da célula de número 2 para a de número 3.
Passo 2. Transfira o conteúdo da célula de número 3 para a de número 2.
Projete outra seqüência de passos que troque, corretamente, os conteúdos dessas duas células.
3. Quantos bits existem na memória de um computador com 4KB (mais precisamente KiB) de memória?

1.3 Armazenamento em massa

Devido à volatilidade e ao tamanho limitado da memória principal de um computador, a maioria das máquinas possui dispositivos de memória adicional chamados **sistemas de armazenamento em massa** — que incluem discos magnéticos, CDs e fitas magnéticas. As vantagens dos sistemas de armazenamento em massa em relação à memória principal incluem menor volatilidade, maior capacidade de armazenamento e, em muitos casos, a possibilidade de se retirar o meio físico no qual são gravados os dados da máquina com o propósito de arquivamento.

Os termos *on-line* e *off-line* são costumeiramente utilizados para referir-se a dispositivos que estejam, respectivamente, ligados diretamente ao computador, ou dele desconectados. **On-line** significa que o dispositivo ou a informação já está conectado(a) ao computador e prontamente disponível para a máquina, sem a necessidade de uma intervenção humana. **Off-line**, pelo contrário, significa que a intervenção humana é necessária antes que seja possível ao computador fazer os acessos necessários à informação ou ao dispositivo — talvez porque o dispositivo precise ser ativado manualmente ou porque um meio físico que contenha a informação desejada deva ser fisicamente inserido em algum equipamento.

A maior desvantagem dos sistemas de armazenamento em massa é que geralmente requerem movimentação mecânica, portanto seus tempos de resposta são muito maiores quando comparados com os da memória principal da máquina, que executa todas as atividades eletronicamente.

Discos magnéticos

Atualmente, a forma mais usual de armazenamento em massa é o disco magnético. Nele, os dados são armazenados em um fino disco giratório com revestimento de material magnético. Cabeçotes de leitura/gravação (*read/write*) ficam instalados nos dois lados do disco, de forma que, ao girá-lo, cada cabeçote percorra, na superfície superior ou inferior, uma trajetória circular, denominada **trilha** (*track*). Por meio do reposicionamento dos cabeçotes de leitura/gravação, é possível acessar as diversas trilhas concêntricas. Em muitos casos, um sistema de armazenamento em disco consiste em vários discos montados em um eixo giratório comum, um em cima do outro, com espaço suficiente para que o cabeçote de leitura/gravação se desloque entre os discos. Nesses casos, os cabeçotes se movem em uníssono. Cada vez que eles são reposicionados, um novo conjunto de trilhas — chamado **cilindro** — se torna acessível.

Considerando que cada trilha pode conter mais informação do que é necessário manipular de uma única vez, as trilhas são divididas em **setores**, nos quais a informação é registrada como cadeia contínua de *bits* (Figura 1.9). Cada trilha em um sistema de discos contém o mesmo número de setores e cada setor contém o mesmo número de *bits*. (Isso significa que os *bits* de um setor são armazenados mais compactados nas trilhas perto do centro do disco do que nas trilhas perto da borda.)

Assim, um sistema de armazenamento em disco consiste em muitos setores individuais, cada qual acessível na forma de uma cadeia independente de *bits*. O número de trilhas por superfície e o número de setores por trilha variam muito de um sistema de discos para outro. O tamanho dos setores tende a assumir valores não maiores que alguns KB; setores de 512 bytes ou 1024 bytes são comuns.

A localização física das trilhas e dos setores não constitui parte permanente da estrutura física de um disco. Ao contrário, eles são marcados magneticamente, como resultado de um processo denominado **formatação** (ou inicialização) do disco. Este processo em geral é efetuado pelo fabricante de discos, e resulta nos chamados discos formatados. A maioria dos sistemas computacionais tem a capacidade de executar essa tarefa. Assim, se a informação de formato em um disco for estragada, ele poderá ser reformatado, embora esse processo destrua toda informação que estava previamente gravada no disco.

A capacidade de um sistema de armazenamento em disco depende do número de superfícies magnéticas nele disponíveis e da densidade das suas trilhas e setores. Os sistemas de capacidade mais baixa consistem em um único disco de plástico conhecido como disquete ou, se for flexível, pelo título de disco flexível (*floppy disk*), de menor prestígio. (Os discos flexíveis atuais com diâmetro de 3½ polegadas são protegidos com capa de plástico rígido, não sendo, portanto, tão flexíveis quanto seus antigos primos com diâmetro de 5¼ polegadas e capa de papel.) Os disquetes podem ser facilmente inseridos na respectiva unidade de leitura/gravação e dela removidos, bem como facilmente armazenados. Conseqüentemente, representam um bom meio de armazenamento de informação *off-line*. O disquete usual de 3½ polegadas é capaz de armazenar 1,44 MB, mas existem disquetes com capacidades muito maiores. Um exemplo é o sistema de discos Zip da Iomega Corporation, com capacidade de centenas de MB em um único disquete rígido.

Os sistemas de disco de alta capacidade de armazenamento, capazes de



Figura 1.9 Um sistema de armazenamento em disco.

comportar vários gigabytes, consistem em cinco a dez discos rígidos, montados em uma coluna comum. Pelo fato de tais discos serem feitos de material rígido, são conhecidos como sistemas de disco rígido (*hard disk*), em contraste com os flexíveis (*floppy disk*). Para permitir uma velocidade maior de rotação nesses sistemas, seu cabeçote de leitura/gravação não chega a ter contato físico com o disco, mas apenas se mantém flutuando pouco acima da superfície do disco em movimento. A distância entre o cabeçote e a superfície do disco é tão pequena que qualquer partícula de pó que se interponha entre eles causará danos a ambos (fenômeno de danificação do cabeçote, conhecido como *head crash*). Por essa razão, os sistemas de disco rígido são montados em recipientes lacrados na fábrica.

São utilizadas várias medidas para avaliar o desempenho de um sistema de disco: (1) **tempo de busca** (tempo necessário para mover os cabeçotes de leitura/gravação de uma trilha para outra); (2) **atraso de rotação** ou **tempo de latência** (metade do tempo utilizado para o disco executar uma rotação completa, que é o tempo médio necessário para os dados necessários chegarem debaixo do cabeçote de leitura/gravação, uma vez que este tenha sido posicionado na trilha desejada); (3) **tempo de acesso** (soma do tempo de busca e do atraso de rotação); (4) **tempo de transferência** (velocidade com que os dados são transferidos do disco para o computador ou vice-versa).

Em geral, os sistemas de discos rígidos apresentam características significativamente melhores que os de discos flexíveis. Considerando que os cabeçotes de leitura/gravação não tocam a superfície do disco em um sistema de disco rígido, é possível operá-lo a velocidades da ordem de 3000 a 4000 rotações por minuto, enquanto nos discos flexíveis a velocidade viável é da ordem de 300 rotações por minuto. Por conseguinte, as taxas de transferência para sistemas de disco rígido, normalmente medidas em megabytes por segundo, são muito maiores do que as de sistemas de disco flexível, geralmente medidas em quilobytes por segundo.

Considerando que a operação dos sistemas de disco requer movimento físico, tanto os discos rígidos como os flexíveis perdem para os circuitos eletrônicos quando comparados em matéria de velocidade. De fato, os tempos de espera no interior de um circuito eletrônico são medidos em nanosegundos (bilionésimos de segundo) ou menos, sendo que os tempos de busca, de latência e de acesso aos sistemas de disco são medidos em milissegundos (milésimos de segundo). Assim, o tempo necessário para obter dados de um sistema de disco pode parecer uma eternidade para um circuito eletrônico à espera de algum resultado.

Discos óticos (*compact disks*)

Outra tecnologia popular de armazenamento em disco é o disco compacto (CD). Esses discos possuem 12 cm (aproximadamente 3 polegadas) de diâmetro e são feitos de material reflexivo, recoberto com uma camada protetora transparente. A informação é gravada neles criando variações na superfície reflexiva. Essa informação pode então ser recuperada por meio de um facho de raios *laser* que monitora as irregularidades da superfície refletiva do CD enquanto ele gira.

A tecnologia do CD foi originalmente aplicada a gravações de áudio, usando um formato de gravação conhecido como CD-DA (*compact disk-digital audio*), e os CDs usados hoje em dia para armazenamento de dados do computador usam essencialmente o mesmo formato. A informação nesses CDs específicos é gravada em uma única trilha disposta em espiral, como o sulco dos antigos discos musicais de vinil (Figura 1.10). (Diferentemente dos primeiros, a espiral em um CD vai do centro para fora). Essa trilha é dividida em unidades chamadas setores, e cada qual contém marcações de identificação e 2 KB disponíveis para os dados, que equivalem a 1/75 de segundo de música nas gravações de áudio.

Note que as distâncias ao longo da espiral são maiores perto da borda do que nas partes interiores. Para maximizar a capacidade de um CD, a informação é gravada com densidade linear uniforme na trilha inteira, o que significa que mais informação é guardada em uma volta na parte mais externa da espiral do que em uma volta na parte mais interna. Como consequência, mais setores são lidos em uma única rotação do disco quando o facho de *laser* está percorrendo a parte mais externa da trilha do que quando percorre a mais interna. Assim, para obter uma taxa de transferência de dados uniforme, os aparelhos de CD são projetados para variar a velocidade de rotação, dependendo da localização do facho de *laser*.

Como consequência dessa decisão de projeto, os sistemas de armazenamento com CD têm um melhor desempenho quando lidam com longas e contínuas cadeias de dados, como é o caso quando reproduzem música. Entretanto, quando uma aplicação exige acesso a itens de dados de uma maneira aleatória, a abordagem usada em armazenamento com discos magnéticos (trilhas individuais, concêntricas, sendo que cada uma contém o mesmo número de setores) supera em muito a abordagem em espiral usada nos CDs.

Os CDs tradicionais têm capacidade de armazenamento em torno de 600 a 700 MB. Contudo, novos formatos, como o DVD (*digital versatile disk*) atingem capacidades na ordem de 10 GB. Tais CDs podem armazenar apresentações de multimídia nas quais os dados de áudio e vídeo são combinados para apresentar a informação de uma maneira mais interessante e informativa do que a obtida apenas com texto. De fato, a principal aplicação do DVD é como meio de gravação de um filme inteiro em um único CD.

Fita magnética

Outra forma, mais antiga, de armazenamento em massa é a fita magnética (Figura 1.11). Aqui, a informação é registrada sobre uma película de material magnético, que recobre uma fita de plástico fina, enrolada sobre um carretel. Para se ter acesso aos dados, esta fita deve ser montada em um dispositivo, denominado unidade de fita, o qual tem a possibilidade de efetuar operações de leitura, gravação e rebobinamento da fita, sob controle do computador. As unidades de fita variam em tamanho, desde as pequenas unidades de cartucho, ou fitas *streaming*, que utilizam fitas semelhantes às dos gravadores de áudio, até as grandes unidades antigas, em que o transporte da fita se fazia de carretel para carretel. Embora a capacidade desses dispositivos dependa do formato utilizado na gravação, alguns podem comportar um volume de dados da ordem de vários gigabytes.

Os sistemas modernos dividem a fita em segmentos, cada qual magneticamente gravado por meio de um processo de formatação, semelhante ao dos dispositivos de armazenamento em disco. Cada segmento contém várias trilhas, que correm paralelamenteumas às outras, ao longo da fita. O acesso a essas trilhas pode ser feito independentemente, ou seja, a fita consiste, em última instância, em numerosas e independentes cadeias de *bits*, de forma similar ao que ocorre nos setores em um disco.



Figura 1.10 Formato de armazenamento do CD.

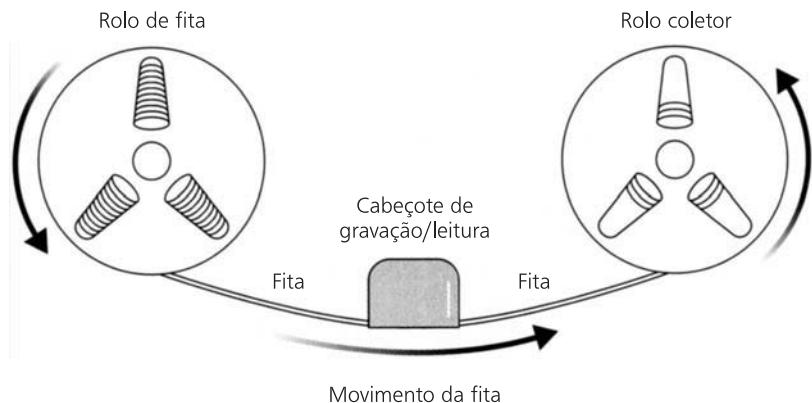


Figura 1.11 Um mecanismo de armazenamento em fita magnética.

A maior desvantagem dos sistemas de fita é o fato de que a movimentação para posicionamento, entre posições diferentes de uma fita, pode ser muito demorada, devido à grande quantidade de fita que deve ser deslocada entre os carretéis. Assim, os sistemas de fita apresentam maiores tempos de acesso de dados do que os sistemas de disco, nos quais setores diferentes podem ser acessados por meio de pequenos movimentos do cabeçote de leitura/gravação. Assim, os sistemas de fita não são adequados para armazenamento de dados *on-line*. Contudo, quando o objetivo é o armazenamento *off-line* com o propósito de arquivamento (*back-up*), a alta capacidade e confiabilidade e o baixo custo da fita tornam essa tecnologia uma escolha popular entre os sistemas de armazenamento de dados atuais.

Armazenamento e recuperação de arquivos

A informação é armazenada nos sistemas de armazenamento em massa em grandes unidades chamadas **arquivos**. Um arquivo típico pode consistir em um documento completo sob a forma de texto, uma fotografia, um programa, ou uma coleção de dados a respeito dos funcionários de uma companhia. As características físicas dos dispositivos de armazenamento em massa ditam que esses arquivos devem ser armazenados e recuperados em unidades com múltiplos bytes. Por exemplo, cada setor de um disco magnético deve ser obrigatoriamente tratado como uma longa cadeia de bits. Um bloco de dados dimensionado com base nas características físicas de um dispositivo de armazenamento recebe a denominação de **registro físico**. Assim, um arquivo guardado em um sistema de armazenamento em massa geralmente consiste em muitos registros físicos.

Ao contrário do particionamento dos dados em registros físicos cujos tamanhos são impostos pelas características físicas dos dispositivos de armazenamento, o arquivo a ser armazenado normalmente apresenta suas divisões naturais específicas. Assim, é conveniente que um arquivo que contenha informação relativa aos funcionários de uma companhia seja dividido em blocos, cada qual com a informação relativa a determinado funcionário. Tais agrupamentos naturais de dados, um para cada funcionário, são denominados **registros lógicos**.

Raramente os comprimentos dos registros lógicos combinam exatamente com os dos registros físicos em um dispositivo de armazenamento em massa. Por essa razão, pode-se encontrar vários registros lógicos em um único registro físico. Entretanto, também é possível que um registro lógico se divida em dois ou mais registros físicos (Figura 1.12). Por conseguinte, um certo desembaraçamento freqüentemente é associado à representação de dados provenientes de sistemas de armazenamento em massa.

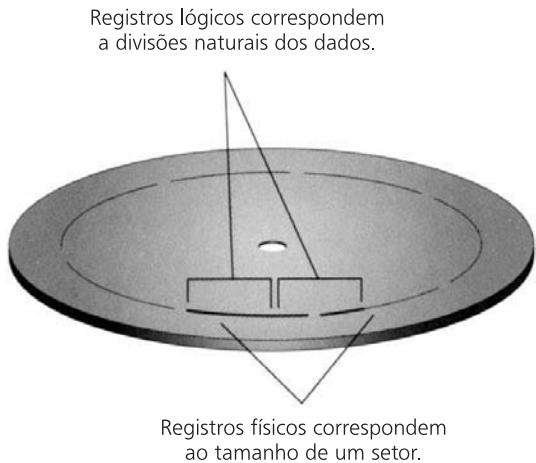


Figura 1.12 Registros lógicos e registros físicos em um disco.

Uma solução comum a esse problema é reservar uma área de memória principal grande o suficiente para guardar vários registros físicos e usar esse espaço de memória como área de agrupamento. Em outras palavras, blocos de dados compatíveis com os registros físicos podem ser transferidos entre a memória principal e o sistema de armazenamento em massa, enquanto os dados residentes na memória principal podem ser referenciados em termos de registros lógicos.

Uma área de memória usada dessa maneira é chamada **retentor** (*buffer*). Em geral, um retentor é uma área de armazenamento para montar os dados temporariamente, em geral, durante o processo de transferência de um dispositivo para outro. Por exemplo, as impressoras modernas contêm circuitos de memória cuja maioria é usada como retentor para montar partes de um documento que foi transferido para a impressora mas ainda não foi impresso.

O uso de um retentor no contexto da transferência de dados entre um dispositivo de armazenamen-

to em massa e a memória principal de um computador exemplifica os papéis relativos da memória principal e do armazenamento em massa. A memória principal é usada para reter os dados a serem processados, enquanto o armazenamento em massa serve como depósito permanente para eles. Assim, a atualização de dados de um sistema de armazenamento em massa envolve a sua transferência para a memória principal, onde são feitas a atualização propriamente dita e a transferência dos dados atualizados de volta para o sistema.

Concluímos que a memória principal, os discos magnéticos, os CDs e as fitas magnéticas exibem graus decrescentes de acesso aleatório aos dados. O sistema de endereçamento usado na memória principal permite acesso rápido e aleatório a *bytes* individuais de dados. Os discos magnéticos permitem acesso aleatório somente a setores inteiros de dados. Além disso, a leitura de um setor envolve o tempo de busca e o atraso de rotação. Os discos óticos também permitem acesso aleatório aos setores individuais, mas os atrasos aí são maiores que os encontrados nos discos magnéticos, devido ao tempo adicional exigido para o posicionamento na trilha em espiral e o ajuste de velocidade de rotação do disco. Finalmente, as fitas magnéticas oferecem pouco em termos de acesso aleatório. Os sistemas de fita modernos marcam posições na fita de tal forma que diferentes segmentos da mesma possam ser referenciados individualmente, mas a estrutura física da fita implica tempos significativos para recuperar um segmento distante.



QUESTÕES/EXERCÍCIOS

1. Quais as vantagens que um sistema de disco rígido apresenta pelo fato de seus discos girarem mais rapidamente que os discos flexíveis?
2. Quando se gravam dados em um sistema de armazenamento com múltiplos discos, devemos preencher completamente uma superfície do disco antes de utilizar uma outra, ou primeiro encher um cilindro inteiro antes de iniciar a gravação em outro cilindro?
3. Por que devem ser armazenados em disco, e não em fita, os dados referentes a um sistema de reservas que seja atualizado com muita freqüência?
4. Algumas vezes, quando modificamos um documento usando um processador de textos, ao se adicionar texto, o tamanho aparente do arquivo no disco não aumenta, mas em outras vezes a adição de um único símbolo pode aumentar o tamanho aparente do arquivo em várias centenas de *bytes*. Por quê?

1.4 Representação da informação como padrões de *bits*

Uma vez consideradas as técnicas para armazenar *bits*, agora consideramos como a informação pode ser codificada como padrões de *bits*. Nosso estudo vai focalizar os métodos populares para codificação de texto, dados numéricos, imagens e sons. Cada um desses sistemas tem repercussões que normalmente são visíveis a um usuário típico de computador. Nosso objetivo é entender suficientemente essas técnicas, de modo a podermos reconhecer as suas reais consequências.

Representação de texto

A informação na forma de texto normalmente é representada por meio de um código, no qual se atribui a cada um dos diferentes símbolos do texto (letras do alfabeto e caracteres de pontuação) um único padrão de *bits*. O texto fica então representado como uma longa cadeia de *bits*, na qual padrões sucessivos representam os símbolos sucessivos no texto original.

American Standard National Institute

O American Standard National Institute (ANSI) foi fundado em 1918 por um pequeno consórcio formado por sociedades de engenharia e agências governamentais como organização sem fins lucrativos para coordenar o desenvolvimento de padrões espontâneos no setor privado. Atualmente, entre os membros do ANSI, incluem-se mais de 1300 empresas, organizações profissionais, associações comerciais e agências governamentais. O ANSI está localizado em Nova York e representa os EUA como membro do ISO. Seu sítio na Web está no endereço <http://www.ansi.org>.

Organizações similares em outros países incluem Standards Australia (Austrália), Standards Council of Canada (Canadá), Chinese State Bureau of Quality and Technical Supervision (China), Deutsches Institut für Normung (Alemanha), Japanese Industrial Standards Committee (Japão), Dirección General de Normas (México), State Committee of the Russian Federation for Standardization and Metrology (Rússia), Swiss Association for Standardization (Suíça) e British Standards Institution (Reino Unido).

demonstra que, neste sistema, o padrão de bits

01001000 01100101 01101100 01101100 01101111 00101110

representa a palavra “Hello.”

ISO — A International Organization for Standardization

A International Organization for Standardization (comumente chamada ISO) foi criada em 1947 como federação mundial de organismos de padronização, cada qual de um país. Atualmente, sua sede fica em Genebra, na Suíça, e possui mais de 100 organizações-membros, bem como numerosos membros correspondentes. (Um membro correspondente geralmente é de um país que não tem um organismo de padronização reconhecido nacionalmente. Esses membros não podem participar diretamente do desenvolvimento de padrões, mas são mantidos informados das atividades da ISO.) A ISO mantém um sítio na Web no endereço <http://www.iso.ch>.

Nos primórdios da computação, muitos códigos diferentes foram projetados e utilizados em associação com diferentes partes do equipamento computacional, acarretando a correspondente proliferação de problemas de comunicação. Para aliviar essa situação, o **American National Standard Institute** (ANSI) adotou o **American Standard Code for Information Interchange** (ASCII, pronunciado como *asc-i-i*). Este código utiliza padrões de sete bits para representar letras maiúsculas e minúsculas do alfabeto, símbolos de pontuação, os dígitos de 0 a 9 e certas informações de controle, como mudanças de linha (*line feed*), posicionamento no início de uma linha (*carriage return*) e tabulações (*tab*). Hoje em dia, o código ASCII tem sido, muitas vezes, estendido para um formato de oito bits por símbolo, mediante a inserção de um 0 (zero) adicional, como **bit** mais significativo, em cada padrão de sete bits. Esta técnica não apenas gera um código cujos padrões se ajustam perfeitamente a uma célula de um byte de memória, como permite 128 padrões adicionais de bits (obtidos atribuindo-se o valor 1 (um) ao **bit** extra) e a representação de símbolos que não figuram no código ASCII original. Infelizmente, como cada fabricante tende a dar sua própria interpretação a esses padrões adicionais de bits, os dados em que tais padrões figuram não costumam apresentar fácil portabilidade entre programas de fabricantes diferentes.

O apêndice A mostra uma parte do código ASCII, em um formato de oito bits por símbolo, e a Figura 1.13

Embora o ASCII tenha sido o código mais utilizado durante muitos anos, estão ganhando popularidade outros códigos de maior alcance, capazes de representar documentos em diversos idiomas. Um deles, o **Unicode**, foi desenvolvido por alguns dos mais proeminentes fabricantes de *hardware* e *software* e está rapidamente ganhando apoio na comunidade da computação. Esse código usa um padrão único de 16 bits para representar cada símbolo. Por conseguinte, o Unicode consiste em 65.536 diferentes padrões de bits — o suficiente para representar os símbolos mais comuns dos idiomas chineses e japoneses. Um código que provavelmente competirá com o Unicode foi desenvolvido pela **International Organization for Standardization** (também conhecida como **ISO**, em alusão à palavra grega *isos*, que significa *igual*). Utilizando padrões de 32 bits, esse sistema de codificação tem a capacidade de representar bilhões de símbolos.

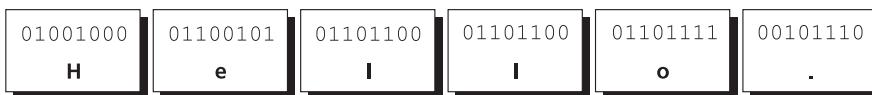


Figura 1.13 A mensagem “Hello.” em ASCII.

Representação de valores numéricos

Embora seja muito útil o método de armazenar informação na forma de caracteres codificados, ele é ineficiente quando a informação a ser registrada é puramente numérica. Para ilustrar, suponha que desejemos armazenar o número 25. Se insistirmos em armazená-lo na forma de símbolos codificados em ASCII, utilizando um byte por símbolo, precisaremos de um total de 16 bits. Além disso, 99 é o maior número que podemos armazenar desta forma, utilizando 16 bits. Um modo mais eficiente de efetuar esse armazenamento seria representar em base dois, ou seja, em notação binária, o valor numérico desejado.

A forma binária permite representar valores numéricos mediante o uso exclusivo dos dígitos 0 e 1, em vez dos dez dígitos 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9 do sistema decimal tradicional. Sabe-se que, na base dez, cada posição, na denotação do número, está associada com uma quantidade. Na representação numérica de 375, o dígito 5 está localizado na posição associada às unidades, o dígito 7, na posição associada às dezenas e o dígito 3, na associada às centenas (Figura 1.14a). O peso correspondente a cada posição é dez vezes maior do que o associado à posição imediatamente à sua direita. O valor representado pela expressão completa é obtido multiplicando-se o valor representado em cada dígito pela quantidade associada à sua posição relativa, e somando-se todos esses resultados intermediários. No exemplo dado, o padrão 375 representará, portanto, $(3 \times \text{cem}) + (7 \times \text{dez}) + (5 \times \text{um})$.

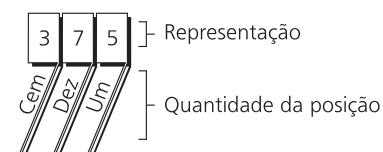
Na notação binária, a posição de cada dígito de uma representação numérica também está associada a uma quantidade. O peso associado a cada posição é o dobro do peso associado à posição imediatamente à sua direita. Mais precisamente, o dígito mais à direita está associado ao peso *um* (2^0); a posição imediatamente à sua esquerda ao peso *dois* (2^1); a seguinte, ao peso *quatro* (2^2); a próxima, a *oito* (2^3), e assim por diante. Por exemplo, seja a seqüência 1011. Em base binária, o dígito mais à direita, 1, está na posição associada ao peso *um*, o 1 à sua esquerda está na posição associada ao peso *dois*, o 0 seguinte, ao peso *quatro*, e o dígito 1 mais à esquerda, a *oito* (Figura 1.14b).

Para recuperar o valor de uma representação binária, seguimos o mesmo procedimento visto para a base dez — multiplicamos o valor de cada dígito pela quantidade associada à sua posição e somamos todos os resultados intermediários assim obtidos. Por exemplo, o valor representado por 100101 é 37, como mostra a Figura 1.15. Note-se que, como a notação binária só utiliza os dígitos 0 e 1, o processo de multiplicação e soma se reduz apenas à adição das quantidades associadas às posições ocupadas por dígitos 1 na representação. Assim, o padrão binário 1011 representa o valor onze, porque os dígitos 1 figuram nas posições associadas aos pesos *um*, *dois* e *oito*.

Note-se que a seqüência de representações binárias, obtida ao contar de zero a oito, é a seguinte:

0
1
10

a. O sistema de base dez



b. O sistema de base dois

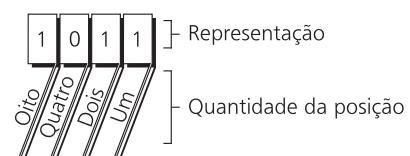


Figura 1.14 Os sistemas decimal e binário.

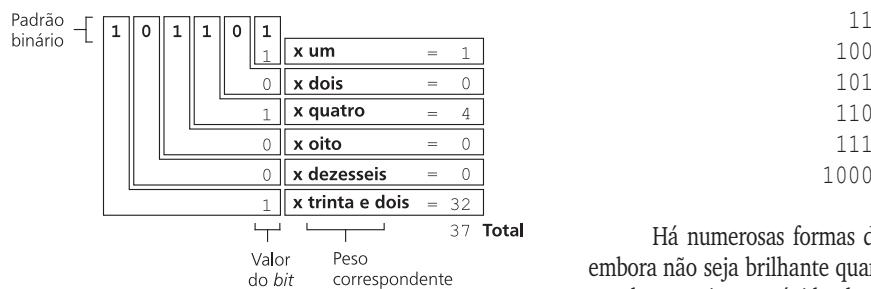


Figura 1.15 Decodificação da representação binária 100101.

nado em 0 e passa a mostrar o dígito 1 conforme o automóvel se desloca. Em seguida, quando este 1 retorna novamente para 0, na posição à sua esquerda no mostrador, deverá aparecer o dígito 1, produzindo-se desta maneira o padrão 10. Em seguida, o dígito 0 mais à direita evolui para 1, produzindo 11. Prosseguindo, este dígito 1, da posição mais à direita do mostrador, volta outra vez para 0, provocando uma alteração no dígito à sua esquerda, fazendo-o passar novamente de 1 para 0. Isto provoca, de modo semelhante, o aparecimento de outro 1 na posição mais à esquerda, produzindo assim o padrão 100.

Para encontrar as representações binárias de valores grandes, é preferível adotar uma forma mais sistemática, como a descrita pelo algoritmo da Figura 1.16. Apliquemos este algoritmo para o valor treze (Figura 1.17). Primeiro, dividimos o número *treze* por *dois*, obtendo *seis* como quociente e *um* como resto. Como o quociente não é nulo, o passo seguinte é dividi-lo por *dois*, obtendo outro quociente, igual a *três*, e o novo resto, igual a *zero*. O novo quociente ainda não é *zero*, assim temos de dividi-lo novamente por *dois*, obtendo um quociente e um resto iguais a *um*. Mais uma vez, dividimos o novo quociente (*um*) por *dois* e, desta vez, obtemos um quociente igual a *zero*, e o resto igual a *um*. Agora, tendo obtido um quociente nulo, passamos ao passo 3 do algoritmo, em que concluímos que a representação binária do valor original (*treze*) é 1101.

Retomemos agora o problema de armazenar o valor 25, com o qual começamos esta subseção. Como vimos, seriam necessários dois *bytes* para armazenar o valor usando um código ASCII por *byte*, e o maior valor que poderíamos armazenar nesses dois *bytes* seria 99. Contudo, se usarmos a notação binária, poderemos armazenar qualquer número inteiro na faixa de 0 a 65.535 em apenas dois *bytes*, o que é uma melhora fantástica.

Por essas e outras razões, é comum representar a informação numérica por meio de uma forma de notação binária, em vez de símbolos codificados. Dizemos uma “forma de notação binária” porque

o sistema binário simples descrito anteriormente serve como base para diversas técnicas de representação de números em computadores. Algumas dessas variantes do sistema binário são discutidas adiante, neste capítulo. Por ora, devemos notar apenas que um sistema, conhecido como notação de complemento de dois, é comum para representar números inteiros por oferecer uma forma conveniente de representação para números negativos e positivos. Para a representação de números com partes fracionárias, como $4\frac{1}{2}$ ou $\frac{3}{4}$, outra técnica, chamada notação de vírgula flutuante, é utilizada. Assim, um valor específico (como 25) pode ser representado por meio de diferentes padrões de *bits* (em caracteres codificados que usem ASCII, em complemento de dois, ou em notação de vírgula flutuante, como

- Passo 1.** Divida o valor a representar por dois e armazene o resto.
- Passo 2.** Enquanto o quociente não for zero, continue dividindo o quociente mais recente por dois e armazene o resto.
- Passo 3.** Agora que o quociente é zero, a representação binária do valor original poderá ser obtida concatenando-se, da direita para a esquerda, na ordem em que foram calculados, os restos das divisões realizadas no passo 2.

Figura 1.16 Um algoritmo para encontrar a representação binária de um inteiro positivo.

$25\%_2$); de maneira recíproca, a cada padrão de bits, diversas interpretações também podem ser associadas.

Encerrando esta seção, devemos mencionar um problema dos sistemas de armazenamento numérico a ser estudado adiante em maior profundidade. Independentemente do tamanho do padrão de bits que uma máquina possa alocar para o armazenamento e valores numéricos, sempre haverá valores excessivamente grandes ou frações demasiadamente pequenas para que possam ser representados no espaço disponível. Decorre um risco permanente da ocorrência de erros de representação, tais como estouro (*overflow*) — valores muito grandes e o truncamento (frações muito pequenas), que deve ser levado em conta para evitar que um usuário desavisado venha a se deparar com miríades de dados incorretos em seus programas.

Representação de imagens

As aplicações dos computadores modernos envolvem não somente caracteres e dados numéricos, mas também figuras, áudio e vídeo. Em comparação com os sistemas de armazenamento restritos a caracteres e dados numéricos, as técnicas para a representação de dados nessas formas adicionais estão em sua infância e, portanto, ainda não suficientemente padronizadas na comunidade de processamento de dados.

As técnicas populares para representar imagens podem ser classificadas em duas categorias: **mapas de bits** e **vetoriais**. No caso das técnicas de mapas de bits, uma imagem é considerada uma coleção de pontos, cada qual chamado **pixel**, abreviatura de *picture element* (“elemento de imagem”). Em sua forma mais simples, uma imagem é representada por uma longa cadeia de bits, os quais representam as linhas de pixel da imagem, onde cada **bit** é 1 ou 0, dependendo de o correspondente *pixel* ser preto ou branco. As imagens coloridas são ligeiramente mais complicadas, uma vez que cada *pixel* pode ser representado por uma combinação de bits que indicam a sua cor. Quando as técnicas de mapas de bits são usadas, o padrão de bits resultante é chamado mapa de bits, o que significa que o padrão de bits é pouco mais que um mapa da imagem que está sendo representada.

A maioria dos periféricos dos computadores modernos, tais como aparelhos de fax, câmeras de vídeo e digitalizadores de imagem, convertem imagens coloridas na forma de mapas de bits. Esses dispositivos normalmente registram a cor de cada *pixel* em três componentes — um vermelho, um verde e um azul — que correspondem às cores primárias. Um byte é usado para representar a intensidade de cada componente de cor. Assim, três bytes de armazenamento são necessários para representar um único *pixel* da imagem original. Essa abordagem de três componentes por *pixel* também corresponde à maneira como a maioria dos monitores de computador exibe imagens. Esses dispositivos exibem uma miríade de pixels, em que cada um consiste em três componentes — um vermelho, um verde e um azul — como pode ser observado inspecionando-se a tela de perto. (Você pode preferir usar uma lente de aumento.)

O formato dos três bytes por *pixel* significa que uma imagem que consiste em 1024 linhas de 1024 pixels (uma fotografia comum) exige vários megabytes para ser armazenada, o que excede a capacidade dos discos flexíveis comuns. Na Seção 1.8, consideraremos duas técnicas populares (GIF e JPEG) usadas para comprimir essas imagens em tamanhos mais manejáveis.

Uma desvantagem das técnicas de mapas de bits é que uma imagem não pode ser facilmente ampliada ou reduzida para qualquer tamanho. Com efeito, o único modo de ampliar uma imagem é aumentar o tamanho dos pixels, o que leva a uma aparência granulada — um fenômeno que também ocorre nas fotografias baseadas em filme. As técnicas vetoriais proporcionam os meios de resolver esses

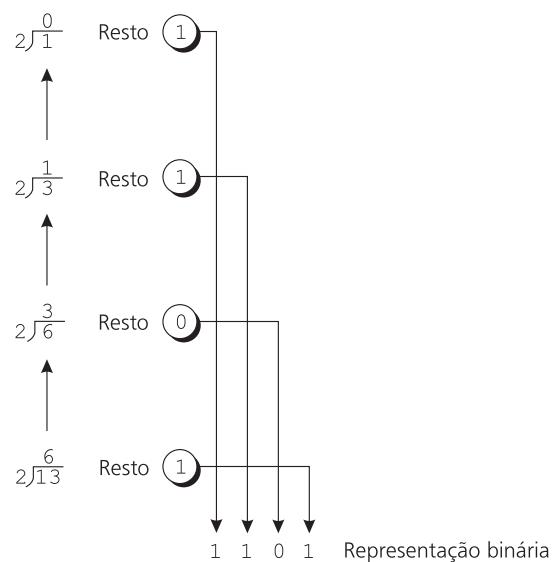


Figura 1.17 Aplicação do algoritmo da Figura 1.15 para obter a representação binária do número 13.

problemas de escala. Em tais sistemas, uma imagem é representada por uma coleção de linhas e curvas. Essa descrição deixa os detalhes de como as linhas e curvas são desenhadas para o dispositivo que efetivamente produz a imagem, em vez de insistir em que o dispositivo reproduza um padrão particular de pixels. As várias fontes de caracteres disponíveis nas impressoras atuais e nos monitores normalmente são codificadas dessa maneira para permitir maior flexibilidade no tamanho do caractere, o que resulta em **fontes em escala** (*scalable fonts*). Por exemplo, o TrueType (desenvolvido pela Microsoft e pela Apple Computer) é um sistema projetado para descrever a maneira como os símbolos utilizados nos textos podem ser desenhados. Do mesmo modo, o Postscript (desenvolvido pela Adobe Systems) fornece meios de descrever não apenas caracteres, mas também outros dados pictóricos mais gerais. As representações vetoriais também são populares nos sistemas de projeto assistido por computador (CAD), onde os desenhos das linhas correspondentes a objetos em três dimensões são exibidos e manipulados na tela do computador. Contudo, as técnicas vetoriais não conseguem a qualidade fotográfica que se obtém com os mapas de bits. Isso explica porque as técnicas de mapas de bits são utilizadas nas câmeras digitais modernas.

Representação de som

O modo mais geral de codificar a informação de áudio para armazená-la e manipulá-la no computador é tirar uma amostragem da amplitude da onda de som em intervalos regulares e registrar a série de valores obtidos. Por exemplo, a série 0; 1,5; 2,0; 1,5; 2,0; 3,0; 4,0; 3,0; 0 pode representar uma onda de som que cresce em amplitude, cai brevemente, aumenta para um nível mais alto e cai para zero. (Figura 1.18). Essa técnica, que usa uma taxa de amostragem de 8.000 amostras por segundo, tem sido usada há anos nas comunicações telefônicas de longa distância. A voz, em uma extremidade da linha, é codificada sob a forma de valores numéricos que representam a amplitude do som a cada oito milésimos de segundo. Esses valores numéricos são então transmitidos pela linha à extremidade receptora, onde são usados para reproduzir o som da voz.

Embora 8.000 amostras por segundo possa parecer uma taxa rápida, ela não é suficiente para a gravação musical com alta fidelidade. Para obter a qualidade de reprodução do som conseguida nos CDs atuais, é usada uma taxa de 44.100 amostras por segundo. Os dados obtidos em cada amostra são representados com 16 bits (32 bits para gravações estereofônicas). Consequentemente, cada segundo de música gravada em estéreo exige mais de um milhão de bits.

Uma alternativa, o sistema de codificação mais econômico, conhecido como *Musical Instrument Digital Interface* (MIDI, pronuncia-se “midi”) é largamente utilizado nos sintetizadores encontrados nos teclados eletrônicos, nos equipamentos de jogos e nos efeitos sonoros que acompanham os sítios na Web. Codificando as diretrizes para produzir música em um sintetizador em vez de codificar o som produzido, o MIDI evita a grande quantidade de armazenamento necessária à técnica de amostragem. Mais precisamente, o MIDI codifica qual instrumento deve tocar qual nota por quanto tempo, o que significa que uma clarineta tocando a nota Ré por dois se-

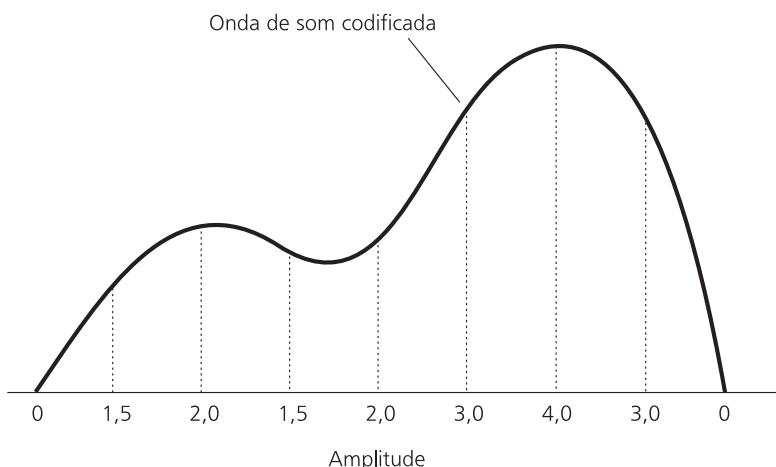


Figura 1.18 A onda de som representada pela seqüência.

gundos pode ser codificada em três *bytes* em vez de mais de dois milhões de *bits* necessários se ela for gravada com taxa de 44.100 amostras por segundo.

Em resumo, o MIDI pode ser entendido como um meio de codificar a partitura de uma música em vez de codificar a música. Assim, uma “gravação” MIDI pode ser significativamente diferente quando tocada em sintetizadores diferentes.



QUESTÕES/EXERCÍCIOS

- 1.** O que está escrito na mensagem abaixo, codificada em ASCII, com oito *bits* por símbolo?
01000011 01101111 01101101 01110000 01110101 01110100
01100101 01110010 00100000 01010011 01100011 01101001
01100101 01101110 01100011 01100101
- 2.** No código ASCII, qual a relação entre o código de uma letra maiúscula e o da mesma letra, porém minúscula?
- 3.** Codifique as sentenças abaixo em ASCII:
 - a. *Where are you?**
 - b. *“How?” Cheryl asked.***
 - c. $2 + 3 = 5$.
- 4.** Descreva um dispositivo do cotidiano que possa assumir um de dois estados, como, por exemplo, uma bandeira em um mastro, que pode estar hasteada ou arriada. Associe o símbolo 1 a um dos estados e 0 ao outro e mostre o aspecto do código ASCII para a letra *b* quando representado com tais *bits*.
- 5.** Converta os seguintes códigos binários na forma decimal:

a. 0101	b. 1001	c. 1011	d. 0110
e. 10000	f. 10010		
- 6.** Converta as seguintes representações decimais em código binário:

a. 6	b. 13	c. 11	d. 18
e. 27	f. 4		
- 7.** Qual o maior valor numérico que poderá ser representado com três *bytes* se cada dígito for codificado na forma de um padrão ASCII por *byte*? E se for utilizada a notação binária?
- 8.** Uma alternativa para a notação hexadecimal, na representação de padrões de *bits*, é a *notação decimal pontuada* (*dotted decimal notation*), na qual cada *byte* do padrão é representado pelo seu equivalente em base dez. Estes códigos de um *byte* estão, por sua vez, separados por pontos. Por exemplo, 12.5 representa o padrão 000011000000101 (o *byte* 00001100 é representado por 12 e o 00000101, por 5), e o padrão 100010000010100000000111 é representado por 136.16.7. Represente os seguintes padrões de *bits* nesta notação.

a. 0000111100001111	b. 001100110000000010000000
c. 0000101010100000	
- 9.** Qual é a vantagem de representar imagens por meio de técnicas vetoriais em relação à representação por mapas de *bits*? E qual é a desvantagem?
- 10.** Suponha que uma gravação estereofônica de uma hora de música tenha sido codificada usando uma taxa de 44.100 amostras por segundo, como discutido no texto. Compare o tamanho da versão codificada com a capacidade de armazenamento de um CD.

*N. de T. *Onde você está?*

**N. de T. *“Como?”, Cheryl perguntou.*

1.5 O sistema binário

Antes de analisar as técnicas de armazenamento numérico utilizadas nas máquinas atuais, precisamos de maiores detalhes sobre o sistema de representação binária.

Adição binária

Para somar dois valores representados em notação binária, iniciamos da mesma maneira como se ensina nas escolas de ensino fundamental para a base dez, memorizando a tabuada de adição (Figura 1.19). Usando essa tabuada, somamos dois valores como se segue: primeiro, somamos os dois dígitos da coluna mais à direita; escrevendo o dígito menos significativo desta soma logo abaixo desta coluna, transportamos o dígito mais significativo desta soma parcial (se houver) para cima da coluna imediatamente à esquerda e prosseguimos somando esta coluna. Por exemplo, para efetuar a conta:

$$\begin{array}{r} 00111010 \\ + \underline{00011011} \end{array}$$

Começamos somando os dígitos da coluna mais à direita (0 e 1), obtendo 1, o qual escrevemos sob esta coluna. Agora somamos os dígitos 1 e 1 da próxima coluna e obtemos 10. Escrevemos o 0 deste 10 sob esta coluna e levamos o 1 para o topo da coluna seguinte. Neste ponto, nossa solução fica assim:

$$\begin{array}{r} 1 \\ 00111010 \\ + \underline{00011011} \\ \hline 01 \end{array}$$

Somamos o 1, o 0 e o 0 da próxima coluna, obtendo 1, e escrevemos o 1 sob esta coluna. Os dígitos 1 e 1 da coluna seguinte totalizam 10. Escrevemos o 0 sob esta coluna e levamos o 1 para a próxima coluna. Neste ponto, a solução se torna:

$$\begin{array}{r} 1 \\ 00111010 \\ + \underline{00011011} \\ \hline 0101 \end{array}$$

Os dígitos 1, 1 e 1 na coluna imediata totalizam 11. Escrevemos o 1 da direita sob esta coluna e levamos o outro 1 ao topo da coluna seguinte. Somamos este 1 com o 1 e o 0 que já estavam naquela coluna, obtendo 10. Novamente, escrevemos o 0 menos significativo e levamos o 1 para a coluna seguinte. Temos então:

$$\begin{array}{r} 1 \\ 00111010 \\ + \underline{00011011} \\ \hline 010101 \end{array}$$

Agora somamos os dígitos 1, 0 e 0 da penúltima coluna, obtendo 1, o qual escrevemos sob esta coluna, nada havendo para levar para a próxima. Finalmente, somamos os dígitos da última coluna, que resulta em 0, que escrevemos sob esta última coluna. Nossa solução fica, então:

$$\begin{array}{r} 00111010 \\ + \underline{00011011} \\ \hline 01010101 \end{array}$$

Figura 1.19 As tabuadas de adição binária.

$$\begin{array}{r} 0 & 1 & 0 & 1 \\ + 0 & + 0 & + 1 & + 1 \\ \hline 0 & 1 & 1 & 10 \end{array}$$

Frações de números binários

Para estender a notação binária de forma que acomode a representação de frações, também utilizamos uma **vírgula**^{*} (*radix point*), que funciona do mesmo modo que a vírgula da notação decimal. Em outras palavras, os dígitos à esquerda da vírgula representam a parte inteira do valor, sendo interpretados exatamente como no sistema binário previamente discutido. Os dígitos à direita da vírgula representam a parte fracionária do valor, sendo interpretados de modo semelhante aos outros *bits*, exceto em que às suas posições são associadas quantidades fracionárias: à primeira posição à direita da vírgula, é associada a quantidade $1/2$; à posição seguinte, a quantidade $1/4$; à próxima, $1/8$, e assim por diante. Note-se que isto é apenas uma extensão da regra previamente estabelecida: cada posição é associada a uma quantidade que é o dobro da quantidade associada à posição à sua direita. Com estes pesos associados às posições dos *bits*, é possível decodificar uma representação binária que contenha uma vírgula usando exatamente o mesmo procedimento empregado no caso em que não existe esta vírgula. Em particular, multiplicamos o valor correspondente a cada *bit* pela quantidade associada à posição por ele ocupada na representação do número. Para ilustrar, a representação binária 101.101 é decodificada em $5\frac{5}{8}$, como mostra a Figura 1.20.

Alternativas ao sistema binário

Os antigos computadores não aproveitavam o sistema de notação binária. De fato, a maneira como os valores numéricos deveriam ser representados nas máquinas computacionais era tema de ativos debates em fins da década de 1930 e durante a de 1940. Um candidato era o sistema biquinário, onde cada dígito na representação de base dez de um número era substituído por dois – um tinha o valor 0,1,2,3, ou 4 e o outro era 0 ou 5 – de tal forma que a soma equivalesse ao dígito original. Este foi o sistema usado no ENIAC. Outro candidato era a notação em base oito. No artigo “Binary Calculation”, que apareceu no *Journal of the Institute or Actuaries*, em 1936, E. W. Phillips escreveu: “A meta final é persuadir o mundo civilizado inteiro a abandonar a numeração decimal e em seu lugar usar a numeração octal; parar de contar em dezenas e sim em oitavas”.

Analógico versus digital

Um debate nos primórdios da computação era se os dispositivos computacionais deveriam ser baseados na tecnologia digital ou analógica. Em um sistema digital, um valor é representado por uma coleção de dispositivos; cada qual pode representar um número limitado de dígitos distintos (tais como 0 e 1). Em um sistema analógico, um valor é representado por um único dispositivo que pode conter qualquer valor em uma faixa contínua.

Vamos comparar as duas abordagens em termos de um recipiente d’água. Para simular um sistema digital, poderíamos convencionar que um recipiente vazio representasse o dígito 0 e um cheio, o dígito 1. Então armazenaríamos um valor numérico em uma carreira de recipientes usando a forma da notação binária. Em oposição a isso, poderíamos simular um sistema analógico enchendo parcialmente um único recipiente até o ponto em que o nível da água corresponesse ao valor numérico representado. A princípio, o sistema analógico pode parecer mais preciso, uma vez que não está sujeito a erros (tais como truncamento) inerentes ao sistema digital. Contudo, qualquer movimento no recipiente no sistema analógico pode causar erros de leitura do nível d’água, enquanto uma perturbação significativa teria de ocorrer no sistema digital para que a distinção entre o recipiente cheio e o vazio não pudesse ser feita. Assim, o sistema digital é menos sensível a erros do que o analógico. Essa robustez é a principal razão pela qual muitas aplicações originalmente baseadas na tecnologia analógica (tais como comunicação telefônica, gravação de áudio e televisão) estão se deslocando para a digital.

*N. de T. Este ponto (ponto decimal) é substituído por vírgula em muitos países, inclusive o Brasil.

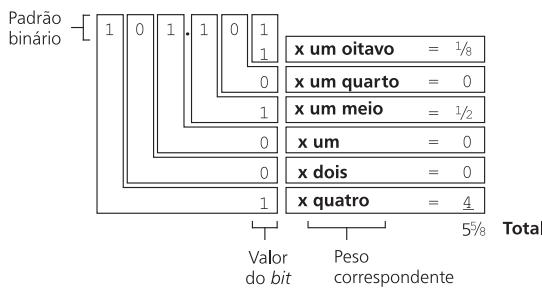


Figura 1.20 Decodificação da representação binária 101.101.

Quanto à adição, as técnicas aplicadas no sistema decimal também são aplicáveis no sistema binário. Assim, para somarmos duas representações binárias que contenham vírgula, basta alinharmos as vírgulas e aplicarmos o mesmo processo de adição estudado anteriormente. Por exemplo, 10,011 somado a 100,110 produz 111,001, conforme mostrado a seguir:

$$\begin{array}{r}
 10,011 \\
 + 100,110 \\
 \hline
 111,001
 \end{array}$$



QUESTÕES/EXERCÍCIOS

1. Converta as seguintes representações binárias em sua forma decimal equivalente:
 - a. 101010
 - b. 100001
 - c. 10111
 - d. 0110
 - e. 11111
2. Converta as seguintes representações decimais em sua forma binária equivalente:
 - a. 32
 - b. 64
 - c. 96
 - d. 15
 - e. 27
3. Converta as seguintes representações binárias nas suas representações equivalentes em base dez:
 - a. 11,01
 - b. 101,111
 - c. 10,1
 - d. 110,011
 - e. 0,101
4. Expressa os seguintes valores na notação binária:
 - a. $4\frac{1}{2}$
 - b. $2\frac{3}{4}$
 - c. $1\frac{1}{8}$
 - d. $\frac{5}{16}$
 - e. $\frac{55}{8}$
5. Efetue as seguintes adições em notação binária:

a. 11011	b. 1010,001	c. 11111
+ 1100	+ 1,101	+ 1
$ \begin{array}{r} \hline \end{array} $		
$ \begin{array}{r} \hline \end{array} $		
$ \begin{array}{r} \hline \end{array} $		

1.6 A representação de números inteiros

Há muito tempo, os matemáticos se interessam pelos sistemas de notação numérica, e muitas de suas idéias mostraram-se compatíveis com o projeto de circuitos digitais. Nesta seção, consideraremos dois desses sistemas: notação de complemento de dois e notação de excesso, usadas para representar valores inteiros nos equipamentos de computação. Esses sistemas são baseados no sistema binário apresentado na Seção 1.5, mas apresentam propriedades adicionais que fazem com que os sistemas fiquem mais compatíveis com o projeto de computadores. Junto com as vantagens, porém, surgem as desvantagens. Nossa meta é entender essas propriedades e seu efeito no uso dos computadores.

A notação de complemento de dois

O sistema mais conhecido para a representação interna de inteiros nos computadores modernos é a **notação de complemento de dois**. Este sistema emprega um número fixo de bits para

representar cada valor numérico. Nos equipamentos atuais, é comum o uso da notação de complemento de dois na qual cada valor é representado por um padrão de 32 bits. Esse sistema é grande e permite uma variedade de números a serem representados, mas é impróprio para fins de demonstração. Assim, para estudar as propriedades dos sistemas de complemento de dois, nos concentraremos nos sistemas menores.

A Figura 1.21 ilustra dois sistemas completos de complemento de dois — um baseado em padrões de bits de comprimento três e o outro, em padrões de bits de comprimento quatro. Tais sistemas são construídos iniciando-se com uma cadeia de zeros com o comprimento adotado e então contando em binário até que o padrão seja formado por um 0 seguido de 1s. Estes padrões representam os valores 0, 1, 2, 3,... Os padrões que representam valores negativos são obtidos começando com uma cadeia de 1s de comprimento apropriado e contando em binário, em ordem decrescente, até que o padrão obtido seja formado de um 1 seguido de 0s. Estes padrões representarão os valores -1, -2, -3,... (Se houver dificuldade para contar em binário, em ordem decrescente, pode-se começar pelo final da tabela, com o padrão formado por um 1 seguido de 0s, e contar em ordem crescente até que seja obtido o padrão formado só de 1s.)

Note-se que, em um sistema de complemento de dois, o bit mais à esquerda do padrão indica o sinal do valor representado. Assim, ele freqüentemente é chamado **bit de sinal**. Em um sistema de complemento de dois, os valores negativos são representados por padrões cujo bit de sinal é 1; valores não-negativos são representados por padrões cujo bit de sinal é 0.

Em um sistema de complemento de dois, existe uma relação conveniente entre padrões que representam números positivos e negativos de mesma magnitude. Estes padrões são idênticos quando lidos da direita para a esquerda, até a ocorrência do primeiro 1, inclusive. Desta posição em diante, os padrões são o complemento um do outro. (O complemento de um padrão é obtido mudando-se todos os 0s para 1s e todos os 1s para 0s; assim, 0110 e 1001 são complementos um do outro.) Por exemplo, no sistema de quatro bits da Figura 1.21, os padrões que representam 2 e -2 apresentam dígitos finais 10, porém o padrão que representa 2 começa com 00, enquanto o que representa -2 começa com 11. Isto conduz a um algoritmo de conversão de padrões de bits para representar números positivos e negativos de mesma magnitude.

O algoritmo consiste em copiar o padrão original da direita para a esquerda até o aparecimento do primeiro bit 1, substituindo-se, em seguida, os demais bits originais pelos seus complementos, ou seja, trocando-se todos os demais 1s por 0s e 0s por 1s (Figura 1.22). (Note que o maior valor negativo em um sistema de complemento de dois não tem um correspondente positivo dentro do sistema.)

A compreensão dessas propriedades básicas da notação de complemento de dois leva à construção de um algoritmo para a decodificação de representações numéricas nesta notação. Se o padrão a ser decodificado apresentar um bit de sinal igual a 0, simplesmente leremos o seu valor como se o padrão estivesse de-

a. Utilização de padrões de comprimento três

Padrão de bits	Valor representado
011	3
010	2
001	1
000	0
111	-1
110	-2
101	-3
100	-4

b. Utilização de padrões de comprimento quatro

Padrão de bits	Valor representado
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

Figura 1.21 Duas tabelas de conversão para a notação de excesso de oito.

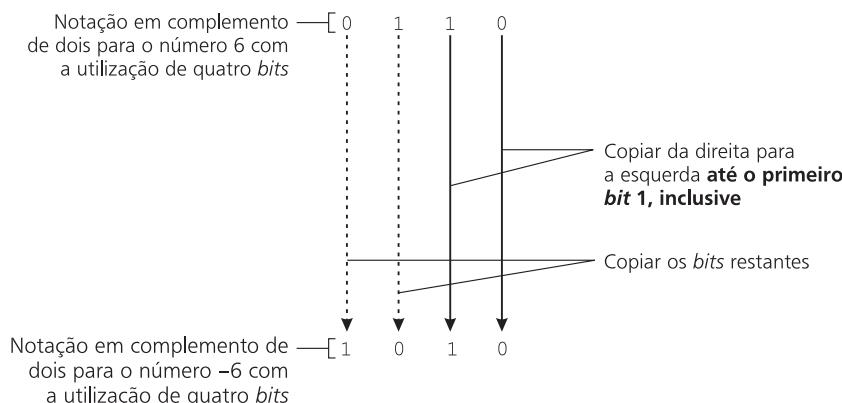


Figura 1.22 Codificação, em quatro *bits*, do valor -6 , em notação de complemento de dois.

meiro dígito 1, complementando os demais *bits* para finalmente decodificarmos o padrão resultante, como se fosse uma representação binária.

Por exemplo, para decodificar o padrão 1010, verificamos primeiro que o *bit* de sinal é 1, logo o valor representado é negativo. Em seguida, convertemos o padrão para 0110 e verificamos que isto representa 6, logo concluímos que o padrão original representa -6 .

Adição na notação de complemento de dois Para somar valores representados em complemento de dois, aplicamos o mesmo algoritmo empregado na adição binária, desde que todos os padrões de *bits*, inclusive o resultado, sejam do mesmo comprimento. Isto significa que quando se efetua uma soma em complemento de dois, deve ser descartado qualquer *bit* extra, gerado no último passo da operação de adição, à esquerda do resultado. Assim, a soma de 0101 com 0010 resulta 0111, e a de 0111 com 1011 resulta 0010 ($0111 + 1011 = 10010$, que é truncado para 0010).

Compreendido esse método, considerem-se as três contas de adição mostradas na Figura 1.23. Em cada caso, traduzimos o problema para a notação de complemento de dois (usando padrões de comprimento quatro), executamos o processo de adição descrito previamente e decodificamos o resultado, retornando à notação decimal usual.

Problema na base dez	Problema em complemento de dois	Resposta na base dez
$\begin{array}{r} 3 \\ + 2 \end{array}$	\rightarrow $\begin{array}{r} 0011 \\ + 0010 \\ \hline 0101 \end{array}$	\rightarrow 5
$\begin{array}{r} -3 \\ + -2 \end{array}$	\rightarrow $\begin{array}{r} 1101 \\ + 1110 \\ \hline 1011 \end{array}$	\rightarrow -5
$\begin{array}{r} 7 \\ + -5 \end{array}$	\rightarrow $\begin{array}{r} 0111 \\ + 1011 \\ \hline 0010 \end{array}$	\rightarrow 2

Figura 1.23 Problemas de adição, convertidos na notação de complemento de dois.

notado em código binário. Por exemplo, 0110 representa o número 6, porque 110 significa 6 em notação binária. Se o padrão a ser decodificado tiver 1 como *bit* de sinal, saberemos que o valor representado é negativo, e tudo o que restará a fazer será determinar a magnitude do número. Isto é feito copiando, da direita para a esquerda, o padrão original, até o aparecimento do prime

Observe-se que se fossem usadas as técnicas tradicionais ensinadas nas escolas de ensino fundamental, a terceira conta (subtração) exigiria um procedimento completamente diferente das outras duas. Por outro lado, ao expressar tais operações na notação de complemento de dois, podemos obter o resultado correto em todos os casos, aplicando o mesmo algoritmo. Esta é, então, a vantagem da notação de complemento de dois: a adição de qualquer combinação de números, positivos e negativos, pode ser efetuada usando-se um mesmo algoritmo e, portanto, o mesmo circuito.

Em contraste com o que ocorre com um estudante do ensino fundamental, que aprende primeiro a somar para depois subtrair, uma máquina que usa a notação de complemento de dois precisa apenas ser capaz de somar e complementar. Por exemplo, o problema de subtração $7 - 5$ equivale à adição $7 + (-5)$.

Por conseguinte, se for solicitado a uma máquina que subtraia 5 (armazenado como 0101) de 7 (armazenado como 0111), ela primeiro mudará 5 para -5 (representado como 1011), e então executará o algoritmo da adição para efetuar 0111 + 1011, obtendo o resultado 0010, que representa 2, conforme demonstrado a seguir:

$$\begin{array}{r} 7 & 0111 & 0111 \\ - 5 \rightarrow - 0101 \rightarrow + 1011 \\ \hline 0010 \rightarrow 2 \end{array}$$

Vemos então que, quando a notação de complemento de dois é usada para representar valores numéricos, um circuito somador, combinado com um circuito para alterar o sinal do número, é o suficiente para resolver os problemas de adição e subtração. (Tais circuitos são mostrados e explicados no Apêndice B.)

O problema do estouro Um problema que evitamos tratar nos exemplos anteriores é que, em qualquer sistema de complemento de dois, existe sempre um limite para o tamanho dos números a serem representados. Quando usamos complemento de dois, com padrões de quatro *bits*, ao valor 9 não está associado padrão algum e, por isso, não conseguimos obter uma resposta correta para a soma 5 + 4. De fato, o resultado apareceria como -7. Este erro é chamado **estouro**, o problema que ocorre quando o valor a ser representado cai fora da faixa permitida. Quando usado o complemento de dois, isto pode ocorrer ao se adicionar dois valores positivos ou dois negativos. Nos dois casos, a condição pode ser detectada verificando-se o bit de sinal do resultado. Em outras palavras, um estouro será indicado se a adição de dois valores positivos resultar em um padrão de valor negativo, ou se a adição de dois valores negativos resultar em um número positivo.

Evidentemente, uma vez que a maioria das máquinas manipula padrões de *bits* maiores do que os usados em nossos exemplos, valores maiores podem ser manipulados sem causar estouro. Hoje é comum usar 32 *bits* para armazenar valores em complemento de dois, permitindo valores positivos até 2.147.483.647 antes que ocorra estouro. Se forem necessários valores maiores, será possível usar padrões de *bits* mais longos ou talvez mudar as unidades de medida. Por exemplo, encontrar uma solução em milhas em vez de polegadas resulta no uso de números menores que podem ainda proporcionar a precisão necessária.

O fato é que os computadores podem errar. Assim, a pessoa que usa a máquina deve estar a par dos perigos envolvidos. Um problema é que os programadores e usuários tornam-se complacentes e ignoram o fato de que pequenos valores podem ser acumulados e produzir grandes números. Por exemplo, no passado, era comum usar padrões de 16 *bits* para representar valores na notação de complemento de dois, o que significa que os estouros não ocorreriam até que o valor $2^{15} = 32.768$ fosse alcançado. No dia 19 de setembro de 1989, o sistema computacional de um hospital falhou após anos de serviços confiáveis. Uma inspeção cuidadosa revelou que esta data era 32.768 dias após o 1º de janeiro de 1900. O que você acha que ocorreu?

A notação de excesso

Outro método para codificar números inteiros é a **notação de excesso**. Neste sistema, cada número é codificado como um padrão de *bits*, de comprimento convencional. Para estabelecer um sistema de excesso, primeiro escolhemos o comprimento do padrão a ser empregado; em seguida, escrevemos todos os diferentes padrões de *bits* com este comprimento, na ordem em que seriam gerados se estivéssemos contando em binário. Logo, observamos que o primeiro desses padrões, que apresenta um dígito 1 como seu *bit* mais significativo, figura aproximadamente no centro dessa lista. Escolhemos este padrão para representar o valor zero; os padrões que o seguem serão utilizados para representar 1, 2, 3...; os que o precedem serão adotados para a representação dos inteiros negativos -1, -2, -3.... O código resultante, para padrões de quatro *bits* de comprimento, é mostrado na Figura 1.24, na qual podemos observar que o valor 5 é representado pelo padrão 1101 e -5, por 0011. (Note

Padrão de bits	Valor representado
1111	7
1110	6
1101	5
1100	4
1011	3
1010	2
1001	1
1000	0
0111	-1
0110	-2
0101	-3
0100	-4
0011	-5
0010	-6
0001	-7
0000	-8

Figura 1.24 Uma tabela de conversão para a notação de excesso de oito.

Padrão de bits	Valor representado
111	3
110	2
101	1
100	0
011	-1
010	-2
001	-3
000	-4

Figura 1.25 Um sistema de notação de excesso, com padrões de três bits de comprimento.

que a diferença entre um sistema de excesso e um de complemento de dois é que os bits de sinal são trocados.)

O sistema apresentado na Figura 1.24 é conhecido como a notação de excesso de oito. Para entender o porquê desta denominação, interpretemos primeiramente cada um dos padrões codificados, utilizando para isso o sistema binário tradicional, e então comparemos os resultados desta observação com os valores representados no código de excesso. Em cada caso, poderemos constatar que a interpretação binária é maior do que a interpretação do código de excesso de oito. Por exemplo, o padrão 1100 normalmen-

te representa o valor 12, mas em nosso sistema de excesso, representa 4; 0000 representa 0, mas no sistema de excesso, representa -8. Do mesmo modo, um sistema de excesso baseado em padrões de comprimento cinco seria chamado de notação de excesso de 16, porque o padrão 10000, por exemplo, seria usado para representar o zero, em vez do seu valor habitual de 16. Do mesmo modo, podemos confirmar que o sistema de excesso que utiliza padrões de três bits seria conhecido como notação de excesso de quatro (Figura 1.25).

QUESTÕES/EXERCÍCIOS

- 
- Converta em decimais as seguintes representações em complemento de dois:
 - 00011
 - 01111
 - 11100
 - 11010
 - 00000
 - 10000
 - Converta as seguintes representações decimais na notação de complemento de dois em padrões de oito bits:
 - 6
 - 6
 - 17
 - 13
 - 1
 - 0
 - Suponha que os seguintes padrões de bits representem valores em notação de complemento de dois. Encontre a representação, em complemento de dois, do negativo de cada valor:
 - 00000001
 - 01010101
 - 11111100
 - 11111110
 - 00000000
 - 01111111
 - Suponha um computador que represente números na notação de complemento de dois. Quais os maiores e menores números representáveis utilizando padrões com os seguintes comprimentos?
 - quatro
 - seis
 - oito
 - Nas seguintes contas, cada padrão de bits representa um valor em notação de complemento de dois. Encontre a resposta para cada uma utilizando o processo de adição descrito no texto. A seguir, confira os resultados, resolvendo o mesmo problema em notação decimal.

a. 0101	b. 0011	c. 0101	d. 1110	e. 1010
+ 0010	+ 0001	+ 1010	+ 0011	+ 1110
<hr/>				

6. Faça as seguintes contas na notação de complemento de dois, mas desta vez preste atenção à ocorrência de estouro e indique quais resultados resultam incorretos devido a tal ocorrência.
- | | | | | | | | | | |
|----|----------------------|----|----------------------|----|----------------------|----|----------------------|----|----------------------|
| a. | 0100 | b. | 0101 | c. | 1010 | d. | 1010 | e. | 0111 |
| | $\underline{+ 0011}$ | | $\underline{+ 0110}$ | | $\underline{+ 1010}$ | | $\underline{+ 0111}$ | | $\underline{+ 0001}$ |
7. Converta as seguintes contas, da notação decimal para a notação de complemento de dois, usando padrões de bits de comprimento quatro. Converta-as então à forma de uma adição equivalente (como um computador faria) e finalmente efetue a adição. Confira suas respostas retornando-as à notação decimal.
- | | | | | | | | | | |
|----|---------------------|----|------------------|----|------------------|----|---------------------|----|------------------|
| a. | 6 | b. | 3 | c. | 4 | d. | 2 | e. | 1 |
| | $\underline{-(-1)}$ | | $\underline{-2}$ | | $\underline{-6}$ | | $\underline{-(-4)}$ | | $\underline{-5}$ |
8. Quando se somam números na notação de complemento de dois, pode ocorrer estouro quando um valor é positivo e o outro é negativo? Explique sua resposta.
9. Converta as seguintes representações de excesso de oito em seu equivalente decimal sem consultar a tabela do texto:
- | | | | | | |
|----|-----------------------|----|-----------------------|----|-----------------------|
| a. | 1110 | b. | 0111 | c. | 1000 |
| | $\underline{d. 0010}$ | | $\underline{e. 0000}$ | | $\underline{f. 1001}$ |
10. Converta as seguintes representações decimais na notação de excesso de oito sem consultar a tabela do texto:
- | | | | | | | | | | | | |
|----|---|----|----|----|---|----|---|----|---|----|----|
| a. | 5 | b. | -5 | c. | 3 | d. | 0 | e. | 7 | f. | -8 |
|----|---|----|----|----|---|----|---|----|---|----|----|
11. O valor 9 pode ser representado na notação de excesso de oito? E quanto a representar o número 6 em notação de excesso de quatro? Explique sua resposta.

1.7 A representação de frações

Em contraste com o que ocorre no caso dos números inteiros, a representação de valores com parte fracionária requer não apenas o armazenamento do padrão de 0s e 1s que representa sua notação binária, mas também deve conter uma informação sobre a posição da vírgula^{*}. Uma notação muito usada, que permite atingir tal objetivo, inspira-se na notação científica e é conhecida como **notação de vírgula flutuante**.

A notação de vírgula flutuante

Vamos explicar a notação de vírgula flutuante por meio de um exemplo que emprega somente um byte de memória. Embora os computadores normalmente usem padrões mais longos, este exemplo é representativo dos casos reais e serve para demonstrar os principais conceitos sem as dificuldades trazidas pelos padrões longos de bits.

Primeiramente, escolhemos o bit mais significativo do byte para ser o bit de sinal do número. Novamente, um 0 neste bit significa que o valor representado é não-negativo, enquanto um 1 indica que é negativo. Em seguida, dividimos os sete bits restantes do byte em dois grupos, ou campos, o **campo de expoente** e o **campo de mantissa**. Vamos atribuir aos três bits que seguem o bit de sinal a função de campo de expoente, e aos quatro bits restantes, a de campo de mantissa. Assim, o byte fica dividido, conforme ilustra a Figura 1.26.

^{*}N. de T. Vírgula binária, que separa as partes inteira e não-inteira dos números, similar à vírgula decimal dos números reais em base dez. Em inglês, usa-se o ponto.

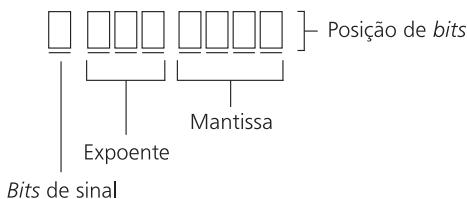


Figura 1.26 Componentes da notação de ponto flutuante.

Podemos explicar o significado desses campos com a ajuda do exemplo seguinte. Suponha que um *byte* conteña o padrão de *bits* 01101011. Interpretando este padrão no formato que acabamos de definir, constatamos que o *bit* de sinal é 0, o expoente é 110, e a mantissa, 1011. Para decodificar o *byte*, extraímos primeiro a mantissa e colocamos a vírgula binária à sua esquerda, obtendo:

,1011

Em seguida, extraímos o conteúdo do campo do expoente (110) e o interpretamos como um inteiro codificado em três *bits* pelo método de representação de excesso (ver Figura 1.25). Assim, o padrão contido no campo de expoente do nosso exemplo representa o número positivo 2. Isto indica que devemos deslocar a vírgula binária dois *bits* à direita. (Um expoente negativo indica um deslocamento para a esquerda.) Como resultado, obtemos:

10,11

que representa $2\frac{3}{4}$. Em seguida, notamos que o *bit* de sinal do nosso exemplo é 0; assim, o valor representado é não-negativo. Concluímos que $2\frac{3}{4}$ é o número representado pelo *byte* 01101011 nesta notação.

Como outro exemplo, consideremos o *byte* 10111100. Extraímos a mantissa, obtendo:

,1100

e deslocamos a vírgula binária para a esquerda, uma vez que o campo de expoente (011) representa o valor -1. Então, temos:

,01100

que representa $\frac{3}{8}$. Uma vez que o *bit* de sinal, no padrão original, é 1, o valor codificado é negativo. Concluímos que o padrão 10111100 representa o valor $-\frac{3}{8}$.

Para armazenar um valor usando a notação de vírgula flutuante, revertemos o processo anterior. Por exemplo, para codificar $1\frac{1}{8}$, primeiramente expressamos este valor na notação binária, obtendo 1,001. Em seguida, copiamos o padrão de *bits* no campo da mantissa, da esquerda para a direita, iniciando com o primeiro *bit* não-nulo da representação binária. Neste momento, o *byte* assume o aspecto:

— — — 1 0 0 1

Agora, devemos preencher o campo de expoente. Para tanto, imaginamos o conteúdo do campo da mantissa com uma vírgula à sua esquerda e determinamos o número de posições e o sentido de deslocamento da vírgula, necessários à obtenção do número binário original. No nosso exemplo, verificamos que a vírgula no padrão ,1001 deve ser deslocada de um *bit* à direita para obtermos 1,001. Como, agora, o expoente deve ser um 1 positivo, colocamos 101 (que representa o valor 1 positivo na notação de excesso de quatro) no campo de expoente. Finalmente, preenchemos o *bit* de sinal com 0, porque o valor a ser codificado é não-negativo. Assim, o *byte* final fica:

0 1 0 1 1 0 0 1

Há um ponto sutil que talvez tenha passado despercebido ao se preencher o campo de mantissa. A regra é copiar o padrão de *bits* que aparece na representação binária da esquerda para a direita, começando com o 1 mais à esquerda. Para esclarecer, considere o processo de armazenar o valor $\frac{3}{8}$, que é ,011 na notação binária. Neste caso, a mantissa será

— — — 1 1 0 0

Ela não será

— — — 0 1 1 0

Isto porque preenchemos o campo de mantissa *começando com o 1 mais à esquerda* que aparece na representação binária. Essa regra elimina a possibilidade de múltiplas representações para o mesmo valor. Também implica que a representação de todos os valores diferentes de zero terá mantissa cujo primeiro bit seja 1. Essa representação está na **forma normalizada**. Note que o valor zero deve ser um caso especial; sua representação em vírgula flutuante é um padrão de bits 0s.

Erros de truncamento

Consideremos o problema incômodo de representar o número $2^{5/8}$ no sistema de vírgula flutuante de um byte. Primeiramente, escrevemos $2^{5/8}$ em binário, obtendo 10,101. Entretanto, ao copiarmos este código no campo da mantissa, não haverá espaço suficiente, e o último 1 (o qual representa a última parcela $1/8$) se perderá (Figura 1.27). Se ignorarmos este problema por ora e continuarmos a preencher o campo de expoente e do bit de sinal, teremos o padrão de bits 01101010, que representa o valor $2^{1/2}$, e não $2^{5/8}$. O fenômeno assim observado é denominado **erro de truncamento** ou **erro de arredondamento**, ou seja, que parte do valor que está sendo armazenado se perdeu porque o campo de mantissa não é grande o suficiente.

A consequência de tais erros pode ser reduzida usando-se uma mantissa maior. De modo semelhante ao que ocorre com a representação de inteiros, é comum o uso de 32 bits na notação de vírgula flutuante em vez dos oito aqui utilizados. Esta abordagem também permite que o campo de expoente seja estendido, tornando-se mais longo. Entretanto, mesmo com estes formatos mais longos, há ocasiões em que se exige uma precisão maior ainda.

Outra fonte de erros de truncamento, muito comum na notação decimal, é o problema das dízimas periódicas, tais como a que ocorre, por exemplo, quando se tenta exprimir $1/3$ em decimal. Alguns valores não podem ser expressos com precisão absoluta, independentemente do número de dígitos utilizados para representá-los.

A diferença entre a notação decimal usual e a binária é a existência de um número muito maior de valores, na notação binária, cujas representações são dízimas. Por exemplo, o valor $1/10$ é uma dízima quando denotado em binário. Imagine os problemas com que se defronta um usuário incauto que se utilize da notação de vírgula flutuante para representar e manipular valores financeiros. Em particular, se a quantia de um real for utilizada como unidade de medida, nem sequer o valor de uma moeda de dez centavos poderá ser representado com precisão. Uma solução para este caso consiste em converter tais valores em centavos, de forma que todos os valores representados sejam inteiros, podendo assim ser representados com precisão mediante a codificação em complemento de dois.

Os erros de truncamento e os problemas deles decorrentes são uma preocupação constante de pessoas que trabalham com análise numérica.

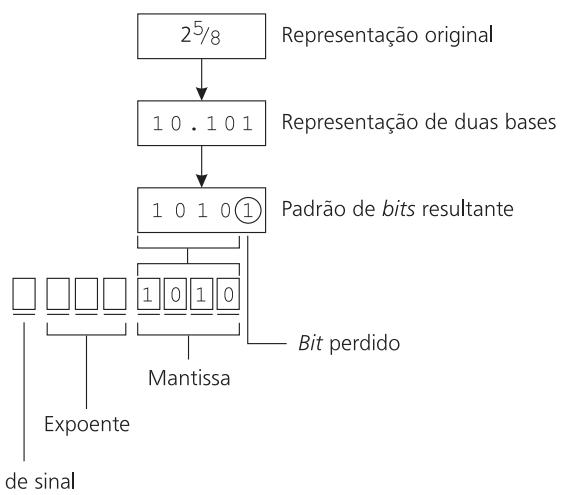


Figura 1.27 Codificação do valor de $2^{5/8}$.

ca. Este ramo da matemática enfrenta tais problemas ao processar dados reais, geralmente volumosos e que exigem uma precisão considerável.

Finalizamos esta seção com um exemplo capaz de entusiasmar qualquer analista numérico. Suponha que se deseje somar os três seguintes valores, utilizando a notação de vírgula flutuante de um byte definida anteriormente:

$$2^{1/2} + \frac{1}{8} + \frac{1}{8}$$

Se somarmos os valores na ordem em que se encontram, primeiro somaremos $2^{1/2}$ e $\frac{1}{8}$ e obteremos $2^{5/8}$, o qual em binário é 10,101. Infelizmente, um erro de truncamento ocorre quando armazenamos este valor, então o resultado deste primeiro passo é igual a $2^{1/2}$ (que é igual a uma das parcelas). O próximo passo é somar tal resultado com o último $\frac{1}{8}$. Aqui ocorre mais uma vez o mesmo erro de truncamento, o que resulta em um valor final incorreto, igual a $2^{1/2}$.

Somemos agora os valores na ordem oposta. Primeiro, somemos $\frac{1}{8}$ com $\frac{1}{8}$ e obteremos $\frac{1}{4}$, que em binário seria ,01; assim, o resultado deste primeiro passo é armazenado em um byte como 00111000, que está correto. Somemos, a seguir, este $\frac{1}{4}$ com o próximo valor da lista, $2^{1/2}$, e obteremos $2^{3/4}$, o qual também pode ser armazenado com precisão em um byte como 01101011, gerando-se desta maneira a resposta correta.

Em resumo, quando se somam números em notação de vírgula flutuante, a ordem em que são somados pode ser importante. O problema é que se um número muito grande for somado com um muito pequeno, o pequeno poderá ser truncado. Assim, a regra geral para somar muitos valores é primeiramente somar os pequenos na esperança de que o valor acumulado seja significativo quando somado aos valores grandes. Este foi o fenômeno constatado no exemplo precedente.

Os projetistas dos pacotes comerciais atuais fazem um bom trabalho ao evitar que um usuário incauto sofra esse tipo de problema. Em uma planilha eletrônica típica, as respostas corretas são obtidas a menos que os valores somados difiram em tamanho por 10^{16} ou mais. Assim, se for necessário somar um ao valor

10.000.000.000.000.000

você pode obter a resposta

10.000.000.000.000.000

em vez de

10.000.000.000.000.001

Tais problemas são significativos em aplicações (como sistemas de navegação) nos quais pequenos erros podem ser combinados em computações adicionais, produzindo graves consequências, mas para um usuário comum de computador pessoal esse grau de exatidão é suficiente.



QUESTÕES/EXERCÍCIOS

1. Decodifique os seguintes padrões de bits utilizando a notação de vírgula flutuante estudada no texto:
 - a. 01001010
 - b. 01101101
 - c. 00111001
 - d. 11011100
 - e. 10101011

2. Codifique os seguintes valores utilizando a notação de vírgula flutuante estudada no texto. Indique as ocorrências de erros de truncamento.
 - a. $2^{3/4}$
 - b. $5^{1/2}$
 - c. $3^{3/4}$
 - d. $-3^{1/2}$
 - e. $-4^{3/8}$

3. Na notação de vírgula flutuante discutida no texto, qual dos padrões 01001001 e 00111101 representa o maior valor? Descreva um procedimento simples para determinar o maior dentre os valores associados a dois padrões fornecidos.
4. Na notação de vírgula flutuante descrita no texto, qual o maior valor que pode ser representado? E qual o menor valor positivo que pode ser representado?

1.8 Compressão de dados

Com a finalidade de armazenar e transferir dados, freqüentemente é útil (e às vezes obrigatório) reduzir o seu tamanho. A técnica para isso se chama **compressão de dados**. Iniciamos esta seção considerando alguns métodos genéricos de compressão de dados; em seguida, apresentamos algumas abordagens projetadas especificamente para comprimir imagens.

Técnicas genéricas de compressão

Numerosas técnicas de compressão de dados têm sido desenvolvidas, cada qual com seu cenário de melhor e pior caso. O método chamado **codificação de tamanho de seqüência** (*run-length encoding*) produz ótimo resultado quando os dados que estão sendo comprimidos consistem em longas seqüências com o mesmo valor. Com efeito, a codificação de tamanho de seqüência é o processo de substituir tais seqüências por um código indicativo do valor repetido e do número de vezes que ele ocorre na seqüência. Por exemplo, é necessário menos espaço para indicar que um padrão de *bits* consiste em 253 uns seguidos por 118 zeros seguidos por 87 uns, do que para listar os 458 *bits*.

Em alguns casos, a informação envolvida consiste em blocos de dados, em que cada um difere ligeiramente do anterior. Um exemplo seria os quadros consecutivos de um filme. Nesses casos, as técnicas que usam a **codificação relativa** são úteis. A abordagem consiste em registrar as diferenças entre blocos consecutivos em vez dos blocos inteiros, isto é, cada bloco é codificado em termos de sua relação com o bloco precedente.

Outra abordagem para reduzir o tamanho dos dados é a **codificação dependente da freqüência**, que é um sistema no qual o tamanho do padrão de *bits* usado para representar um item de dados é inversamente proporcional à freqüência de uso do item. Esses códigos são exemplos dos **códigos de comprimento variável**, o que significa que os itens são representados por padrões de tamanho diferente, em oposição a códigos como o Unicode, no qual todos os símbolos são representados usando 16 *bits*. Mais precisamente, na língua inglesa, as letras *e*, *t*, *a* e *i* são usadas mais freqüentemente do que as letras *z*, *q* e *x*. Assim, quando se constrói um código para texto em inglês, o espaço pode ser economizado usando pequenos padrões de *bits* para representar as primeiras letras e longos padrões para as últimas. O resultado é um código no qual um texto em inglês teria representação menor do que a que seria obtida com códigos de comprimento uniforme como o ASCII e o Unicode. David Huffman foi o descobridor de

Compressão do som

Como aprendemos na Seção 1.4, um segundo de música estéreo codificada com uma taxa de 44.100 amostras por segundo necessita mais de um milhão de *bits* para ser armazenado. Essas exigências de espaço são aceitáveis para gravações musicais distribuídas em CDs, mas desafiam as capacidades da tecnologia quando combinadas com vídeo para produzir gravação de cinema. Assim, o *Motion Picture Expert Group* do ISO tem desenvolvido técnicas de compressão que reduzem significativamente a necessidade de espaço de armazenamento para áudio. Uma delas é conhecida como MP3 (MPEG-1 Audio Layer-3) que pode obter taxas de compressão de 12 para 1. Usando MP3, as gravações de música podem ser reduzidas para um tamanho que pode ser economicamente transmitido através da Internet – uma possibilidade que ameaça revolucionar a indústria de gravações musicais.

um algoritmo amplamente utilizado no desenvolvimento de códigos dependentes de freqüência, e é comum referir-se aos códigos desenvolvidos dessa maneira como **Códigos de Huffman**. De fato, a maioria dos códigos dependentes de freqüência atualmente em uso é código de Huffman.

Embora tenhamos introduzido a codificação de tamanho de seqüência, a codificação relativa e a codificação dependente de freqüência como técnicas genéricas de compressão, cada uma tende a ter seu próprio domínio de aplicação. No entanto, os sistemas baseados na **codificação de Lempel-Ziv** (assim chamada por causa de seus criadores Abraham Lempel e Jacob Ziv) são verdadeiramente genéricos. Com efeito, os usuários da Internet provavelmente já viram e talvez usaram programas que utilizam as técnicas Lempel-Ziv para comprimir arquivos, conhecidos como arquivos *zip*.

Os sistemas de codificação Lempel-Ziv são exemplos da **codificação adaptável de dicionário**. Aqui, o termo *dicionário* refere-se a uma coleção de módulos a partir da qual a mensagem a ser comprimida é construída. Se quisermos comprimir um texto em inglês, os módulos poderão ser os caracteres do alfabeto. Se quisermos comprimir dados já codificados como uma cadeia de 0s e 1s, os módulos poderão ser os dígitos 0 e 1. Em um sistema de codificação adaptável de dicionário, o dicionário pode mudar durante o processo de codificação. Por exemplo, no caso de texto em inglês, após codificar parte da mensagem, podemos decidir adicionar *ing* e *the* ao dicionário. Assim, qualquer nova ocorrência de *ing* e *the* pode ser codificada com uma única referência ao dicionário, em vez de três. Os sistemas de codificação Lempel-Ziv usam maneiras inteligentes e eficientes para adaptar o dicionário durante o processo de codificação (compressão).

Como exemplo, vamos considerar como poderíamos comprimir uma mensagem usando um sistema Lempel-Ziv particular, conhecido como LZ77. Iniciamos tomando a parte inicial da mensagem. Então representamos o restante da mesma como seqüências de triplas (que consistem em dois números e um símbolo da mensagem), e cada uma descreve como a próxima parte da mensagem deve ser construída em função das partes precedentes. (Assim, o dicionário a partir do qual a mensagem foi construída consiste na própria mensagem).

Por exemplo, considere a mensagem comprimida

$xyxxzy(5, 4, x)$

que consiste no segmento inicial *xyxxzy* seguido pela tripla (5, 4, x). A cadeia *xyxxzy* é a parte da mensagem que já está na forma descompactada. Para descompactar o resto da mensagem, devemos decodificar a tripla (5, 4, x) como especificado na Figura 1.28. O primeiro número da tripla diz-nos até que ponto devemos contar retroativamente na cadeia descompactada. No caso, contamos retroativamente cinco símbolos, o que nos leva ao segundo x da esquerda para a direita da cadeia descompactada. Agora acrescentamos ao final desta os símbolos encontrados nesta posição. O segundo número da tripla nos diz quantos símbolos consecutivos devem ser acrescentados. No caso, este número é 4; assim, acrescentamos os símbolos *xyz* ao fim da cadeia descompactada e obtemos

$xyxxzyxyz$

Finalmente, a última parte da tripla deve ser colocada no final da cadeia estendida. Isso produz

$xyxxzyxyzx$

que é a mensagem descompactada.

Agora suponha que a versão compactada da mensagem fosse

$xyxxzy(5, 4, x)(0, 0, w)(8, 6, y)$

Começaríamos descompactando a primeira tripla como antes para obter

$xyxxzyxyzx(0, 0, w)(8, 6, y)$

Então decodificaremos a segunda tripla para obter

$xyxxzyxyzxw(8, 6, y)$

Note que a tripla $(0, 0, w)$ foi usada porque o símbolo w ainda não havia aparecido na mensagem. Finalmente, decodificando a terceira tripla, teríamos a mensagem descompactada

$\text{xyxxzyxxyzwxzyxxyz}$

Para comprimir uma mensagem usando o LZ77, podemos inicialmente tomar um segmento inicial da mensagem e então procurar o maior segmento no padrão tomado que coincide com o restante da mensagem a ser comprimida. Este será o padrão referenciado na primeira tripla. As outras tripas são formadas por um processo similar.

Finalmente, você deve ter notado que os nossos exemplos não refletem muita compressão, uma vez que as tripas envolvidas representam segmentos pequenos. Quando aplicado a padrões longos de bits, contudo, é razoável que longos segmentos sejam representados por tripas — o que resulta em uma compressão de dados significativa.

Compressão de imagens

Na Seção 1.4, vimos que os mapas de bits produzidos pelos digitalizadores atuais tendem a representar as imagens em um formato de três bytes por pixel, o que leva a mapas de bits grandes e intratáveis. Muitos esquemas de compressão têm sido desenvolvidos para reduzir esta necessidade de armazenamento. Um sistema conhecido como GIF (abreviatura de Graphic Interchange Format e pronunciado “Guif” por uns e “Jif” por outros) foi desenvolvido pela CompuServe. Ele aborda o problema reduzindo o número de cores que podem ser atribuídas a um pixel a apenas 256; assim, o valor de cada pixel é representado em um único byte em vez de três. Cada um dos 256 valores possíveis é associado a uma combinação de vermelho, verde e azul por meio de uma tabela conhecida como palheta. Ao mudar uma palheta associada a uma imagem, podemos mudar as cores que aparecem nesta.

A uma das cores na palheta GIF normalmente é atribuído o valor “transparente”, o que significa que o fundo é mostrado através de qualquer região que tenha essa “cor”. Essa opção, combinada com a relativa simplicidade do sistema GIF, faz dele uma escolha lógica em jogos de ação computadorizados nos quais múltiplas imagens se movem na tela.

Outro sistema de compressão de imagens coloridas é o JPEG (pronuncia-se j-peg, em português, ou *jei-pig* em inglês). É um padrão desenvolvido pelo Joint Photographic Expert Group (daí o nome do padrão) um grupo da ISO. O JPEG demonstrou ser um padrão efetivo para representar fotografias coloridas. Com efeito, ele é adotado pelos fabricantes das câmeras digitais atuais e promete um grande impacto no contexto das imagens digitais nos anos vindouros.

O padrão JPEG na verdade engloba vários métodos para representar imagens, cada qual com seus objetivos. Por exemplo, nas situações em que se exige o máximo em precisão, o JPEG provê o modo “menos perda”, cujo nome implica que nenhuma informação é perdida no processo de codificação da imagem. Nesse modo, o espaço é economizado ao armazenar a diferença entre pixels consecutivos em vez das intensidades dos pixels — a teoria é que, na maioria dos casos, a quantidade pela qual pixels adjacentes diferem pode ser representada por menos padrões de bits do que os usados para representar os valores dos pixels. (Este é um exemplo de codificação relativa.) Essas diferenças são então codificadas usando códigos de tamanho variável para reduzir o espaço de armazenamento.

Infelizmente, o uso do modo “menos perda” do JPEG não leva a mapas de bits com tamanhos manejáveis pela tecnologia atual, e por isso raramente é usado. Em seu lugar, a maioria das aplicações

- x y x x y z y
 a. Conte retroativamente 5 símbolos.
- x y [x x y z] y
 b. Identifique o segmento de quatro símbolos a ser acrescentado ao final da cadeia.
- x y [x x y z] y [x x y z]
 c. Copie o segmento de quatro símbolos no final da mensagem.
- x y x x y z y x x y z [x]
 d. Acrescente o símbolo identificado na tripla ao fim da mensagem.

Figura 1.28 Descompactação de $\text{xyxxzy} (5, 4, \text{x})$.

usa o padrão JPEG básico, que reduz o tamanho de uma imagem codificada distinguindo o brilho da cor de cada *pixel*.

O propósito de distinguir entre luminosidade e cor é que o olho humano é mais sensível a mudança de brilho do que de cor. Considere, por exemplo, dois fundos azuis idênticos, exceto em que um deles contenha um pequeno ponto brilhante, enquanto o outro, um pequeno ponto verde com o mesmo brilho do fundo azul. Seu olho encontraria mais prontamente o ponto brilhante do que o verde. O padrão básico do JPEG tira vantagem desse fenômeno, codificando cada componente de brilho, mas dividindo a imagem em blocos de quatro *pixels* e registrando apenas a cor média de cada bloco. Assim, a representação final preserva mudanças repentinas no brilho, mas tende a obscurecer mudanças repentinas de cor. O benefício é que cada bloco de quatro *pixels* é representado por apenas seis valores (quatro valores de brilho e dois de cor) em vez de 12 valores que seriam necessários em um sistema de três *bytes* por *pixel*.

Economiza-se mais espaço ao registrar dados que indicam como os vários componentes de brilho e cor mudam, em vez de seus valores. Aqui, como no modo “menos perda” do JPEG, a motivação é que à medida que a imagem vai sendo percorrida, as diferenças entre valores de *pixels* próximos podem ser registradas usando menos *bits* do que seria necessário se os próprios valores fossem registrados. (Na realidade, essas diferenças são codificadas aplicando-se uma técnica matemática conhecida como transformação discreta de cossenos, cujos detalhes não nos interessam aqui.) O padrão final de *bits* é compreendido a seguir, aplicando um sistema de codificação de comprimento variável.

Em resumo, o padrão básico do JPEG pode codificar imagens coloridas de alta qualidade usando padrões de *bits* que estão na faixa de um vigésimo do tamanho necessário ao formato de três *bytes* por *pixel*, usado pela maioria dos digitalizadores. Isso explica por que ele está crescendo em popularidade. Outras técnicas, contudo, têm vantagens em certas aplicações. GIF, por exemplo, faz um serviço melhor ao representar imagens que consistem em blocos de cores uniformes com bordas estreitas (como os desenhos coloridos) do que o JPEG.

Para encerrar, devemos notar que a pesquisa em compressão de dados representa um campo largo e ativo. Discutimos apenas duas de muitas técnicas para comprimir imagens. Além disso, existem muitas estratégias para comprimir áudio e vídeo. Por exemplo, técnicas similares às usadas no padrão básico do JPEG têm sido adotadas pelo Motion Picture Expert Group (MPEG) da ISO com a finalidade de estabelecer padrões para codificar (comprimir) filmes de cinema. A idéia subjacente é iniciar a seqüência de quadros com uma imagem similar ao padrão básico do JPEG e então representar o resto da seqüência usando técnicas de codificação relativa.



QUESTÕES/EXERCÍCIOS

1. Abaixo está uma mensagem que foi comprimida usando o LZ77. Qual é a cadeia descompactada?
101101011 (7,5,0) (12,10,1) (18,13,0)
2. Embora não tenhamos nos concentrado no algoritmo para comprimir dados baseado no LZ77, tente comprimir a mensagem
bbabbbbaababaababaabaaaa
3. Nesta seção, afirmamos que o GIF é melhor do que o JPEG para representar desenhos coloridos. Explique por quê.
4. No máximo, quantos *bytes* seriam necessários para representar uma imagem de 1024 por 1024 *pixels* usando GIF? E quantos se for usado o padrão básico do JPEG?
5. Qual característica do olho humano foi explorada pelo padrão básico do JPEG?
7. Identifique um fenômeno complicador que é comum quando se codifica informação numérica, imagens e sons com padrões de *bits*.

1.9 Erros de comunicação

Quando há troca de informação de um lado para outro entre as partes de um computador, ou transmissão da Terra para a Lua e vice-versa, ou quando essa informação é simplesmente armazenada, existe uma possibilidade de que, ao final desse processo, os padrões de bits recuperados não sejam exatamente idênticos aos originais. Partículas de sujeira ou de gordura sobre a superfície magnética do meio de armazenamento, ou um circuito com falhas de funcionamento, podem causar erros de gravação ou de leitura. Além disso, no caso de algumas tecnologias, a radiação de fundo pode alterar padrões armazenados na memória principal de um computador.

Para solucionar tais problemas, foram desenvolvidas diversas técnicas de codificação para permitir a detecção e até mesmo a correção de tais erros. Atualmente, por serem extensivamente empregadas na estrutura interna dos componentes de um sistema computacional, essas técnicas acabam passando despercebidas aos usuários. Todavia, a sua presença é muito importante e representa uma contribuição significativa para a pesquisa científica. É razoável, portanto, a investigação de algumas dessas técnicas, nas quais se apóia a confiabilidade dos equipamentos modernos.

Bits de paridade

Um método simples para a detecção de erros se baseia no princípio de que, se cada padrão correto de bits tiver um número ímpar de 1s, então, se for encontrado um padrão com um número par de 1s, isto indicará a presença de um erro.

Para utilizar este princípio, precisamos de um sistema em que cada padrão correto contenha sempre um número ímpar de 1s. Isto é fácil de obter, acrescentando-se primeiramente um bit, denominado **bit de paridade**, para cada padrão de um sistema de codificação já disponível (em geral, à esquerda do bit mais significativo). (Assim, o código ASCII de oito bits será convertido em um de nove bits, ou um padrão de dezesseis bits, na notação de complemento de dois, se tornará um padrão de dezessete.) Nos dois casos, atribuímos os valores 1 ou 0 para este novo bit de tal forma que o padrão resultante tenha sempre um número ímpar de 1s. Como a Figura 1.29 mostra, o código ASCII para a letra A é 101000001 (bit de paridade 1), e o código ASCII para a letra F é 001000110 (bit de paridade 0). Embora o padrão de oito bits para a letra A tenha um número par de 1s e aquele para a letra F tenha um número ímpar de 1s, os dois padrões de nove bits apresentam um número ímpar de 1s. Uma vez que o nosso sistema de codificação tenha sido modificado de acordo com o esquema descrito anteriormente, a detecção de um padrão com um número par de 1s será sintoma da presença de um erro no padrão de bits em questão.

Este sistema de paridade que acabamos de descrever é conhecido como **paridade ímpar**, pois cada padrão correto contém um número ímpar de bits. Uma alternativa é a utilização da **paridade par**, na qual cada padrão é construído de forma que contenha um número par de 1s. Nesse caso, um erro é sinalizado pela detecção de um padrão de bits que contenha um número ímpar de 1s.

Atualmente, não é raro encontrarmos bits de paridade utilizados na memória principal de um computador. Embora tais computadores sejam tidos por seus usuários como máquinas cujas posições de memória têm oito bits cada uma, na realidade, essas memórias podem ter células de nove bits, um

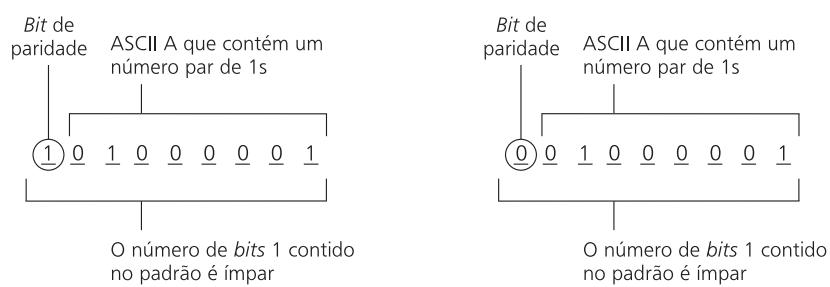


Figura 1.29 Os códigos ASCII das letras A e F, adaptados para paridade ímpar.

dos quais opera como *bit* de paridade. Todas as vezes que um padrão de oito *bits* é fornecido ao circuito de memória para armazenamento, este lhe acrescenta um *bit* de paridade adequado, armazenando o resultado como padrão de nove *bits*. Mais tarde, quando esse padrão for recuperado, os circuitos de leitura irão conferir a paridade do padrão de nove *bits*. Se não for indicado um erro, o *bit* de paridade será removido, e o padrão resultante voltará a ser de oito *bits*. Caso contrário, a memória devolverá, juntamente com os oito *bits*, um aviso de que o padrão recuperado talvez não seja idêntico ao originalmente apresentado à memória para ser armazenado.

Longos padrões de *bits* freqüentemente são acompanhados por um conjunto de *bits* de paridade, que formam um **byte de verificação**. Cada *bit* desse byte é um *bit* de paridade associado a um conjunto de *bits* espalhados ao longo do padrão. Por exemplo, um *bit* de paridade pode estar associado ao conjunto de todos os oitavos *bits* contados a partir do primeiro *bit* do padrão, enquanto um outro talvez esteja associado a todos os oitavos *bits*, contados a partir do segundo *bit*. Desta forma, provavelmente será mais fácil encontrar vários erros, concentrados em uma área do padrão original, dado que estarão no escopo de vários *bits* de paridade. Algumas variações deste conceito de byte de verificação para a detecção de erros dão origem a esquemas de detecção de erros, conhecidos como verificação de soma (*checksum*) e como códigos de redundância cíclica (*cyclic redundancy codes — CRC*).

Códigos de correção de erros

Embora seja possível detectar a presença de um erro com um único *bit* de paridade, ele não fornece informações suficientes para corrigir o erro. Muitas pessoas se surpreendem ao saber que é possível projetar **códigos de correção** de forma que eventuais erros sejam não só descobertos como também corrigidos. Afinal de contas, a intuição diz que não podemos corrigir os erros de uma mensagem incorretamente recebida a menos que já tenhamos conhecimento da informação nela contida. Contudo, a Figura 1.30 apresenta um código simples que exibe essa propriedade corretiva.

Para entender como este código funciona, definimos primeiro a **distância de Hamming** (assim denominada em homenagem a R. W. Hamming, pioneiro na busca de códigos de correção de erros, motivado pela falta de confiabilidade das primeiras máquinas dos anos 1940) entre dois padrões de *bits* como o número de *bits* diferentes existente entre eles. Por exemplo, a distância de Hamming entre A e B no código da Figura 1.30 é quatro e entre B e C, três. A principal característica do código é que dois padrões quaisquer estão separados por uma distância de Hamming de pelo menos três. Se um único *bit* for alterado em consequência de um mau funcionamento de um dispositivo, o erro será detectado, uma vez que o resultado obtido não será um padrão legal. (Devemos alterar pelo menos três *bits* de qualquer padrão legal antes que seja aceito como outro padrão legal.)

Se tiver ocorrido somente um erro no padrão da Figura 1.30, será possível descobrir qual era o seu padrão original. De fato, o padrão modificado estará a uma distância de Hamming de um único *bit* da sua forma original, mas, no mínimo, a dois *bits* de quaisquer outros padrões legais. Para decodificar uma mensagem, simplesmente comparamos cada padrão recebido com os padrões da tabela de códigos, até encontrar um que esteja a uma distância de um *bit* do padrão recebido. Este será o símbolo correto, obtido por meio da decodificação. Por exemplo, suponha o padrão de *bits* 010100. Se o compararmos com os padrões da tabela de códigos, obteremos a tabela da Figura 1.31. Assim, concluiremos que o símbolo transmitido deve ter sido um D, por apresentar o padrão mais próximo do original.

Percebe-se que, utilizando esta técnica com os códigos da Figura 1.30, é possível descobrir até dois erros por padrão e corrigir um deles. Se projetássemos o código de forma que cada padrão tivesse, pelo menos,

Símbolo	Código
A	000000
B	001111
C	010011
D	011100
E	100110
F	101001
G	110101
H	111010

Figura 1.30 Um código de correção de erros.

uma distância de Hamming de cinco bits dos demais padrões, seria possível descobrir até quatro erros por padrão e corrigir dois deles. Logicamente, projetar códigos eficientes associados a distâncias de Hamming grandes não é uma tarefa fácil. De fato, constitui uma parte do ramo da matemática denominado teoria de codificação algébrica, que é uma subárea da álgebra linear e da teoria de matrizes.

As técnicas de correção de erros são usadas extensivamente para aumentar a confiabilidade dos equipamentos de computação. Por exemplo, elas freqüentemente são usadas nos dispositivos de discos magnéticos de alta capacidade para reduzir a possibilidade de um defeito na superfície magnética corromper os dados. Além disso, a principal distinção entre o formato original dos CDs usados como discos de áudio e o formato mais recente, usado como armazenamento para dados de computador, é o grau de correção de erros envolvido. O formato CD-DA incorpora características de correção de erros que reduzem a taxa para apenas um erro em dois CDs. Isto é adequado para gravações de áudio, mas uma companhia que usa CDs para fornecer software a seus clientes consideraria que erros em 50% dos discos são intoleráveis. Assim, características adicionais de correção de erros são empregadas em CDs usados para armazenamento de dados, reduzindo a possibilidade de erros para 1 em 20.000 discos.

Símbolo	Distância entre o padrão recebido e o símbolo em análise
A	2
B	4
C	3
D	① menor distância
E	3
F	5
G	2
H	4

Figura 1.31 Decodificação do padrão 010100 utilizando a tabela da Figura 1.30.



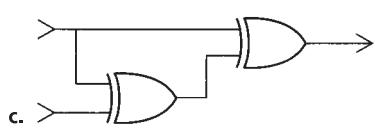
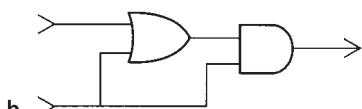
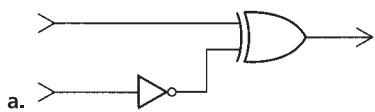
QUESTÕES/EXERCÍCIOS

- Os bytes seguintes foram originalmente codificados com paridade ímpar. Em quais deles é possível afirmar que existe algum erro?
 a. 10101101 b. 10000001 c. 00000000 d. 11100000
 e. 11111111
- Nos bytes da questão 1, pode haver erros imperceptíveis? Explique sua resposta.
- Suas respostas para as questões 1 e 2 seriam diferentes se a paridade utilizada fosse par e não ímpar?
- Codifique as sentenças abaixo em ASCII, usando paridade ímpar, por meio da inserção de um bit de paridade como bit mais significativo de cada código dos caracteres:
 a. *Where are you?*
 b. *“How”? Cheryl asked.*
 c. $2+3 = 5$
- Usando o código de correção de erros apresentado na Figura 1.30, decodifique as seguintes mensagens:
 a. 001111 100100 001100
 b. 010001 000000 001011
 c. 011010 110110 100000 011100
- Construa códigos para os caracteres A, B, C e D usando padrões de bits de comprimento cinco, de modo que a distância de Hamming entre dois padrões quaisquer seja no mínimo três.

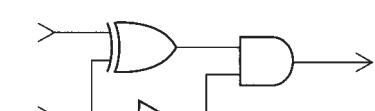
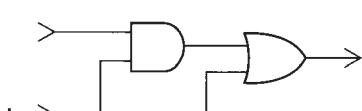
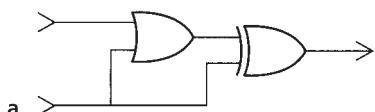
Problemas de revisão de capítulo

(Os problemas marcados com asterisco se referem às seções opcionais.)

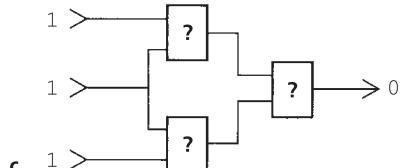
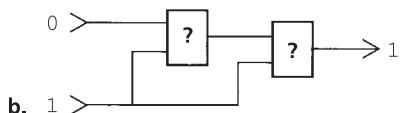
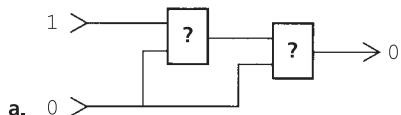
- 1.** Determine a saída dos seguintes circuitos pressupondo que a entrada superior é 1 e a inferior é 0.



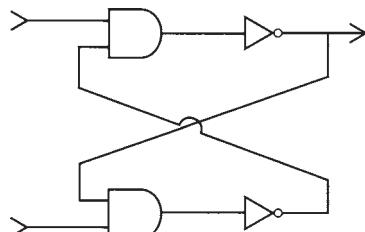
- 2.** Nos seguintes circuitos, identifique a combinação de entradas que produzirá uma saída igual a 1.



- 3.** Em cada circuito abaixo, o retângulo representa o mesmo tipo de porta lógica. Baseado na informação dada de entrada e saída, identifique se a porta lógica envolvida é AND, OR ou XOR.



- 4.** Suponha que as duas entradas do circuito abaixo sejam 1. Descreva o que ocorrerá se a entrada superior for alterada temporariamente para 0. Ou se a entrada inferior for alterada temporariamente para 0. Redesenhe o circuito usando portas NAND.



5. A seguinte tabela representa, em notação hexadecimal, os endereços e conteúdos de algumas posições da memória principal de um computador. A partir desta configuração inicial de memória, execute a seqüência de instruções indicada abaixo, e registre os resultados finais de cada posição de memória:

Endereço	Conteúdo
00	AB
01	53
02	D6
03	02

Passo 1. Transfira o conteúdo da posição de memória de endereço 03 para a posição de endereço 00.

Passo 2. Coloque o valor 01 na posição de memória de endereço 02.

Passo 3. Transfira o valor armazenado no endereço 01 para a posição de endereço 03.

6. Quantas posições existem na memória principal de um computador se o endereço de qualquer dessas posições pode ser representado por três dígitos hexadecimais?
7. Que padrões de bits são representados pelas seguintes notações hexadecimais?
- BC
 - 67
 - 9A
 - 10
 - 3F
8. Qual é o valor do bit mais significativo dos padrões de bits representados pelas seguintes notações hexadecimais?
- FF
 - 7F
 - 8F
 - 1F
9. Expressse os seguintes padrões de bits na notação hexadecimal:
- 101010101010
 - 110010110111
 - 000011101011
10. Suponha que uma tela de monitor apresente 24 linhas e cada uma contenha 80 caracteres. Se o conteúdo da tela fosse armazenado na memória, por meio da codificação dos caracteres em ASCII (um caractere por byte), quantos bytes de memória seriam necessários para apresentar a imagem completa?
11. Suponha que uma figura seja representada na tela de um monitor através de uma matriz retangular de 1024 colunas e 768 linhas de pixels. Se forem necessários oito bits para

codificar a cor e a intensidade de cada pixel, quantos bytes de memória serão precisos para guardar a figura toda?

12. a. Identifique duas vantagens que a memória principal apresenta em relação ao armazenamento em disco.
- b. Identifique duas vantagens que o armazenamento em disco apresenta em relação à memória principal.
13. Suponha que se deseje utilizar um computador pessoal para escrever um ensaio com 40 páginas, digitadas em espaço duplo. O computador tem uma unidade de disco com disquetes de 3 1/4 polegadas (1 polegada = 2,54 cm) com uma capacidade de 1,44 MB por disquete. Seu ensaio completo caberá em um desses discos? Neste caso, quantos ensaios podem ser armazenados em um disco? Se não, quantos discos serão necessários?
14. Suponha que apenas 100 MB dos 10 GB do disco rígido de seu computador pessoal estejam livres e que você esteja a ponto de substituí-lo por um outro de 30 GB. Você deseja guardar temporariamente todas as informações armazenadas no antigo disco rígido em disquetes de 3 1/4" enquanto efetua a conversão. É prático proceder dessa maneira? Explique a sua resposta.
15. Se cada setor de um disco comporta 512 bytes, quantos setores serão necessários para conter uma página de texto, digitada em espaço duplo? Cada símbolo ocupa um byte.
16. Um disquete comum de 3 1/4" tem uma capacidade de 1,44MB. Relacione essa capacidade com o tamanho de um romance de 400 páginas, sendo que cada página contém 3500 caracteres.
17. Se um disquete, com 16 setores por trilha e 512 bytes por setor, gira à razão de 300 revoluções por minuto, a que taxa aproximada, expressa em bytes por segundo, os dados passam pelo cabeçote de leitura/gravação?
18. Se um microcomputador, utilizando o disquete do Problema 17, executar dez instruções por microsegundo (milionésimo de segundo), quantas instruções poderá executar durante o período transcorrido entre as passagens de bytes consecutivos pelo cabeçote de leitura / gravação?

- 19.** Se um disquete girar a 300 rotações por minuto e o computador executar dez instruções por microsegundo (milionésimo de segundo), quantas instruções ele poderá executar durante o tempo de latência do disco?
- 20.** Compare o tempo de latência de um disquete comum, como o do Problema 19, a um dispositivo de disco rígido que gire a 60 rotações por segundo.
- 21.** Qual é o tempo médio de acesso de um disco rígido de 60 revoluções por segundo, com um tempo de busca igual a 10 milissegundos?
- 22.** Suponha que um digitador, trabalhando continuamente dia e noite digite a uma velocidade constante de 60 palavras por minuto. Quanto tempo ele levará para preencher um CD cuja capacidade seja de 640 MB? Pressuponha que uma palavra seja formada por cinco caracteres e que cada um ocupe um byte.
- 23.** Decodifique a mensagem abaixo, escrita em ASCII:
- | | | |
|----------|----------|----------|
| 01010111 | 01101000 | 01100001 |
| 01110100 | 00100000 | 01100100 |
| 01101111 | 01100101 | 01110011 |
| 00100000 | 01101001 | 01110100 |
| 00100000 | 01110011 | 01100001 |
| 01111001 | 00111111 | |
- 24.** A seguinte mensagem, inicialmente codificada em ASCII com caracteres de um byte, foi expressa abaixo, em notação hexadecimal. Qual é a mensagem?
68657861646563696D616C
- 25.** Codifique as seguintes sentenças, em código ASCII com caracteres de um byte:
- $100/5 = 20$
 - Ser ou não ser?
 - O custo total é de R\$ 7,25.
- 26.** Expresse as respostas do Problema 25 em notação hexadecimal.
- 27.** Liste as representações binárias dos inteiros de 6 a 16.
- 28.**
 - Escreva o número 13, codificando em ASCII os dígitos 1 e 3.
 - Escreva o número 13, usando codificação binária.
- 29.** Que números exibem, na sua representação binária, somente um dos seus bits igual a 1? Liste a representação binária dos seis menores números que apresentam esta propriedade.
- *30.** Expresse os textos abaixo em código ASCII, utilizando um byte por símbolo. Use o bit mais significativo de cada byte como bit de paridade (empregue a paridade ímpar).
- $100/5 = 20$
 - Ser ou não ser?
 - O custo total é de R\$ 7,25.
- *31.** A seguinte mensagem foi transmitida originalmente com paridade ímpar em cada uma das suas cadeias menores de bits. Em quais dessas pequenas cadeias é possível garantir a presença de erros?
- | | | | | | |
|-------|-------|-------|-------|-------|-------|
| 11011 | 01011 | 10110 | 00000 | 11111 | 10101 |
| 10001 | 00100 | 01110 | | | |
- *32.** Seja um código de 24 bits gerado representando-se cada símbolo como três cópias sucessivas de sua representação ASCII (por exemplo, o símbolo A seria representado pela cadeia de bits 010000010100000101000001). Analise este novo código quanto a suas propriedades referentes à correção de erros.
- *33.** Utilizando o código de correção de erros descrito na Figura 1.30, decodifique as seguintes palavras:
- 111010 110110
 - 101000 100110 001100
 - 011101 000110 000000 010100
 - 010010 001000 001110 101111
000000 110111 100110
 - 010011 000000 101001 100110
- *34.** Converta as seguintes representações binárias em sua forma decimal equivalente:
- 111
 - 0001
 - 11101
 - 10001
 - 10111
 - 000000
 - 100
 - 1000
 - 10000
 - 11001
 - 11010
 - 11011
- *35.** Converta as seguintes representações decimais em sua forma binária equivalente:
- 7
 - 12
 - 16
 - 15
 - 33
- *36.** Converta na forma decimal as seguintes representações, em excesso de 16:
- 10000
 - 10011
 - 01101
 - 01111
 - 10111

- *37.** Converta na notação de excesso de quatro os seguintes números decimais:
- 0
 - 3
 - 3
 - 1
 - 1
- *38.** Converta em decimais as seguintes representações em complemento de dois:
- 10000
 - 10011
 - 01101
 - 01111
 - 10111
- *39.** Converta as seguintes representações decimais na notação de complemento de dois com padrões de sete bits:
- 12
 - 12
 - 1
 - 0
 - 8
- *40.** Execute as seguintes adições, pressupondo que as cadeias de bits representam valores na notação de complemento de dois. Identifique os casos em que a resposta esteja incorreta devido à ocorrência de estouro:
- | | | |
|----------|----------|----------|
| a. 00101 | b. 01111 | c. 11111 |
| + 01000 | + 00001 | + 00001 |
- | | | |
|----------|----------|----------|
| d. 10111 | e. 00111 | f. 00111 |
| + 11010 | + 00111 | + 01100 |
- | | | |
|----------|----------|----------|
| g. 11111 | h. 01010 | i. 01000 |
| + 11111 | + 10101 | + 01000 |
- | | | |
|----------|--|--|
| j. 01010 | | |
| + 00011 | | |
- *41.** Resolva as contas abaixo, convertendo inicialmente os valores na notação de complemento de dois (usando padrões de 5 bits), substituindo qualquer operação de subtração por uma equivalente de adição e executando tal adição. Confira o resultado reconvertendo-o na notação decimal. (Mantenha-se atento à ocorrência de estouro).
- | | | |
|------|------|-------|
| a. 7 | b. 7 | c. 12 |
| + 1 | - 1 | - 4 |
- | | | |
|------|-------|------|
| d. 8 | e. 12 | f. 4 |
| - 7 | + 4 | + 11 |
- *42.** Converta em decimais as seguintes representações binárias:
- 11,001
 - 100,1101
 - ,0101
 - 1,0
 - 10,01
- *43.** Expresse em notação binária os seguintes valores numéricos:
- $\frac{5}{4}$
 - $\frac{1}{16}$
 - $\frac{7}{8}$
 - $\frac{11}{4}$
 - $\frac{65}{8}$
- *44.** Decodifique os seguintes padrões de bits utilizando a notação de vírgula flutuante discutida na Seção 1.7:
- 01011100
 - 11001000
 - 00101010
 - 10111001
- *45.** Codifique os seguintes valores utilizando a notação de vírgula flutuante discutida na Seção 1.7. Aponte os casos em que ocorrerem erros de truncamento:
- $\frac{1}{2}$
 - $7\frac{1}{2}$
 - $-3\frac{3}{4}$
 - $\frac{3}{32}$
 - $\frac{31}{32}$
- *46.** Qual a melhor aproximação da raiz quadrada de 2 que possa ser expressa na notação de vírgula flutuante descrita na Seção 1.7? Que resultado se obterá caso tal valor aproximado seja elevado ao quadrado por um computador que utilize a notação de vírgula flutuante?
- *47.** Qual a melhor aproximação, para o valor *um décimo*, que pode ser codificada utilizando a notação de vírgula flutuante descrita na Seção 1.7?
- *48.** Explique como podem ocorrer erros quando medidas que usam o sistema métrico são gravadas na notação de vírgula flutuante. Por exemplo, o que acontece se 110 cm for gravado na unidade metro?
- *49.** Usando o formato de vírgula flutuante com oito bits descrito na Seção 1.7, qual seria o resultado da seguinte soma $\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + 2\frac{1}{2}$, quando resolvida da esquerda para a direita? E quando resolvida da direita para a esquerda?
- *50.** Que resposta seria obtida para as seguintes contas de adição executadas por uma máquina usando o formato de vírgula flutuante com 8 bits descrito na Seção 1.7?
- $1\frac{1}{2} + \frac{3}{16} =$
 - $3\frac{1}{4} + 1\frac{1}{8} =$
 - $2\frac{1}{4} + 1\frac{1}{8} =$
- *51.** Nas seguintes contas de adição, interprete os padrões de bits utilizando o formato de vírgula flutuante com 8 bits discutido na Seção 1.7, some os valores representados e codifique a resposta no mesmo formato de vírgula flutuante. Identifique os casos em que ocorrerem erros de truncamento.
- | | |
|-------------|-------------|
| a. 01011100 | b. 01101010 |
| + 01101000 | + 00111000 |

$$\begin{array}{ll} \text{c. } & 01111000 \\ & + 00011000 \\ & \hline \end{array} \quad \begin{array}{ll} \text{d. } & 01011000 \\ & + 01011000 \\ & \hline \end{array}$$

- *52.** Um destes padrões de bits (01011 ou 11011) representa um valor codificado na notação de excesso de 16 e o outro, o mesmo valor, codificado na notação de complemento de dois.
- O que é possível determinar acerca deste valor comum?
 - Usando o mesmo número de bits para as duas representações, qual é a relação existente entre o padrão que representa um valor na notação de complemento de dois e aquele que o representa em notação de excesso?
- *53.** Os três padrões de bits 01101000, 10000010 e 00000010 são representações do mesmo valor na notação de complemento de dois, na de excesso e no formato de vírgula flutuante com oito bits, discutidos na Seção 1.7, mas não necessariamente nesta ordem. Qual é o valor comum representado? Qual notação corresponde a cada padrão?
- *54.** Nos casos a seguir, as diferentes cadeias de bits representam o mesmo valor, mas em diferentes sistemas, já discutidos, de codificação numérica. Identifique cada número representado e os sistemas de codificação associados a cada representação.
- 11111010 0011 1011
 - 11111101 01111101 11101100
 - 1010 0010 01101000
- *55.** Quais dos seguintes padrões de bits não são representações válidas, na notação de excesso de 16?
- 01001
 - 101
 - 010101
 - 00000
 - 1000
 - 000000
 - 1111

- *56.** Quais dos seguintes valores não podem ser representados com precisão no formato de vírgula flutuante introduzido na Seção 1.7?
- $6\frac{1}{2}$
 - 9
 - $\frac{13}{16}$
 - $17\frac{1}{32}$
 - $\frac{15}{16}$
- *57.** Duplicando de quatro para oito bits o comprimento das cadeias de bits utilizadas para representar inteiros em binário, como se modifica o valor do maior inteiro representável nessa notação? E na notação de complemento de dois?
- *58.** Qual será a representação hexadecimal do endereço mais alto de uma memória de 4 MB, se cada posição de memória comportar um só byte?
- *59.** Usando portas lógicas, projete um circuito com quatro entradas e uma só saída, de modo que esta seja 1 ou 0, dependendo de o padrão formado pelas suas quatro entradas ter paridade ímpar ou par, respectivamente.
- *60.** A seguinte mensagem foi comprimida usando o LZ77. Faça a sua descompactação:
abr (3, 1, c) (2, 1, d) (7, 3, a)
Por que a mensagem comprimida é maior do que a original?
- *61.** A seguinte mensagem foi comprimida usando o LZ77. Faça a sua descompactação:
0100101 (4, 3, 0) (8, 7, 1) (17, 9, 1) (8, 6, 1)
- *62.** Aqui está parte de uma mensagem que foi comprimida usando o LZ77. Baseando-se na informação dada, qual é o tamanho da mensagem original?
xyz (_, 3, y) (_, 6, z)
- *63.** Escreva um conjunto de diretrizes que expliquem como comprimir uma mensagem usando o LZ77.
- *64.** Escreva um conjunto de diretrizes que expliquem como descompactar uma cadeia usando o LZ77.

Questões sociais

As seguintes questões procuram auxiliar o leitor no entendimento de alguns assuntos éticos, sociais e legais no campo da computação. O objetivo não é meramente o de fornecer respostas para tais questões; o leitor também deve justificá-las e verificar se as justificativas preservam sua consistência de uma questão para outra.

1. Suponha a ocorrência de um erro de truncamento em uma situação crítica, causando enormes danos e perda de vidas. Quem é o responsável, se houver: o projetista do *hardware*? o projetista do *software*? o programador que escreveu esta parte do programa? a pessoa que decidiu aplicar o *software* naquela situação em particular? E se o *software* já tivesse sido corrigido pela companhia que originalmente o desenvolveu, mas o *software* de atualização (*upgrade*) do programa não tivesse sido adquirido, tendo sido empregada a versão antiga nessa aplicação crítica? E se o *software* tiver sido pirateado?
2. É aceitável que um indivíduo ignore a possibilidade de erros de truncamento durante o desenvolvimento de suas aplicações?
3. Foi ético desenvolver *software* na década de 1970 usando apenas dois dígitos para representar o ano (com usar 76 para representar 1976) ignorando-se o fato de que o *software* se tornaria incorreto na virada do século? Atualmente, é ético usar três dígitos para representar o ano? (como 982 para 1982 e 015 para 2015). E se forem usados quatro dígitos?
4. Muitos defendem o ponto de vista de que a codificação dilui ou distorce a informação, uma vez que ela, em sua essência, força a quantificação da informação. Argumentam que codificar opiniões acerca de determinados assuntos por meio de questionários, em uma graduação de 1 a 5, é inherentemente um método que acaba por truncar a informação. Até que ponto a informação é quantificável? Podem ser quantificados os argumentos pró e contra acerca do local de instalação de uma fábrica de reciclagem de lixo? O debate sobre a energia nuclear e o “lixo” que produz é quantificável? É perigoso tomar decisões baseadas em médias e outras análises estatísticas? É ético que as agências de notícias informem o resultado das apurações das respostas aos questionários sem mencionar o teor exato das perguntas? É possível quantificar o valor de uma vida humana? É aceitável que uma companhia deixe de investir na melhoria de um produto, mesmo sabendo que tal investimento reduziria as possibilidades da ocorrência de morte relacionadas com o uso desse produto?
5. Com o desenvolvimento de câmeras digitais, a possibilidade de alterar ou produzir fotografias é colocada à disposição do público em geral. Que mudanças isso traz à sociedade? Quais as questões éticas e legais que poderão surgir?
6. Deve haver uma distinção nos direitos de coletar e disseminar dados em função de sua forma? Em outras palavras, o direito de coletar e disseminar fotografia, áudio e vídeo é o mesmo direito de coletar e disseminar texto?
7. Intencionalmente ou não, em geral, a reportagem de um jornalista reflete a sua opinião. Freqüentemente a mudança de algumas poucas palavras pode dar uma conotação positiva ou negativa a um relato. (Compare “A maioria dos observadores não acreditava que...” com “Uma parte significativa dos observadores concordava que...”). Existe diferença entre alterar uma história (deixando de lado alguns pontos ou selecionado cuidadosamente as palavras) e alterar uma fotografia?
8. Suponha que o uso de um sistema de compressão de dados resulte na perda de itens de informação sutis, mas significativos. Que responsabilidades podem ser questionadas? Como elas devem ser resolvidas?

Leituras adicionais

- Gibbs, S. J., and D. C. Tsichritzis. *Multimedia Programming*. Reading, MA: Addison-Wesley, 1995.
- Hamacher, V. C., Z. G. Vranesic, and S. G. Zaky. *Computer Organization*, 4th ed. New York: McGraw-Hill, 1996.
- Kay, D. C. and J. R. Levine. *Graphics File Formats* 2nd ed. New York, McGraw-Hill, 1995.
- Knuth, D. E. *The Art of Computer Programming I*, vol. 2, 3rd ed. Reading MA: Addison-Wesley Longman, 1998.
- Miano, J. *Compressed Image File Formats*. New York: ACM Press, 1999.
- Patterson, D. A., and J. L. Hennessy. *Computer Organization and Design*. 2nd ed. San Francisco: Morgan Kaufmann, 1997.
- Sayood, K. *Introduction to Data Compression* 2nd ed. San Francisco: Morgan Kaufmann, 2000.

C A P Í T U L O

2

Manipulação de dados

No Capítulo 1, estudamos conceitos relativos ao armazenamento de dados e à memória de um computador. Além de armazenar dados, o computador deve ser capaz de manipulá-los de acordo com um algoritmo. Tal manipulação exige que a máquina seja dotada de mecanismos capazes de executar operações com dados e coordenar a seqüência das mesmas. Essas tarefas são executadas por um mecanismo denominado Unidade Central de Processamento (UCP). Esta unidade e os tópicos a ela relacionados são o objeto de estudo deste capítulo.

- 2.1 Arquitetura de computadores**
- 2.2 Linguagem de máquina**
 - O repertório de instruções
 - Uma linguagem de máquina ilustrativa
- 2.3 Execução de programas**
 - Um exemplo de execução de programa
 - Programas versus dados
- * 2.4 Instruções aritméticas e lógicas**
 - Operações lógicas
 - Operações de rotação e deslocamento
 - Operações aritméticas
- * 2.5 Comunicação com outros dispositivos**
 - Comunicação via controladores
 - Taxas de comunicação de dados
- * 2.6 Outras arquiteturas**
 - Canalização (*pipelining*)
 - Máquinas com multiprocessamento

*Os asteriscos indicam sugestões de seções consideradas opcionais.

2.1 Arquitetura de computadores

Os circuitos de um computador que executam operações com dados (como adição e subtração) não são diretamente conectados às células de armazenamento da memória principal do computador, ficando isolados em uma parte conhecida como **unidade central de processamento** ou UCP (*central processing unit*) ou, simplesmente, processador. Esta unidade consiste em duas partes: a **unidade aritmética e lógica**, que contém os circuitos que manipulam os dados, e a **unidade de controle**, que contém os circuitos que coordenam as atividades da máquina.

Para armazenamento temporário de informação, a UCP contém células ou **registradores**, que são semelhantes às posições da memória principal. Tais registradores podem ser classificados como de **propósito geral** ou **propósito específico**. Conheceremos alguns dos registradores de propósito específico na Seção 2.3. Por ora, estudaremos a função dos registradores de propósito geral.

Os registradores de propósito geral funcionam como posições temporárias de armazenamento para dados que estão sendo manipulados pela UCP. Esses registradores guardam os dados de entrada da unidade aritmética e lógica e proporcionam um local de armazenamento para os seus resultados. Para executar uma operação sobre dados guardados na memória principal, é responsabilidade da unidade de controle transferir os dados da memória para os registradores de propósito geral, informar a unidade aritmética e lógica, cujos registradores armazenam os dados, ativar o circuito apropriado contido nesta unidade e informá-la sobre qual registrador receberá o resultado.

Para poder transferir padrões de bits entre o processador de um computador e a memória principal, tais elementos são interconectados por um conjunto de fios chamado **via** (*bus*) (Figura 2.1). Através desta via, o processador pode extrair ou ler dados da memória principal e, para tanto, ele fornece o endereço da posição de memória correspondente juntamente com um comando de leitura. Do mesmo modo, o processador pode colocar ou escrever dados na memória, fornecendo, para isso, o endereço da posição de destino e os dados a serem armazenados, acompanhados de um sinal de gravação.

Com este mecanismo em mente, verificamos que executar uma operação de adição com dados armazenados na memória principal é mais do que efetuar uma mera execução desta operação. O processo envolve esforços combinados entre a unidade de controle, que coordena a transferência de informação entre os registradores e a memória principal, e a unidade aritmética e lógica, que efetua a operação de adição propriamente dita, quando for instruída para isso pela unidade de controle.

O processo completo da adição de dois números armazenados na memória poderia ser dividido em uma sequência de passos, conforme ilustra a Figura 2.2. Em resumo, os dados devem ser transferidos da memória principal para o processador, onde os valores são somados, e o resultado deve ser armazenado em uma célula de memória.

Os primeiros computadores não eram conhecidos por sua flexibilidade — os passos que cada dispositivo executava estavam embutidos na unidade de controle, fazendo parte da máquina. Isto é semelhante a uma caixa de música que sempre toca a mesma melodia quando o que se deseja realmente é uma flexibilidade como a exibida

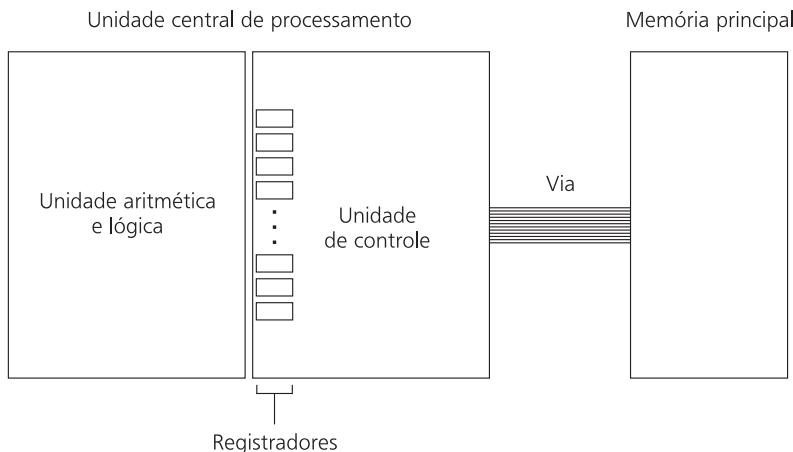


Figura 2.1 UCP e memória principal ligadas por uma via.

- Passo 1.** Obter da memória um dos valores a somar e guardá-lo em um registrador.
- Passo 2.** Obter da memória a outra parcela e armazená-la em outro registrador.
- Passo 3.** Acionar o circuito da adição, tendo os registradores utilizados nos passos 1 e 2 como entradas, e escolher outro registrador para armazenar o resultado.
- Passo 4.** Armazenar o resultado na memória.
- Passo 5.** Finalizar.

Figura 2.2 Um roteiro para somar valores armazenados na memória.

por equipamentos de troca automática de CDs. Para aumentar a flexibilidade, alguns computadores eletrônicos da época foram projetados de maneira a se poder alterar a fiação da unidade de controle. Isso foi conseguido com um painel de controle semelhante às antigas centrais telefônicas, nas quais as extremidades dos cabos eram inseridas nos orifícios correspondentes à conexão desejada.

Uma inovação (creditada, talvez incorretamente, a John von Neumann) surgiu a partir da constatação de que o programa, da mesma forma que os dados, poderia ser codificado e armazenado na memória principal. Se a unidade de controle fosse projetada com capacidade de extrair o programa da memória, decodificar as instruções e executá-las, um programa de computador poderia ser modificado, simplesmente alterando o conteúdo da memória, em vez de refazer a fiação da unidade de controle.

O **conceito de programa armazenado** tornou-se a abordagem padrão usada atualmente — na realidade, tão padronizada que parece óbvia. O que originalmente a tornava difícil é que todo mundo pensava nos programas e dados como entidades diferentes. Os dados eram armazenados na memória; os programas eram parte da unidade de controle. O resultado era um exemplo primordial de não conseguir perceber a floresta por estar observando as árvores. É fácil cair e ficar preso nessas valetas, e o desenvolvimento da ciência da computação pode muito bem permanecer em muitas delas sem que nos apercebamos disto. Sem dúvida, uma parte empolgante da ciência é que novas percepções estão constantemente abrindo as portas para novas teorias e novas aplicações.



QUESTÕES/EXERCÍCIOS

1. Qual a seqüência de eventos que o computador necessita para copiar o conteúdo de uma para outra posição de memória?
2. Quais são as ordens que o processador deve dirigir aos circuitos da memória principal para que seja guardado um número em uma certa posição de memória?
3. O armazenamento em massa, a memória principal e os registradores de propósito geral são três sistemas de armazenamento. Qual é a diferença quanto à maneira como são usados?

Quem inventou o quê?

Atribuir a um único indivíduo o crédito por uma invenção é sempre um empreendimento duvidoso. Thomas Edison é considerado o inventor da lâmpada incandescente, mas outros pesquisadores estavam desenvolvendo lâmpadas similares e, até certo ponto, Edison teve sorte de ser o primeiro a obter a patente. Os irmãos Wright são considerados os inventores do avião, mas se beneficiaram das pesquisas de outros e, em algum grau, foram influenciados por Leonardo da Vinci, que brincava com a idéia de máquinas voadoras já no século XVII. Entretanto, os projetos de Leonardo eram aparentemente baseados em idéias mais antigas. Obviamente, nesses casos, o inventor designado ainda tem reivindicações legítimas quanto ao crédito concedido. Em outros casos, a História parece ter conferido o crédito inapropriadamente — um exemplo é o conceito de programa armazenado. Sem dúvida, John von Neumann foi um cientista brilhante que merece crédito por muitas contribuições. Contudo, a contribuição pela qual a História popular escolheu dar-lhe o crédito, o conceito de programa armazenado, foi aparentemente desenvolvida por pesquisadores liderados por J. P. Eckert, da Moore School of Electrical Engineering, University of Pennsylvania. John von Neumann foi meramente o primeiro a publicar um trabalho sobre a idéia e, assim, os eruditos da computação o selecionaram como inventor.

2.2 Linguagem de máquina

Para aplicar o conceito de programa armazenado, os processadores são projetados para reconhecer as instruções codificadas como padrões de *bits*. A coleção de instruções, juntamente com o sistema de codificação, é denominada **linguagem de máquina**. Uma instrução expressa nessa linguagem é chamada instrução em nível de máquina, ou mas comumente uma **instrução de máquina**.

O repertório de instruções

É surpreendente que a lista de instruções de máquina que uma UCP comum deve decodificar e executar é bem pequena. Sem dúvida, um dos aspectos fascinantes da ciência da computação é que se uma máquina pode executar certas tarefas elementares, mas bem escolhidas, ao adicionar novos recursos a ela, isso não aumenta suas capacidades teóricas. Em outras palavras, além de um certo ponto, as novas características podem aumentar certas facilidades, como a conveniência, mas não acrescentam coisa alguma às habilidades básicas da máquina. A decisão em relação ao aproveitamento desse fato tem levado a duas filosofias de projeto para a UCP. Uma delas é que a UCP deve executar um conjunto mínimo de instruções de máquina. A abordagem leva ao chamado **computador com conjunto mínimo de instruções** (*reduced instruction set computer* — RISC). O argumento em favor da arquitetura RISC é que essas máquinas são eficientes e rápidas. No entanto, argumenta-se a favor de UCPs com capacidade para executar muitas instruções complexas, ainda que muitas delas sejam tecnicamente redundantes. O resultado dessa abordagem é conhecido como **computador com conjunto de instruções complexas** (*complex instruction set computer* — CISC). O argumento em favor da arquitetura CISC é que a UCP mais complexa é mais fácil de programar, uma vez que uma única instrução é capaz de desempenhar uma tarefa que necessitaria uma seqüência de múltiplas instruções em um projeto RISC.

Memória cache (escondida)

É instrutivo considerar os registradores de propósito geral da UCP como recursos globais de memória de um computador. Os registradores são usados para manter os dados imediatamente aplicáveis à manipulação do momento; a memória principal é usada para manter os dados que serão necessários em um futuro próximo, e o armazenamento em massa, para manter os dados que dificilmente serão necessários no futuro próximo. Muitas máquinas são projetadas com um nível adicional de memória, chamada **memória cache**, uma porção (às vezes vários KB) de memória de alta velocidade com tempo de resposta comparável ao dos registradores da UCP, normalmente localizada no próprio processador. Nessa área especial de memória, a máquina procura manter uma cópia de parte da memória principal que é de interesse corrente. Com isso, as transferências de dados que normalmente seriam entre registradores e memória principal são feitas entre os registradores e a memória *cache*. Quaisquer alterações são coletivamente transferidas para a memória em um momento mais oportuno. O resultado é uma máquina cujo ciclo pode ser executado mais rapidamente, o que aumenta a velocidade de execução das tarefas.

Os processadores CISC e RISC estão disponíveis comercialmente. A série Pentium de processadores, desenvolvida pela Intel, é exemplo de arquitetura CISC; as séries de processadores PowerPC, desenvolvidas pela Apple Computer, IBM e Motorola são exemplos de arquitetura RISC. Para facilitar a apresentação, enfocaremos a abordagem RISC neste capítulo.

Ao se discutir as instruções no repertório de uma máquina, é útil reconhecer que elas podem ser classificadas em três categorias: (1) o grupo de transferência de dados, (2) o grupo aritmético/lógico e (3) o grupo de controle.

Transferência de dados O primeiro grupo consiste em instruções que promovem a movimentação de dados de um local para outro. Os passos 1, 2 e 4 da Figura 2.2 pertencem a esta categoria. Como acontece na memória principal, não é comum os dados que estão sendo transferidos de um local para outro serem apagados no local original. Uma instrução de transferência está mais associada à idéia da cópia dos dados do que à da mudança física de posição. Logo, os termos *transferência* ou *mudança* geralmente empregados na verdade não são muito apropriados; termos como *cópia* ou *clone* descreveriam melhor tal operação.

Ainda com relação à terminologia, deveríamos mencionar a existência de termos específicos referentes a transferências de dados entre a UCP e a memória principal. Um pedido para preencher um registrador de propósito geral com os dados de uma célula de memória é comumente chamado *carga* (LOAD); de maneira recíproca, um pedido para transferir os dados de um registrador para uma posição de memória é denominado *armazenamento* (STORE). Na Figura 2.2, os passos 1 e 2 são instruções de carga, e o passo 4 é uma instrução de armazenamento.

Um grupo importante de instruções da categoria de transferência de dados refere-se aos comandos de comunicação com dispositivos externos ao par processador — memória principal (impresora, teclado, monitor, unidade de disco etc.). Já que tais instruções manipulam ações de entrada ou saída do computador, são classificadas como instruções de E/S e, às vezes, consideradas uma categoria à parte. No entanto, na Seção 2.5, demonstraremos como as operações de E/S freqüentemente são manipuladas pelas mesmas instruções que solicitam transferências de dados entre o processador e a memória principal. Assim, consideraremos as instruções de E/S como parte do grupo de transferência de dados.

Instruções aritméticas/lógicas O grupo de instruções aritméticas/lógicas consiste em instruções que dizem à unidade de controle para desencadear atividades na unidade aritmética/lógica. O passo 3 da Figura 2.2 está nesse grupo. Como o próprio nome sugere, a unidade aritmética/lógica destina-se a executar operações outras além das operações aritméticas básicas. Algumas destas operações adicionais são as operações lógicas AND, OR e XOR, que introduzimos no Capítulo 1 e discutiremos com mais detalhes neste capítulo. Em geral, essas operações são utilizadas para manipular bits independentemente em um registrador de propósito geral, sem interferir no restante do registrador. Outro conjunto de operações, disponível na maioria das unidades aritméticas e lógicas, permite deslocar o conjunto dos bits dos registradores para a direita ou para a esquerda. Essas operações são conhecidas como operações de deslocamento simples (SHIFT) se, em decorrência da execução da operação, forem descartados os bits que “caírem para fora” do registrador. São chamadas de operações de rotação (ROTATE) quando os bits são reutilizados para preencher as lacunas deixadas na extremidade oposta do registrador como resultado da execução desta operação.

Controle O grupo de controle consiste em instruções que tratam da execução do programa, em vez da manipulação de dados. Embora seja extremamente elementar, o passo 5 da Figura 2.2 ilustra esta categoria de instruções. O grupo de instruções de controle engloba muitas das instruções mais interessantes disponíveis em um computador, como a família de instruções de desvio (JUMP ou BRANCH), utilizadas para orientar a unidade de controle a executar uma instrução que não seja a próxima da lista. Essas instruções podem ser de dois tipos: desvios incondicionais e desvios condicionais. Um exemplo do primeiro seria “Desvie para a instrução de número 5”, e um do segundo, “Se o valor obtido for 0, então desvie para a instrução de número 5”. A diferença é que um desvio condicional resultará em uma alteração no rumo do programa se — e somente se — uma certa condição for satisfeita. Como exemplo, a sequência de instruções da Figura 2.3 representa um algoritmo de divisão de dois números, no qual o passo 3 é um desvio condicional, cuja finalidade é evitar a possibilidade de ocorrência de uma divisão por zero.

Uma linguagem de máquina ilustrativa

Verifiquemos de que maneira podem ser codificadas as instruções de uma linguagem de máquina comum. O computador que usaremos para a nossa discussão está resumido na Figura 2.4 e descrito mais detalhadamente

- Passo 1.** CARREGUE um registrador com um valor da memória.
- Passo 2.** CARREGUE outro registrador com outro valor da memória.
- Passo 3.** Se o segundo valor for zero, DESVIE para o passo 6.
- Passo 4.** Divida o conteúdo do primeiro registrador pelo segundo e guarde o resultado em um terceiro registrador.
- Passo 5.** GUARDE na memória o conteúdo do terceiro registrador.
- Passo 6.** PARE.

Figura 2.3 Roteiro para a divisão de dois números armazenados na memória.

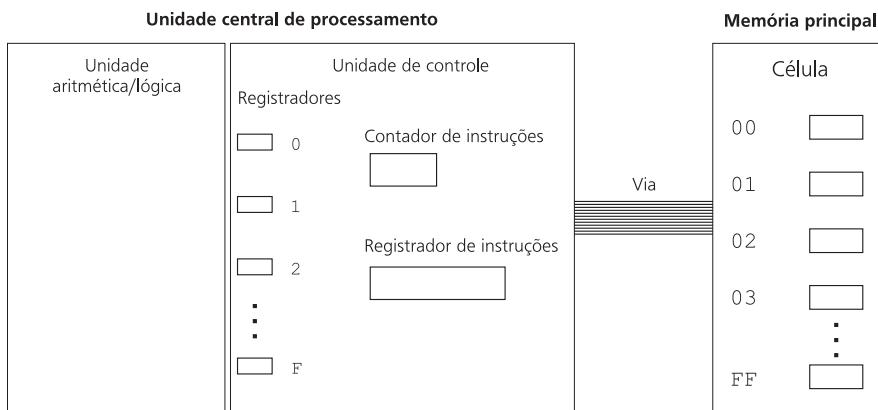


Figura 2.4
A arquitetura do computador usado no Apêndice C.

no Apêndice C. Possui 16 registradores de propósito geral e 256 células na memória principal, cada qual com capacidade de oito bits. Para fins de referência, rotulamos os registradores com os valores de 0 a 15 e endereçamos as células com os valores de 0 a 255. Por conveniência, representamos esses rótulos e endereços com valores na base dois e comprimimos os padrões de bits usando a notação hexadecimal. Assim, os registradores são rotulados com 0 a F e as células de memória, endereçadas com 0 a FF.

A versão codificada de uma instrução de máquina normalmente é composta de duas partes: o **campo do código de operação** e o **campo do operando**. O padrão de bits que aparece no campo do código de operação indica qual das operações elementares, tais como STORE, SHIFT, XOR e JUMP, está sendo especificada pela instrução. Os padrões de bits encontrados no campo do operando complementam a informação sobre a operação indicada pelo código de operação. Por exemplo, no caso da operação STORE, a informação no campo de operando indica qual registrador contém os dados a serem armazenados e qual a posição da memória que deverá receber tais dados.

A linguagem de máquina completa do nosso computador (Apêndice C) consiste em apenas 12 instruções básicas. Cada uma é codificada usando um total de 16 bits, representados por quatro dígitos hexadecimais (Figura 2.5). O código de operação correspondente a cada instrução consiste nos quatro primeiros bits ou, de maneira equivalente, no primeiro dígito hexadecimal. Note (Apêndice C) que esses códigos de operação são representados pelos dígitos hexadecimais entre 1 e C, inclusive. Mais especificamente, a tabela no Apêndice C nos diz que uma instrução iniciada com o dígito hexadecimal 3 se refere a uma instrução STORE, e qualquer código de instrução que comece com o dígito hexadecimal A corresponde a uma instrução ROTATE.

Observando o campo do operando, percebe-se que ele consiste em três dígitos hexadecimais (12 bits) e, em cada caso (com exceção da instrução STOP, que não precisa de qualquer refinamento adicional), complementa a instrução geral especificada pelo código de operação. Por exemplo (Figura 2.6), se o primeiro dígito hexadecimal de uma instrução for 3 (código da operação STORE, para armazenar o conteúdo de um registrador), o próximo dígito hexadecimal da instrução indicará qual dos registradores será utilizado, e os dois últimos dígitos hexadecimais indicarão a posição de memória que receberá os dados. Assim, a instrução 35A7 (em hexade-

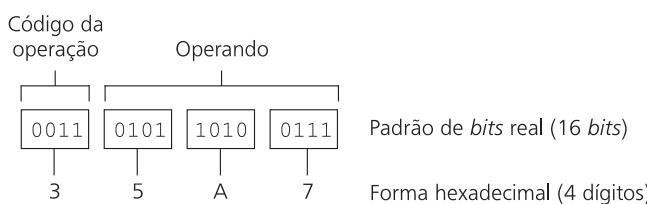


Figura 2.5 A composição de uma instrução para a máquina do Apêndice C.

cimal) é traduzida como “Armazene o padrão de bits encontrado no registrador 3 na célula de memória cujo endereço é A7”.

Outro exemplo, o código de operação 7 (em hexadecimal), especifica que seja aplicada a operação OR sobre o conteúdo de dois registradores. (Veremos mais tarde na Seção 2.4 o que isso significa, por ora nosso interesse é meramente a codificação das instruções.) Nesse caso, o dígito

hexadecimal seguinte indica onde o resultado deverá ser colocado, enquanto os dois últimos dígitos hexadecimais do campo do operando são utilizados para indicar quais serão os dois registradores cujos conteúdos irão sofrer a ação da operação OR. Assim, a instrução 70C5 equivale ao comando “Efetue a operação lógica OR sobre os conteúdos dos registradores C e 5 e coloque o resultado no registrador 0”.

Existe uma distinção sutil entre as duas instruções LOAD da nossa máquina. Aqui, o código de operação 1 (hexadecimal) se refere à instrução que carrega um registrador com o conteúdo de uma posição de memória, enquanto o código de operação 2 (hexadecimal) refere-se à instrução que carrega um registrador com um dado valor. A diferença é que o campo do operando de uma instrução do primeiro tipo contém um endereço, enquanto o do segundo tipo contém o próprio padrão de bits a ser carregado.

A máquina tem duas instruções de adição (ADD), uma para somar números representados em complemento de dois e outra para somar representações em vírgula flutuante. Esta distinção resulta do fato de que efetuar a adição de padrões de bits em notação binária exige da unidade aritmética/lógica ações diferentes de quando se efetuam adições em notação de vírgula flutuante.

Encerramos esta seção com a Figura 2.7, que contém uma versão codificada das instruções na Figura 2.2. Pressupomos que os valores a serem somados estão armazenados na notação de complemento de dois nos endereços de memória 6C e 6D, e que a soma deve ser posta na célula de memória no endereço 6E.

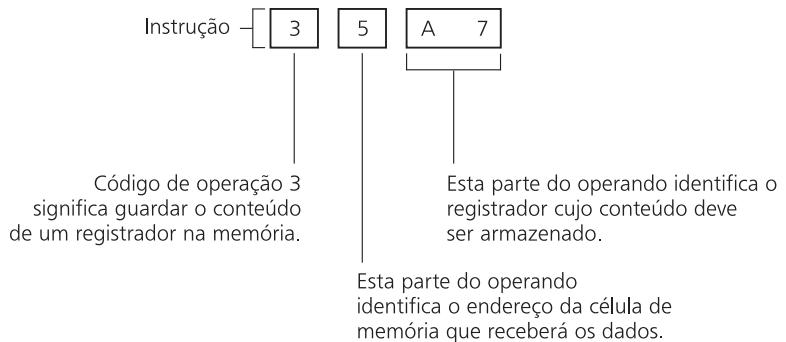


Figura 2.6 Decodificação da instrução 35A7.

Instrução codificada	Tradução
156C	Carregue o registrador 5 com o padrão de bits encontrado na célula de memória no endereço 6C.
166D	Carregue o registrador 6 com o padrão de bits encontrado na célula de memória no endereço 6D.
5056	Some o conteúdo dos registradores 5 e 6 considerando que representam números na notação de complemento de dois e deixe o resultado no registrador 0.
306E	Armazene o conteúdo do registrador 0 na célula de memória de endereço 6E.
C000	Pare.

Figura 2.7 Uma versão codificada das instruções da Figura 2.2.



QUESTÕES/EXERCÍCIOS

1. Por que o termo mover é considerado incorreto para denominar a operação de transferência de dados de uma posição do computador para outra?
2. No texto, as instruções de desvio (JUMP) foram expressas identificando, de forma explícita, a instrução-alvo pelo nome (ou o número do passo) destinatário (por exemplo, “Desvie para o passo 6”). A desvantagem desta técnica é que, se futuramente o nome (ou número) da instrução for alterado, deveremos localizar todos os desvios para tal instrução e alterar o nome referenciado, para que aponte para o destino correto. Descreva outro modo de expressar uma instrução como essa, de forma que o nome da instrução-alvo não seja explicitamente declarado.
3. A instrução “Se 0 for igual a 0, então desvie para o passo 7” é um desvio condicional ou incondicional? Explique sua resposta.
4. Escreva, em padrões binários, o programa exemplo da Figura 2.7.
5. As seguintes instruções foram codificadas na linguagem de máquina descrita no Apêndice C. Reescreva-as em português.
 - a. 368A
 - b. BADE
 - c. 803C
 - d. 40F4
6. Qual a diferença entre as instruções 15AB e 25AB na linguagem de máquina do Apêndice C?
7. Eis algumas instruções escritas em português. Traduza-as para a linguagem de máquina do Apêndice C.
 - a. Carregue o registrador número 3 com o número hexadecimal 56.
 - b. Rode, três bits à direita, o registrador número 5.
 - c. Execute uma operação AND entre o conteúdo do registrador A e o conteúdo do registrador 5 e coloque o resultado no registrador 0.

2.3 Execução de programas

Um computador efetua a execução de um programa armazenado em sua memória copiando, sempre que necessário, as instruções da memória para a unidade de controle. Uma vez na unidade de controle, cada instrução é decodificada e executada. A ordem em que as instruções são trazidas da memória corresponde à ordem na qual elas estão armazenadas na memória, exceto se algo em contrário for especificado por meio de uma instrução JUMP de desvio.

Para compreender o funcionamento de todo o processo de execução, é necessário observar mais detalhadamente a unidade de controle, no interior da UCP. Nela, estão dois registradores de propósito específico: o **contador de instruções**^{*} e o **registrador de instruções** (Figura 2.4). O contador de instruções contém o endereço da próxima instrução a ser executada, servindo como instrumento para o computador manter-se informado sobre a posição do programa em que está ocorrendo a execução. O registrador de instruções é usado para manter a instrução que estiver sendo executada.

A unidade de controle executa seu trabalho repetindo, continuamente, um algoritmo, o **ciclo de máquina**, que consiste em três passos: busca, decodificação e execução (Figura 2.8). Durante o passo de busca, a unidade de controle solicita que a memória principal forneça a instrução armazenada no endereço indicado pelo contador de instruções. Uma vez que cada instrução da nossa máquina possui dois bytes, esse processo de busca inclui a transferência de duas células da memória principal. A unidade de controle guarda a instrução recebida da memória em seu registrador de instruções e então aumenta

*N. de T. Em inglês, *program counter*.

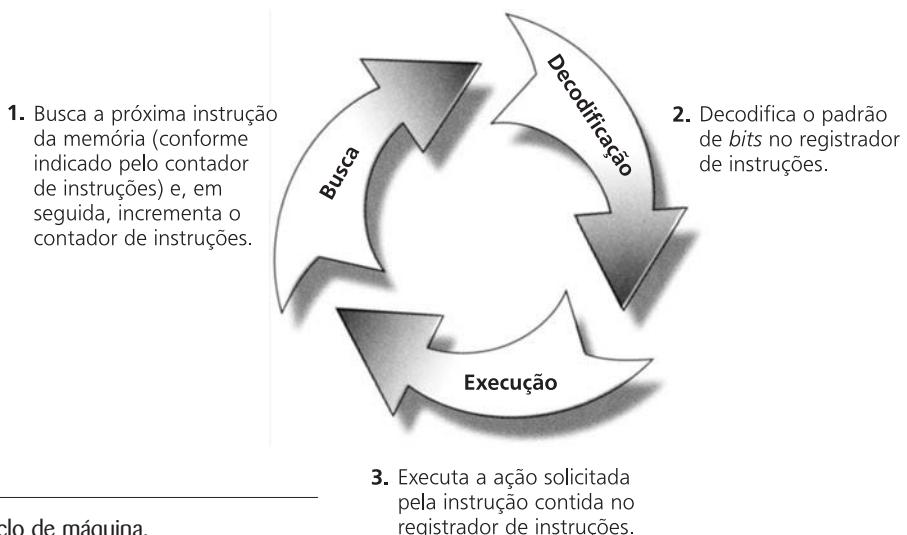


Figura 2.8 O ciclo de máquina.

dois o contador de instruções de modo que este passasse a conter o endereço da próxima instrução armazenada na memória. Assim, o contador de instruções estará pronto para ser usado durante a próxima busca.

Agora, dispondo da instrução em seu registrador de instruções, a unidade de controle inicia a fase da decodificação, que envolve a separação dos componentes do campo de operando, de acordo com o código da operação.

A unidade de controle então executa a instrução, ativando os circuitos necessários para a realização da tarefa. Por exemplo, se a instrução solicitar que se carregue um registrador com um dado da memória, a unidade de controle ativará tal carregamento; se a instrução corresponder a uma operação aritmética, ela ativará o circuito apropriado da unidade aritmética/lógica, alimentando-o com os registradores indicados como os operandos da instrução.

Quando a instrução tiver sido totalmente executada, a unidade de controle recomeçará o ciclo, partindo do passo de busca. Observe-se que, como o contador de instruções, foi incrementado ao término do passo de busca anterior, ele agora indicará novamente para a unidade de controle o endereço correto da próxima instrução a ser executada.

Um caso muito especial refere-se à execução de uma instrução de desvio (JUMP). Por exemplo, considere a instrução B258 (Figura 2.9) que significa “Desvie para a instrução do endereço 58 se o conteúdo do registrador 2 for igual ao do registrador 0”. Neste caso, o passo de execução do ciclo de máquina inicia com a comparação dos conteúdos dos registradores 2 e 0. Se forem diferentes, o passo de execução terminará, e iniciar-se-á a próxima busca. Contudo, se seus conteúdos forem iguais, a máquina colocará o valor 58 no contador de instruções durante o passo de execução. Nesse caso, então, o próximo passo de busca encontrará 58 no contador de instruções; assim, a instrução contida nesse endereço será a próxima a ser buscada e executada.

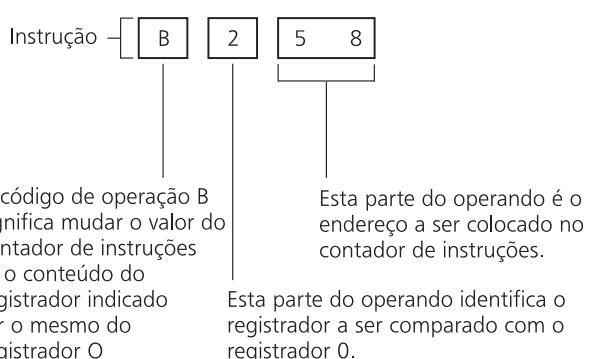


Figura 2.9 Decodificação da instrução B258.

Note que se a instrução fosse B058, então a decisão de alterar o contador de instruções dependeria da igualdade de conteúdos entre o registrador 0 e o registrador 0. Todavia, esses são o mesmo registrador, portanto, devem ter o mesmo conteúdo. Logo, qualquer instrução na forma B0XY causará um desvio para a posição de memória com endereço XY.

Um exemplo de execução de programa

Sigamos o ciclo de máquina percorrido ao executar o programa apresentado na Figura 2.7, que toma dois valores da memória, computa a sua soma e armazena o total em outra célula de memória. Primeiro, colocamos o programa em alguma área da memória. No nosso exemplo, ele é armazenado em endereços sucessivos a partir do endereço hexadecimal A0. Com o programa armazenado desta forma, colocamos o endereço da primeira instrução (A0) no contador de instruções e acionamos a máquina (Figura 2.10).

A unidade de controle começa o passo de busca extraiendo a instrução (156C) da posição A0 e colocando-a em seu registrador de instruções (Figura 2.11a). Note-se que, no nosso computador, as instruções possuem 16 bits (dois bytes). Logo, a instrução que está sendo buscada ocupa duas posições de memória, nos endereços A0 e A1. A unidade de controle deve estar projetada para aceitar tal situação. Desta forma, ela lê o conteúdo de tais posições de memória e o deposita adequadamente no registrador de instruções, de 16 bits. A seguir, a unidade de controle soma 2 ao conteúdo do contador de instruções, de forma que este passe a conter o endereço da próxima instrução a ser executada (Figura 2.11b). Ao término do passo de busca do primeiro ciclo de máquina, o contador e o registrador de instruções apresentam os seguintes dados:

Contador de instruções: A2

Registrador de instruções: 156C

Em seguida, a unidade de controle analisa a instrução existente no seu registrador de instruções e conclui que deve carregar no registrador 5 o conteúdo da posição de memória de endereço 6C. Este carregamento é executado durante o passo de execução, e então a unidade de controle inicia um novo ciclo.

O contador de instruções contém os endereços das primeiras instruções.

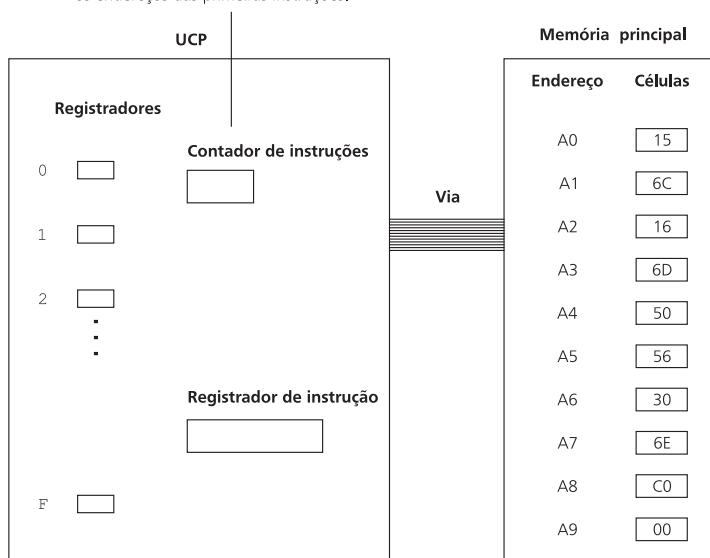
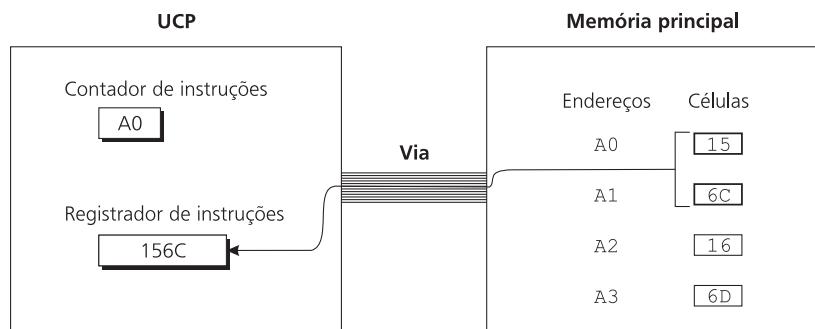
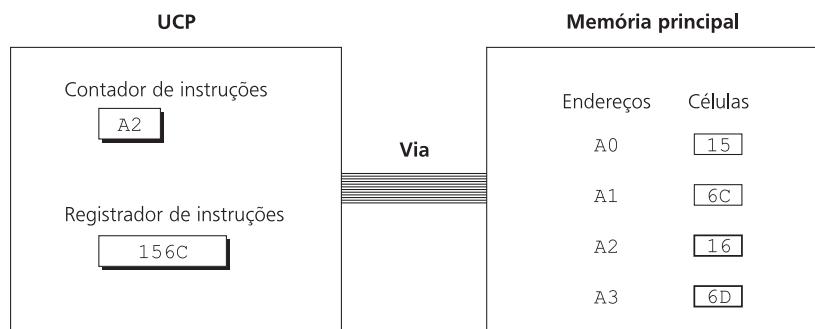


Figura 2.10 Programa da Figura 2.7 armazenado na memória pronto para a execução.

O programa é armazenado na memória iniciada no endereço A0.



- a. No início do passo de busca, a instrução iniciada no endereço A0 é trazida da memória e colocada no registrador de instruções.



- b. Então o contador de instruções é incrementado, de modo a apontar para a próxima instrução.

Figura 2.11
Executando o passo de busca do ciclo de máquina.

Esse ciclo inicia buscando a instrução 166D nas posições de memória de endereços A2 e seguinte. A unidade de controle coloca esta instrução em seu registrador de instruções e incrementa o contador para A4. Os valores contidos no contador e no registrador de instruções serão:

Contador de instruções: A4
Registrador de instruções: 166D

Agora, a unidade de controle decodificará a instrução 166D e determinará o carregamento, no registrador 6, do conteúdo da memória de endereço 6D. A seguir, entrará no passo de execução, quando o registrador 6 será devidamente carregado.

Como o contador de instruções agora contém A4, a unidade de controle obterá a próxima instrução a partir de tal endereço. Como resultado, o número 5056 é depositado no registrador de instruções, sendo o contador de instruções incrementado para A6. Em seguida,

Instruções de tamanho variável

Para simplificar a explanação do texto, a linguagem de máquina descrita no Apêndice C usa tamanho fixo (dois bytes) para todas as instruções; a UCP sempre traz o conteúdo de duas células consecutivas da memória e aumenta o contador de instruções em dois. Na realidade, a maioria das linguagens de máquina usa instruções de tamanho diferentes. A série Pentium, por exemplo, possui instruções com tamanhos que variam de um único byte até múltiplos bytes, dependendo do uso exato da instrução. As UCPs com essas linguagens de máquina determinam o tamanho da instrução pelo seu código de operação, isto é, a UCP primeiramente busca o código da operação da instrução e então, baseada no padrão de bits recebido, sabe quantos bytes devem ser buscados da memória para obter o restante da instrução.

unidade de controle decodifica o conteúdo do seu registrador de instruções, entra no passo de execução e ativa o circuito de adição em complemento de dois, tendo como dados de entrada os registradores 5 e 6.

Durante este passo, a unidade aritmética/lógica executa a adição, deposita o resultado no registrador 0 (conforme solicitado pela unidade de controle) e informa à unidade de controle que a sua tarefa está terminada. A unidade de controle começa então outro ciclo de máquina. Mais uma vez, com a ajuda do contador de instruções, vai buscar a próxima instrução (306E), localizada nas posições de memória de endereços A6 e A7, e incrementa o contador de instruções para A8. A instrução é então decodificada e executada. Neste ponto, a soma obtida é depositada na posição de memória 6E.

A próxima instrução é obtida na posição de memória A8 e o contador de instruções é incrementado para AA. O conteúdo do registrador de instruções (C000) é decodificado como uma instrução de parada. Em consequência, o computador pára durante o próximo passo de execução do ciclo de máquina e a execução do programa se completa.

Em resumo, verificamos que a execução de um programa armazenado na memória envolve o mesmo processo que um de nós usaria para seguir uma lista detalhada de instruções. Da mesma forma que se necessita cuidar para não perder o ponto da lista enquanto se efetuam as operações indicadas, o computador faz uso do contador de instruções para esta finalidade. Determinada a próxima instrução a executar, ela é lida, e o seu significado, extraído. Finalmente, a tarefa solicitada é executada, e retorna-se à lista, em busca da próxima instrução, da mesma forma como o computador executa suas instruções durante o passo de execução, prosseguindo com a próxima busca.

Programas versus dados

Muitos programas podem ser simultaneamente armazenados na memória principal de um computador, desde que ocupem posições diferentes. Assim, é fácil escolher o programa que deve ser executado ao ligar o computador, fixando-se adequadamente um valor para o contador de instruções.

Todavia, deve-se considerar que, como os dados também estão colocados na memória e codificados como cadeias de 0s e 1s, o computador não tem como distinguir os dados dos programas. Se for atribuído ao contador o endereço de algum dado em vez do endereço de uma instrução do programa e o computador não dispuser de outra informação, interpretará esses dados, escritos em padrões de bits, como instruções e os executará. O resultado dessa execução dependerá, portanto, dos dados envolvidos.

Não devemos concluir que o fato de programas e dados terem uma aparência comum na memória da máquina seja de todo ruim. Na verdade, ele tem demonstrado ser um atributo útil, pois permite ao programa manipular outros programas (ou mesmo a si próprio) como se fossem dados. Imagine, por exemplo, um programa que se automodifique em resposta a sua interação com o ambiente e assim apresente a capacidade de aprender, ou talvez um programa que escreva e execute outros programas com o propósito de resolver os problemas apresentados a ele.



QUESTÕES/EXERCÍCIOS

- Suponha que as células de memória nos endereços 00 a 05, do computador descrito no Apêndice C, contenham os padrões hexadecimais de bits da seguinte tabela:

Endereço	Conteúdo
00	14
01	02
02	34
03	17
04	C0
05	00

Se ativarmos o computador e o seu contador de instruções contiver 00, qual será o padrão de bits na posição de memória de endereço hexadecimal 17 quando o computador parar?

2. Suponha que as células de memória nos endereços B0 a B8, do computador descrito no Apêndice C, contenham os padrões hexadecimais de bits da seguinte tabela:

Endereço	Conteúdo
B0	13
B1	B8
B2	A3
B3	02
B4	33
B5	B8
B6	C0
B7	00
B8	0F

- a. Se o contador de instruções começar com B0, qual será o padrão de bits no registrador número 3 depois de executada a primeira instrução?
 b. Qual será o padrão de bits na posição de memória B8 quando a instrução de parada for executada?
3. Suponha que as células de memória nos endereços de A4 a B1, do computador descrito no Apêndice C, contenham os padrões hexadecimais da seguinte tabela:

Endereço	Conteúdo
A4	20
A5	00
A6	21
A7	03
A8	22
A9	01
AA	B1
AB	B0
AC	50
AD	02
AE	B0
AF	AA
B0	C0
B1	00

Responda às seguintes questões, pressupondo que o computador inicie enquanto o seu contador de instruções contenha A4:

- a. O que há no registrador 0 na primeira vez em que a instrução de endereço AA foi executada?
 b. O que há no registrador 0 na segunda vez em que a instrução de endereço AA foi executada?
 c. Quantas vezes a instrução do endereço AA é executada antes de o computador parar?
4. Suponha que as células de memória nos endereços de F0 a F9, do computador descrito no Apêndice C, contenham os padrões hexadecimais de bits listados na seguinte tabela:

Endereço	Conteúdo
F0	20
F1	C0

F2	30
F3	F8
F4	20
F5	00
F6	30
F7	F9
F8	FF
F9	FF

Se iniciarmos o computador e seu contador de instruções contiver F0, o que ele executará ao alcançar a instrução do endereço F8?

2.4 Instruções aritméticas e lógicas

Conforme foi mencionado anteriormente, o grupo das instruções aritméticas e lógicas compõe-se de instruções que solicitam operações aritméticas, lógicas e de deslocamentos (*shifts*). Nesta seção, analisaremos com maior atenção tais operações.

Operações lógicas

No Capítulo 1, foram apresentadas as operações lógicas AND (e), OR (ou) e XOR (ou exclusivo), como operações que combinam dois *bits* de entrada para produzir um único *bit* de saída. Essas operações podem ser estendidas de forma que combinem duas cadeias de *bits* para produzir uma única cadeia de saída, aplicando, para tanto, a operação básica para cada coluna individual. Por exemplo, o resultado da operação AND dos padrões 10011010 e 11001001 será:

$$\begin{array}{r} 10011010 \\ \text{AND} \quad 11001001 \\ \hline 10001000 \end{array}$$

onde apenas escrevemos, na base de cada coluna de dois dígitos, o resultado da operação AND dos dois *bits* em cada coluna. Do mesmo modo, efetuando-se as operações OR e XOR nestes mesmos moldes, produzir-se-iam:

$$\begin{array}{r} 10011010 \\ \text{OR} \quad 11001001 \\ \hline 11011011 \end{array} \qquad \begin{array}{r} 10011010 \\ \text{XOR} \quad 11001001 \\ \hline 01010011 \end{array}$$

Uma das principais aplicações do operador AND é para colocar Os em uma parte de um padrão de *bits* sem alterar a outra parte. Por exemplo, suponha que seja o byte 00001111 o primeiro operando da operação AND. Mesmo desconhecendo o conteúdo do segundo operando, podemos concluir que os quatro *bits* mais significativos do resultado são Os. Além disso, que os quatro *bits* menos significativos do resultado são uma cópia da parte correspondente do segundo operando, como demonstrado no seguinte exemplo:

$$\begin{array}{r} 00001111 \\ \text{AND} \quad 10101010 \\ \hline 00001010 \end{array}$$

Esta utilização do operador AND exemplifica a técnica denominada *mascaramento* (*masking*), em que um operando, chamado **máscara**, determina a parte do outro operando que irá afetar o resultado. No caso da operação AND, a máscara produz uma saída que é uma réplica parcial de um dos operandos, cujas posições não-duplicadas são preenchidas com Os.

Essa operação é útil na manipulação de **mapas de bits**, que constituem cadeias de *bits* em que cada um representa a presença ou a ausência de um objeto. Já vimos os mapas de *bits* no contexto da representação de imagens, onde cada *bit* é associado a um *pixel*. Como outro exemplo, uma cadeia de 52 *bits*, em que cada um está associado a uma carta de um baralho, pode ser usada para representar a mão de um jogador de pôquer. Para tanto, basta associar 1s aos 5 *bits* que representam as cartas da mão e 0s a todos os demais. Do mesmo modo, um mapa de 52 *bits*, dos quais 13 são 1s, pode ser utilizado para representar uma mão em um jogo de bridge e um mapa de 32 *bits*, para representar os 32 tipos disponíveis de sabores de sorvete.

Suponhamos, então, que uma célula de memória de oito *bits* esteja sendo usada como mapa de *bits*, e que desejemos saber se o objeto associado ao terceiro *bit* mais significativo está presente. Precisamos apenas aplicar a operação AND entre o *byte* completo e a máscara 00100000, o que resultará em um *byte* só de 0s, se, e somente se, o terceiro *bit* mais significativo do mapa for 0. O programa pode, então, tomar a ação apropriada, se a operação AND for seguida de uma instrução condicional. Além disso, se o terceiro *bit* mais significativo for 1 e desejarmos alterá-lo para 0, mas sem modificar os demais *bits*, poderemos efetuar a operação AND entre o mapa de *bits* e a máscara 11011111 e então armazenar o resultado no mapa de *bits* original.

Enquanto o operador AND pode ser utilizado para duplicar uma parte de uma cadeia e ao mesmo tempo preencher com 0s a parte não-duplicada, o operador OR pode ser usado para duplicar uma parte de uma cadeia de *bits*, preenchendo com 1s a parte não-duplicada. Para isso, novamente usamos uma máscara, mas, desta vez, indicamos com 0s a posição dos *bits* a serem duplicados e com 1s as posições não-duplicadas. Por exemplo, aplicar o operador OR entre qualquer *byte* e a máscara 11110000 produz um resultado com 1s nos seus quatro *bits* mais significativos, enquanto os demais *bits* conterão uma cópia dos quatro *bits* menos significativos do outro operando, como demonstra o seguinte exemplo:

$$\begin{array}{r} 11110000 \\ \text{OR} \quad 10101010 \\ \hline 11111010 \end{array}$$

Como consequência, enquanto a máscara 11011111 pode ser usada pelo operador AND para forçar a inserção de um 0 no terceiro *bit* mais significativo de um mapa de oito *bits*, a operação OR com a máscara 00100000 pode sê-lo para forçar a inserção de um 1 naquela posição.

A principal aplicação do operador XOR está em formar o complemento de uma cadeia de *bits*. Por exemplo, note a relação existente entre o segundo operando e o resultado do seguinte exemplo:

$$\begin{array}{r} 11111111 \\ \text{XOR} \quad 10101010 \\ \hline 01010101 \end{array}$$

Comparação da potência de computadores

Quando se compra um computador pessoal, a freqüência do relógio é comumente usada para comparar as máquinas. O **relógio** de um computador é um circuito oscilador que gera pulsos usados para coordenar as atividades da máquina – quanto mais rápido for esse circuito oscilador, mais rapidamente a máquina executa o seu ciclo. A velocidade do relógio é medida em hertz (abreviado com Hz) – um Hz é igual a um ciclo (ou pulso) por segundo. As velocidades normais nos computadores de mesa estão na faixa de algumas centenas de MHz (modelos antigos) a vários GHz. (MHz é abreviação de megahertz, que é um milhão de Hz. GHz é abreviação de gigahertz, que é 1000 MHz.)

Infelizmente, projetos diferentes de UCP podem realizar quantidades de trabalho diferentes em cada ciclo de máquina. Assim sendo, a velocidade do relógio sozinha deixa de ser relevante quando se compara máquinas com UCPs diferentes. Quando se deseja comparar uma máquina baseada no PowerPC com outra baseada no Pentium, é mais significativo comparar os desempenhos por meio de *benchmarking*, que é o processo de comparar o desempenho de máquinas diferentes executando o mesmo programa, conhecido como **amostra de teste** (*benchmark*). Selecionando-se as amostras de teste que representam diferentes tipos de aplicação, as comparações obtidas podem ser significativas para diferentes segmentos do mercado. A melhor máquina para uma aplicação pode não ser a melhor para outra.

Aplicar o operador XOR entre qualquer *byte* e uma máscara de 1s produz o complemento do primeiro *byte*.

Na linguagem de máquina descrita no Apêndice C, os códigos de operação 7, 8 e 9 são usados para as operações lógicas OR, AND e XOR, respectivamente. Cada um deles especifica que a operação correspondente deve ser efetuada com o conteúdo de dois registradores designados e o resultado, colocado em outro registrador. Por exemplo, a instrução 7ABC indica que o resultado da operação AND nos registradores B e C deve ser colocado no registrador A.

Operações de rotação e deslocamento

As operações do tipo rotação e deslocamento permitem movimentar *bits* no interior de um registrador e freqüentemente são utilizadas para resolver problemas de alinhamento, como preparar um *byte* para usos futuros em operações que envolvam máscaras ou na manipulação da mantissa das representações em vírgula flutuante. Tais operações são classificadas de acordo com a direção de seu movimento (à direita ou à esquerda) e o fato de o deslocamento ser ou não circular. Neste esquema de classificação existem misturas de terminologia muito diversas. Analisemos rapidamente as idéias envolvidas.

Se iniciarmos com uma seqüência de *bits* com um *byte* de comprimento e aplicarmos a operação de deslocamento (*shift*) de um *bit* para a direita ou para a esquerda ao seu conteúdo, poderemos imaginar um *bit* saindo para fora do *byte* por uma das extremidades e um vazio aparecendo na extremidade oposta. O que é feito com este *bit* perdido e com o vazio decorrente é que distingue os vários tipos de operações de deslocamento. Uma das possibilidades consiste em depositar o *bit* extra na lacuna formada na outra extremidade. O resultado é um deslocamento circular, também conhecido como rotação. Assim, se executarmos oito vezes um deslocamento circular à direita em um dado *byte*, obteremos o mesmo padrão de *bits* com o qual iniciamos.

Outra possibilidade é descartar o *bit* perdido, preenchendo cada lacuna resultante com um 0. O termo **deslocamento lógico** é empregado para designar tais operações. Um deslocamento lógico à esquerda pode ser utilizado para multiplicar números em notação de complemento de dois por 2. Afinal de contas, nessa notação, deslocar dígitos binários à esquerda corresponde a multiplicar por 2, e um deslocamento similar com dígitos decimais corresponde a multiplicar por 10. Do mesmo modo, a divisão inteira por 2 pode ser realizada deslocando-se para a direita a cadeia binária. Nos dois tipos de deslocamento, é preciso tomar alguns cuidados para que o *bit* de sinal seja preservado quando se usam certos

sistemas de notação. Por essa razão, freqüentemente encontramos deslocamentos à direita que sempre preenchem a lacuna (que fica na posição do *bit* de sinal) não com zeros, mas com seu valor original. Os deslocamentos que mantêm inalterado o *bit* de sinal às vezes são chamados de **deslocamentos aritméticos**.

Dentre as diversas possibilidades de instruções de deslocamento e rotação, a linguagem de máquina descrita no Apêndice C contém apenas uma instrução de deslocamento

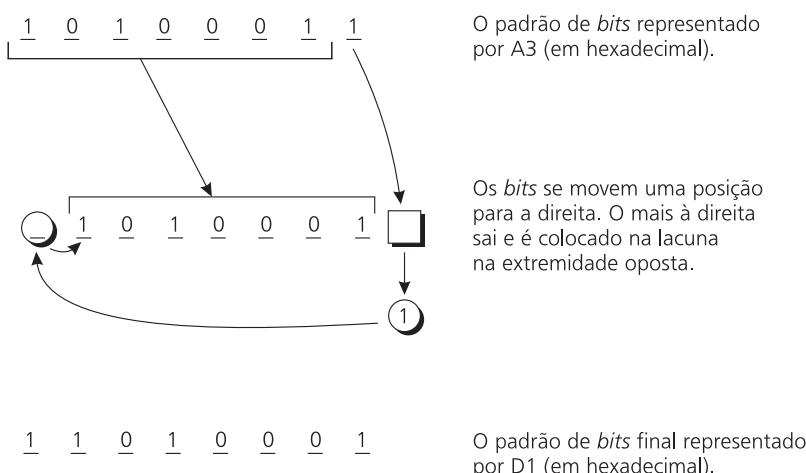


Figura 2.12 Rotação à direita de um *bit* no padrão de *bits* A3.

circular à direita designada pelo código de operação A. Neste caso, o primeiro dígito hexadecimal no operando especifica o registrador a ser rotacionado e o restante do operando, o número de bits a rotacionar. Assim, a instrução A501 significa “Rotacionar o registrador 5 à direita em 1 bit”. Se o registrador 5 originalmente continha o padrão de bits A3, passará a conter D1 após a execução da instrução (Figura 2.12). (Você pode constatar que outras instruções de deslocamento e rotação podem ser produzidas a partir de combinações das instruções contidas na linguagem de máquina do Apêndice C. Por exemplo, uma vez que um registrador tenha comprimento de oito bits, um deslocamento circular à direita de três bits produzirá o mesmo resultado que um deslocamento circular à esquerda de cinco bits.)

Operações aritméticas

Embora já tenhamos mencionado as operações aritméticas de adição, subtração, multiplicação e divisão, alguns detalhes ainda precisam ser esclarecidos. Primeiramente, como já foi dito no Capítulo 1, este conjunto de operações geralmente pode ser realizado a partir de uma única operação de adição e de um processo de negação. Por esta razão, alguns computadores pequenos são projetados apenas com a instrução de adição.

Também devemos lembrar que existem diversas variantes para cada operação aritmética. Já comentamos isso em relação às operações de adição disponíveis em nosso computador do Apêndice C. No caso da adição, por exemplo, se os números a somar forem representados na notação de complemento de dois, o processo de adição deverá ser executado como uma adição binária. Contudo, se os operandos forem representados como números em vírgula flutuante, o processo de adição irá extrair a mantissa de cada número, deslocá-la para a direita ou esquerda de acordo com o conteúdo do campo de expoente, conferir os bits de sinal, executar a adição e converter o resultado para a notação de vírgula flutuante. Assim, embora ambas sejam denominadas operações de adição, o comportamento do computador é totalmente diferente em cada caso, pois, para a máquina, as duas operações não guardam qualquer relação entre si.



QUESTÕES/EXERCÍCIOS

1. Execute as operações indicadas:

a. $\begin{array}{r} 01001011 \\ \text{AND } 10101011 \end{array}$

b. $\begin{array}{r} 10000011 \\ \text{AND } 11101100 \end{array}$

c. $\begin{array}{r} 11111111 \\ \text{AND } 00101101 \end{array}$

d. $\begin{array}{r} 01001011 \\ \text{OR } 10101011 \end{array}$

e. $\begin{array}{r} 10000011 \\ \text{OR } 11101100 \end{array}$

f. $\begin{array}{r} 11111111 \\ \text{OR } 00101101 \end{array}$

g. $\begin{array}{r} 01001011 \\ \text{XOR } 10101011 \end{array}$

h. $\begin{array}{r} 10000011 \\ \text{XOR } 11101100 \end{array}$

i. $\begin{array}{r} 11111111 \\ \text{XOR } 00101101 \end{array}$

2. Suponha que você deseja isolar os três bits do centro de uma cadeia de sete sem alterá-los e preencher os outros quatro bits com Os. Qual máscara você deverá usar, em conjunto com qual operação?
3. Suponha que você deseja obter o complemento dos três bits centrais de uma cadeia de sete, mantendo inalterados os outros quatro. Qual máscara deverá ser usada, em conjunto com qual operação?
4. a. Suponha que o operador XOR seja inicialmente aplicado entre os dois primeiros bits de uma cadeia de bits e que o operador XOR seja aplicado sucessivamente para os demais bits entre cada resultado intermediário obtido e o próximo bit da cadeia. Qual a relação entre o resultado obtido e o número de 1s existente na cadeia?

- b. Como este problema se relaciona com o da determinação do *bit* apropriado de paridade, nos casos de codificação de mensagens?
5. Em geral, é mais conveniente usar uma operação lógica do que uma operação aritmética equivalente. Por exemplo, a operação lógica AND combina dois *bits*, de forma muito similar à multiplicação aritmética. Qual a operação lógica cujo resultado se assemelha ao de somar dois *bits* e qual a discrepância que se observa neste caso?
6. Qual operação lógica, em conjunto com qual máscara, poderá ser usada para alterar códigos ASCII, convertendo letras minúsculas em maiúsculas? E para efetuar a operação inversa?
7. Qual o resultado obtido ao executar um deslocamento circular de três *bits* para a direita sobre as seguintes cadeias de *bits*?
a. 01101010 b. 00001111 c. 01111111
8. Qual o resultado obtido ao executar um deslocamento circular de um *bit* para a esquerda sobre os seguintes *bytes*, representados em notação hexadecimal? Dê sua resposta em formato hexadecimal.
a. AB b. 5C c. B7 d. 35
9. Um deslocamento circular de três *bits* para a direita sobre uma cadeia de oito *bits* é equivalente a um deslocamento circular de quantos *bits* para a esquerda?
10. Qual o padrão de *bits* que representa a soma de 01101010 e 11001100 se os padrões representam valores denotados em complemento de dois? E se os padrões representarem valores na notação de vírgula flutuante estudada no Capítulo 1?
11. Utilizando a linguagem de máquina do Apêndice C, escreva um programa que coloque um 1 no *bit* mais significativo da posição de memória de endereço A7, sem alterar os demais *bits*.
12. Utilizando a linguagem de máquina do Apêndice C, escreva um programa que copie os quatro *bits* centrais da posição de memória ED nos quatro *bits* menos significativos da posição de memória E1, completando com Os os quatro *bits* mais significativos desta mesma posição.

2.5 Comunicação com outros dispositivos

A memória principal e a UCP formam o núcleo de um computador. Nesta seção, investigaremos como esse núcleo, ao qual nos referiremos como computador, se comunica com os dispositivos periféricos, como os sistemas de armazenamento em disco, as impressoras, os teclados, ou *mouses*, os monitores e mesmo outros computadores.

Comunicação via controladores

Em geral, a comunicação entre o computador e outros dispositivos é feita através de um dispositivo intermediário, conhecido como **controlador**. No caso de um computador pessoal, o controlador corresponde fisicamente a uma placa de circuitos, que é encaixada na placa onde está o circuito principal do computador (placa-mãe). A seguir, o controlador é conectado por cabos aos dispositivos dentro do computador ou talvez a um conector na parte posterior, onde os dispositivos externos são ligados. Esses controladores geralmente são eles próprios pequenos computadores, cada um com os seus circuitos de memória e processador, que executam um programa para dirigir as atividades do controlador.

O controlador converte as mensagens e os dados de um lado para outro, em formatos respectivamente compatíveis com as características internas do computador e do(s) dispositivo(s) periférico(s) ligado(s) ao controlador. Assim, cada controlador é projetado para um tipo específico de dispositivo.

Isso explica por que às vezes um novo controlador deve ser adquirido quando se compra um novo dispositivo periférico. Um controlador usado com um disco antigo pode não ser compatível com um novo.

Quando um controlador é afixado à placa-mãe do computador, ele fica eletronicamente ligado à mesma via que conecta o processador com a memória principal (Figura 2.13). Nesta posição, cada controlador se habilita a monitorar os sinais provenientes da UCP e da memória principal, bem como a injetar seus próprios sinais na via.

Mais especificamente, a UCP pode se comunicar com o controlador conectado à via da mesma maneira como ela se comunica com a memória principal. Para enviar um padrão de bits ao controlador, ele deve ser inicialmente construído e colocado em um registrador de propósito geral da UCP. Então, uma instrução similar a STORE é executada para “armazenar” o padrão de bits no controlador. Da mesma forma, para receber um padrão de bits armazenado no controlador, uma instrução similar a LOAD é usada. Em alguns projetos de computador, códigos adicionais de operação são incorporados à linguagem de máquina para estas operações. As instruções com esses códigos de operação são chamadas instruções de E/S. Elas identificam os diversos controladores por um sistema de endereçamento similar ao usado na memória principal. Cada controlador recebe um conjunto único de endereços (endereços de E/S) que é usado nas instruções de E/S para identificá-lo. O conjunto de endereços atribuído a um controlador é chamado **porta** porque representa um “local” por onde a informação entra e sai do computador. Uma vez que o aspecto desses endereços de E/S pode ser idêntico ao dos endereços da memória principal, as vias dos computadores desse tipo de projeto contêm um sinal que indica se a mensagem transmitida na via é para a memória principal ou para um controlador. Assim, em resposta a uma instrução de E/S para enviar o conteúdo de um registrador para um controlador específico, a UCP reage de uma maneira similar a uma instrução para enviar um padrão de bits para uma posição de memória, exceto em que ela indica na via que o padrão de bits vai para o controlador e não para a memória.

Uma alternativa para estender a linguagem de máquina incluindo códigos especiais de operação para as instruções de E/S é usar os mesmos códigos de operação LOAD e STORE já disponíveis para a comunicação com a memória principal. Nesse caso, cada controlador é projetado para responder a referências a um único conjunto de endereços (novamente chamado porta), enquanto a memória principal é projetada para ignorar as referências a essas localizações. Assim, quando a UCP envia uma mensagem na via para armazenar um padrão de bits em uma posição de memória atribuída a um controlador, ele é recebido pelo controlador, e não pela memória. De maneira semelhante, se a UCP tentar ler dados destas posições de memória com uma instrução de carga (LOAD), receberá um padrão de bits do controlador, e não da memória. Esse sistema de comunicação é cha-

Projeto de via

O projeto das vias de um computador é uma matéria delicada. Por exemplo, os fios de uma via mal projetada podem funcionar como pequenas antenas – captando sinais transmitidos (de rádio, televisão, etc) e interrompendo a comunicação entre a UCP, a memória e dispositivos periféricos. Além disso, o comprimento da via (em torno de 15 cm) é significativamente maior do que os “fios” dentro do processador (medidos em microns). Assim, o tempo necessário para os sinais viajarem ao longo da via é muito maior do que o de transferência de sinais dentro da UCP. O resultado é que a tecnologia de vias está em constante competição para poder acompanhar a tecnologia das UCPs. Os computadores pessoais atuais refletem a variedade de projetos de vias, que diferem em características como a quantidade de dados transferidos simultaneamente, a taxa em que os sinais na via podem ser mudados e as propriedades físicas das conexões entre a via e a placa do controlador.

Figura 2.13 Controladores conectados a uma via.

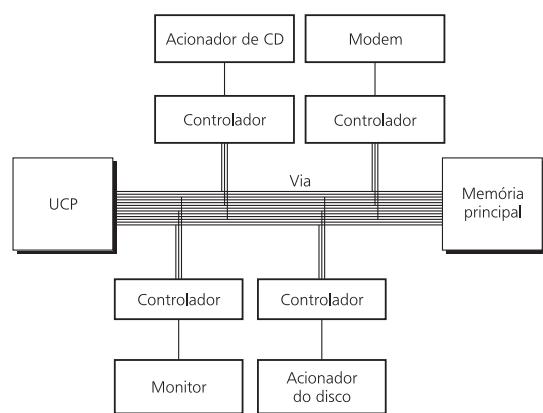


Figura 2.13 Controladores conectados a uma via.

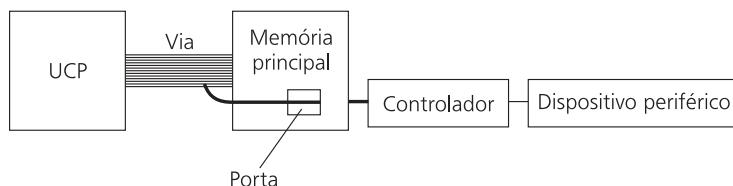


Figura 2.14 Uma representação conceitual da entrada/saída mapeada na memória (*memory mapped I/O*).

te os nanossegundos em que a UCP não estiver utilizando a via. Essa capacidade do controlador de ter acesso à memória principal é conhecida como **acesso direto à memória** (*direct memory access — DMA*), e é um recurso significativo para o desempenho do computador. Por exemplo, para recuperar dados do setor de um disco, a UCP pode enviar as requisições codificadas em padrões de bits ao controlador ligado ao disco, pedindo-lhe que leia o setor e coloque os dados em um bloco específico de células da memória. A UCP pode então continuar com outras tarefas enquanto o controlador realiza a operação de leitura e deposita os dados na memória via DMA. Assim, duas atividades estão em andamento ao mesmo tempo: a UCP está executando um programa enquanto o controlador supervisiona a transferência de dados entre o disco e a memória principal. Desta maneira, os recursos computacionais do processador não são desperdiçados durante a transferência de dados relativamente lenta.

O uso do DMA também tem um efeito prejudicial de aumentar a quantidade de comunicação que trafega pela via do computador. Os padrões de bits devem mover-se entre a UCP e a memória principal, entre a UCP e cada controlador e entre cada controlador e a memória principal. A coordenação de toda essa atividade na via é uma questão fundamental de projeto. Mesmo em um excelente projeto, a via central pode tornar-se um impedimento, uma vez que a UCP e os controladores competem entre si visando o acesso à via. Esse impedimento é conhecido como **gargalo de Von Neumann**, pois é uma consequência da **arquitetura de Von Neumann** subjacente, na qual a UCP busca as suas instruções através da via central.

Finalmente, devemos notar que a transferência de dados entre dois componentes de um computador raramente é feita em um só sentido. Ainda que possamos pensar na impressora como um dispositivo que apenas recebe dados, a verdade é que ela também envia dados de retorno ao computador. De fato, um computador produz e envia dados a uma impressora muito mais rapidamente do que a sua velocidade de impressão. Se um computador enviar os dados a uma impressora de maneira cega, ela poderá ficar saturada, e isso resultaria na perda de dados. Assim, um processo como o de impressão de um documento envolve um diálogo constante nos dois sentidos, no qual o computador e o dispositivo periférico trocam informações sobre o estado do dispositivo.

Esse diálogo freqüentemente envolve uma **palavra de estado** (*status word*), um padrão de bits gerado pelo dispositivo periférico e enviado ao controlador. Os bits na palavra de estado refletem as condições do dispositivo. Por exemplo, no caso de uma impressora, o valor do bit menos significativo da palavra de estado pode indicar que ela está sem papel, enquanto o próximo bit, se ela está pronta para receber mais dados. Dependendo do sistema, o controlador responde a uma informação de estado por si próprio, ou a disponibiliza para a UCP. Em cada caso, o programa do controlador ou o que está sendo executado pela UCP pode ser projetado para sustar o envio de dados para a impressora até que a informação de estado apropriada seja recebida.

Taxas de comunicação de dados

A taxa em que os bits são transferidos de um componente a outro dentro de um computador é medida em **bits por segundo** (bps). Unidades comuns incluem kbps (kilo-bps, igual a 1.000 bps), Mbps (mega-bps, igual a 1 milhão de bps) e Gbps (giga-bps, igual a 1 bilhão de bps). A taxa de

mado E/S mapeada na memória (*memory mapped I/O*) porque os dispositivos de entrada/saída do computador aparecem estar em várias posições da memória (Figura 2.14).

Uma vez que o controlador esteja conectado à via do computador, ele poderá realizar as suas próprias comunicações com a memória principal durante

transmissão disponível em um caso particular depende do tipo de caminho para a comunicação e da tecnologia usada em sua implementação. Essa taxa máxima freqüentemente é igual à **largura de banda** (*bandwidth*) do caminho de comunicação, isto é, dizer que um caminho de comunicação possui uma alta largura de banda significa que ele é capaz de transferir *bits* a uma taxa alta. Infelizmente, este é um uso impreciso do termo, uma vez que “taxas altas de transferência” podem significar taxas diferentes em diferentes situações.

Existem dois tipos básicos de caminhos de comunicação: paralelo e serial. Esses termos se referem à maneira como os padrões de *bits* são transferidos. No caso da **comunicação paralela**, vários *bits* são transferidos ao mesmo tempo, cada um em uma linha separada. Essa técnica permite a rápida transferência de dados, mas exige um caminho de comunicação relativamente complexo. Exemplos incluem a via interna do computador e a maioria das comunicações entre este e seus dispositivos periféricos, como os sistemas de armazenamento em massa e as impressoras. Nesse caso, as taxas medidas em Mbps e unidades maiores são comuns.

A **comunicação serial**, por sua vez, se baseia na transferência de um único *bit* por vez. Esta técnica tende a ser mais lenta, porém requer um caminho de comunicação mais simples, porque todos os *bits* são transferidos em uma só linha, um após o outro. Geralmente é utilizada para a comunicação entre computadores, em que os caminhos de comunicação mais simples são mais econômicos.

Por exemplo, as linhas telefônicas atuais são inherentemente sistemas de comunicação serial, uma vez que transmitem tons um após o outro. A comunicação entre computadores nessas linhas é realizada primeiramente pela conversão dos padrões de *bits* em tons audíveis por meio de um **modem** (abreviatura de modulador / demodulador), transferência dos tons serialmente pelo sistema telefônico e finalmente a reconversão dos mesmos para *bits* por meio de outro modem instalado no destino.

Na realidade, a simples representação de padrões de *bits* por tons de diferentes freqüências (conhecida como chave por deslocamento de freqüência) é usada apenas nas comunicações em baixa velocidade, a não mais que 1200 bps. Para alcançar taxas de transferência de 2.400 bps, 9.600 bps e maiores, os modems combinam as mudanças de freqüência dos tons com as de amplitude (volume) e de fase (grau de atraso no qual a transmissão é realizada). Para alcançar taxas ainda maiores, as técnicas de compressão de dados freqüentemente são aplicadas, produzindo taxas aparentes de transferência de até 57,6 Kbps.

Essas taxas de transferência aparentam ser o limite que pode ser atingido na faixa de freqüência utilizada pelas linhas telefônicas tradicionais (de até 3 Hz), e são insuficientes para as necessidades atuais de comunicação. A transferência de gráficos a 57,6 Kbps pode se tornar exasperante, e o vídeo está à frente da razão. Assim, novas tecnologias vêm sendo desenvolvidas para proporcionar taxas de transferência maiores para os clientes que até então usavam as linhas telefônicas tradicionais. Uma dessas técnicas é conhecida como DSL (*Digital Subscriber Line*) que leva em conta o fato de que as linhas telefônicas atuais são capazes de trabalhar com freqüências maiores do que as utilizadas para a comunicação de voz. Esses sistemas normalmente obtêm taxas de transferência em torno de 1,5 Mbps, mas podem atingir até 6 Mbps em uma direção se as funcionalidades na outra direção forem restrinpidas. O desempenho exato depende da versão da DSL utilizada e do comprimento da linha até a central telefônica, que geralmente é limitado a não mais que 6 Km (3,5 milhas). Outras tecnologias que competem com a DSL incluem a tecnologia a cabo, como a utilizada nos sistemas de televisão a cabo, que pode atingir taxas na ordem de 40 Mbps, e as fibras ópticas, cujas taxas se situam na faixa entre Mbps e Gbps.



QUESTÕES/EXERCÍCIOS

1. Pressuponha que a máquina descrita no Apêndice C usa E/S mapeada na memória e que o endereço B5 é a posição da porta da impressora para a qual os dados a serem impressos devem ser enviados.
 - a. Se o registrador 7 contém a código ASCII da letra A, que instrução em linguagem de máquina deve ser usada para efetivar a impressão dessa letra?

- b. Se a máquina executa um milhão de instruções por segundo, quantas vezes esse caractere pode ser enviado à impressora em um segundo?
 - c. Se a impressora é capaz de imprimir cinco páginas convencionais de texto em um minuto, ela será capaz de acompanhar a velocidade em que os caracteres são enviados em (b)?
2. Suponha que o disco rígido de seu computador pessoal gire a uma velocidade de 3.000 rotações por minuto, cada trilha contenha 16 setores e cada setor, 1.024 bytes. Aproximadamente que taxa de transferência é necessária entre o acionador do disco e o controlador para que este último receba os bits do disco assim que forem lidos?
 3. Quanto tempo leva para se transferir um romance de 300 páginas codificado em ASCII a uma taxa de transferência de 57.600 bps?

2.6 Outras arquiteturas

Para ampliar nossa perspectiva, consideremos algumas alternativas à arquitetura de máquina estudada nas seções precedentes.

Canalização (*pipelining*)

Os sinais elétricos percorrem um fio condutor com uma velocidade não superior à da luz. Como a luz viaja a aproximadamente 1 pé por nanosegundo*, são necessários, no mínimo, 2 nanosegundos para a unidade de controle do processador buscar uma instrução de uma posição de memória que esteja a 1 pé de distância. (O pedido de leitura deve ser enviado para a memória, o que requer pelo menos 1 nanosegundo, e a instrução deve ser enviada de volta à unidade de controle, o que requer outro nanosegundo, no mínimo.) Como consequência, buscar e executar uma instrução nesta máquina exige vários nanosegundos, o que significa que aumentar a velocidade de execução de um computador corresponde, em última instância, a um problema de miniaturização. Embora muitos avanços fantásticos tenham sido realizados nesta área, isto continua a ser um fator limitante.

Em um esforço para resolver tal dilema, os engenheiros de computação se concentraram no conceito de vazão (*throughput*), em vez da mera velocidade de execução. A **vazão** se refere à quantidade total de trabalho realizado pelo computador em um dado período de tempo, e não ao tempo que ele leva para realizar uma dada tarefa.

Um exemplo de como a vazão de um computador pode ser aumentada sem exigir aumento na velocidade de execução envolve a **canalização** (*pipelining*)**, que é a técnica de permitir que os passos de um ciclo da máquina se sobreponham. Especificamente, enquanto uma instrução está sendo executada, a próxima instrução pode ser buscada, o que significa que mais de uma instrução pode estar no “cano” a qualquer momento, cada uma em um estágio diferente de seu processamento. Portanto, a vazão total da máquina é aumentada, muito embora o tempo necessário para buscar e executar cada instrução individual permaneça o mesmo. (Obviamente, quando uma instrução de desvio é encontrada, qualquer ganho que teria sido obtido com a busca prévia não se efetiva, pois as instruções do “cano” não são as desejadas.)

Os projetos das máquinas modernas empurram o conceito de canalização para além do nosso simples exemplo. Elas são capazes de buscar várias instruções ao mesmo tempo e de fato executar mais de uma instrução simultaneamente desde que elas não dependam uma da outra.

*N. de T. No sistema métrico, 1 pé = 0,3048m; 1 nanosegundo = 10^{-9} segundo = um bilionésimo de segundo. A velocidade da luz é cerca de 3,10⁸m/s.

**N. de T. *Pipeline* é um conceito similar ao de uma *linha de produção*, capaz de aumentar a vazão de uma fábrica sem alterar a velocidade do trabalho dos operários, por meio da especialização dos mesmos e do trabalho em equipe, explorando o paralelismo.

Máquinas com multiprocessamento

A canalização pode ser vista como um primeiro passo para o **processamento paralelo**, que é a realização de várias atividades ao mesmo tempo. Contudo, o processamento verdadeiramente paralelo exige mais de uma unidade de processamento, o que resulta em computadores conhecidos como máquinas com multiprocessamento.

Atualmente, diversas máquinas têm sido projetadas dentro deste princípio. Uma forma de atingir tal meta consiste em conectar a uma mesma memória principal vários processadores, todos similares ao processador central existente em um computador convencional. Nesta configuração, os processadores podem funcionar de modo independente, embora coordenando seus esforços por meio de mensagens enviadas de uns para os outros através das suas áreas comuns de memória. Por exemplo, quando um processador recebe a incumbência de executar uma tarefa de grande porte, ele pode armazenar um programa que execute parte desta tarefa na área comum de memória para, em seguida, solicitar que algum outro processador o execute. O resultado é uma máquina em que diferentes seqüências de instruções são executadas com diferentes conjuntos de dados. Máquinas deste tipo são denominadas arquiteturas **MIMD** (*multiple-instruction stream, multiple-data stream* — múltiplos fluxos de instruções, múltiplos fluxos de dados), em vez da tradicional arquitetura **SISD** (*single-instruction stream, single-data stream* — único fluxo de instruções, único fluxo de dados).

Uma outra versão da arquitetura baseada em múltiplos processadores é obtida pela reunião dos processadores, de forma que executem a mesma seqüência de instruções em uníssono, cada qual sobre o seu próprio conjunto de dados. O resultado é um caso de arquitetura **SIMD** (*single-instruction stream, multiple-data stream* — único fluxo de instruções, múltiplos fluxos de dados). Tais máquinas são úteis em aplicações nas quais uma mesma tarefa deva ser aplicada a diversos conjuntos de itens semelhantes, dentro de um grande bloco de dados.

Outra abordagem ao processamento paralelo consiste em construir grandes computadores na forma de um conglomerado de computadores menores, cada qual com sua própria memória e UCP. Neste tipo de arquitetura, cada um dos computadores menores é ligado a seus vizinhos de forma que as tarefas solicitadas ao sistema como um todo possam ser distribuídas entre os diversos computadores individuais. Por conseguinte, se uma tarefa designada a um dos computadores internos puder ser dividida em subtarefas independentes, esta máquina poderá, por sua vez, solicitar que outras máquinas as executem ao mesmo tempo. Consequentemente, a tarefa original seria completada em um tempo muito menor do que se fosse executada por uma máquina convencional, com um único processador.

No Capítulo 10, estudaremos outra arquitetura de multiprocessamento, as redes neurais artificiais, cujo projeto se baseia nas teorias de funcionamento do cérebro humano. Tais máquinas consistem em muitos processadores elementares cujas saídas apenas representam reações à combinação de seus dados de entrada. Estes processadores simples são interligados, formando uma rede em que os dados de saída de alguns processadores são utilizados como dados de entrada para os demais. Esta máquina se programa ajustando-se o grau com que cada saída de um processador influí na reação dos outros processadores aos quais se conecta. Isto simula a teoria segundo a qual as redes neurais biológicas aprendem a produzir uma reação particular para um determinado estímulo, ajustando a composição química das conexões (sinapses) entre neurônios, as quais, em troca, ajustam a capacidade que o neurônio apresenta de influir no comportamento dos demais.

Os proponentes das redes neurais artificiais argumentam que embora a tecnologia esteja se habilitando a construir circuitos eletrônicos que contenham aproximadamente o mesmo número de unidades de chaveamento do que o de neurônios no cérebro humano (acredita-se que os neurônios sejam as unidades de chaveamento da natureza), a capacidade das máquinas atuais ainda está muito aquém da do cérebro humano. Isto, argumentam eles, é resultado do uso ineficiente que se faz dos componentes tradicionais dos computadores submetidos à arquitetura de Von Neumann. Afinal de contas, se uma máquina é construída com um grande circuito de memória que suporte alguns processadores, então a maioria de seus circuitos está fadada a ficar ociosa a maior parte do tempo. Em contraste, a maior parte da mente humana pode estar ativa a cada momento.

Assim, o projeto dos computadores atuais está expandindo o modelo básico UCP-memória principal e em alguns casos abandonando-o completamente, a fim de desenvolver máquinas mais úteis.



QUESTÕES/EXERCÍCIOS

1. Com relação à questão 3 da Seção 2.3, se a máquina usar a técnica de canalização discutida no texto, qual será o conteúdo do “cano” quando a instrução de endereço AA for executada? Sob que condições a técnica de canalização não seria benéfica nesse ponto do programa?
2. Quais conflitos devem ser solucionados ao executar o programa da questão 4 da Seção 2.3 em uma máquina com canalização?
3. Suponha dois processadores “centrais”, ligados à mesma memória e que executem programas diferentes. Suponha também que um destes processadores necessite somar uma unidade ao conteúdo de uma célula de memória, ao mesmo tempo em que um outro processador necessite subtrair uma unidade do conteúdo desta mesma célula (como resultado, essa célula de memória deveria permanecer intacta).
 - a. Descreva uma seqüência de ações que produza como resultado, na posição de memória em questão, uma unidade a menos que o valor inicial.
 - b. Descreva uma seqüência de ações cujo resultado, na posição de memória em questão, seja uma unidade a mais que o valor inicial.

Problemas de revisão de capítulo

(Os problemas marcados com asteriscos se referem às seções opcionais.)

1. Dê uma breve definição de cada item abaixo:
 - a. Registrador
 - b. Memória cache
 - c. Memória principal
 - d. Armazenamento em massa
2. Seja um bloco de dados armazenado nas posições de memória do computador descrito no Apêndice C nos endereços de B9 a C1, inclusive. Quantas células de memória estão neste bloco? Liste os seus endereços.
3. Qual o valor do contador de instruções do computador descrito no Apêndice C logo após a execução da instrução B0BA?
4. Suponha que as células de memória dos endereços de 00 a 05 do computador descrito no Apêndice C contenham os seguintes padrões hexadecimais de bits:

Endereço	Conteúdo
00	21
01	04
02	31
03	00
04	C0
05	00

Pressupondo que inicialmente o contador de instruções contenha 00, registre o conteúdo do contador, do registrador de instruções e da posição de memória de endereço 00 ao término de cada fase de busca do ciclo de máquina, até a sua parada.

5. Suponha que três valores (x , y e z) estejam armazenados na memória de um computador. Descreva a seqüência de eventos (carregar registradores com dados da memória, armazenar valores na memória e assim por diante) para o cálculo das sentenças matemáticas $x + y + z$. Qual é o resultado ($2x$) + y ?
6. Seguem abaixo algumas instruções, escritas na linguagem de máquina descrita no Apêndice C. Traduza-as para o português.
 - a. 407E
 - b. 9028
 - c. A302
 - d. B3AD
 - e. 2835
7. Seja uma linguagem de máquina projetada com códigos de operação de quatro bits. Quantos tipos de instrução diferentes esta linguagem pode conter? O que aconteceria se o tamanho do código de operação fosse aumentado para oito bits?

- 8.** Traduza as seguintes instruções, descritas em português, para a linguagem de máquina descrita no Apêndice C.
- Carregar o registrador 8 com o conteúdo da posição de memória 55.
 - Carregar o registrador 8 com o valor hexadecimal 55.
 - Deslocar ciclicamente três bits à direita o registrador 4.
 - Aplicar o operador lógico AND sobre o conteúdo dos registradores F e 2 e guardar o resultado no registrador 0.
 - Desviar para a instrução de memória de endereço 31 se o conteúdo do registrador 0 for igual ao do registrador B.

- 9.** Classifique as instruções abaixo (na linguagem de máquina do Apêndice C) sob os seguintes aspectos: a execução da instrução modifica o conteúdo da posição de memória do endereço 3B, utiliza o conteúdo desta posição de memória, ou é independente deste conteúdo?
- 153B
 - 253B
 - 353B
 - 3B3B
 - 403B

- 10.** Suponha que as células de memória dos endereços de 00 a 03 do computador descrito no Apêndice C contenham os seguintes padrões hexadecimais de bits:

Endereço	Conteúdo
00	23
01	02
02	C0
03	00

- Traduza a primeira instrução para o português.
- Se o computador iniciar suas atividades com o valor 00 no seu contador de instruções, qual será o padrão de bits existente no registrador 3 quando o computador parar?

- 11.** Suponha que as células de memória dos endereços de 00 a 05 do computador descrito no Apêndice C contenham os seguintes padrões hexadecimais de bits:

Endereço	Conteúdo
00	10
01	04
02	30
03	45
04	C0
05	00

Responda às seguintes questões, pressupondo que 00 seja o conteúdo inicial do contador de instruções:

- Traduza para o português as instruções executadas.
- Qual será o padrão de bits existente na posição de memória do endereço 45 quando o computador parar?
- Qual será o padrão de bits existente no contador de instruções quando o computador parar?

- 12.** Suponha que as células de memória dos endereços de 00 a 09 do computador descrito no Apêndice C contenham os seguintes padrões hexadecimais de bits:

Endereço	Conteúdo
00	1A
01	02
02	2B
03	02
04	9C
05	AB
06	3C
07	00
08	C0
09	00

Pressupondo que a máquina inicie com seu contador de instruções igual a 00:

- O que a célula de memória de endereço 00 conterá quando a máquina parar?
- Qual será o padrão de bits no contador de instruções quando a máquina parar?

- 13.** Suponha que as células de memória dos endereços de 00 a 0D do computador descrito no Apêndice C contenham os seguintes padrões hexadecimais de bits:

Endereço	Conteúdo
00	20
01	03
02	21
03	01
04	40
05	12
06	51
07	12
08	B1
09	0C
0A	B0

0B	06
0C	C0
0D	00

Supondo que a máquina inicie com o seu contador de instruções igual a 00:

- Qual será o padrão de bits existente no registrador 1 quando o computador parar?
- Qual será o padrão de bits existente no registrador 0 quando o computador parar?
- Qual será o padrão de bits existente no contador de instruções quando o computador parar?

- 14.** Suponha que as células de memória dos endereços de F0 a FD do computador descrito no Apêndice C contenham os seguintes padrões hexadecimais de bits:

Endereço	Conteúdo
F0	20
F1	00
F2	21
F3	01
F4	23
F5	05
F6	B3
F7	FC
F8	50
F9	01
FA	B0
FB	F6
FC	C0
FD	00

Se iniciarmos a máquina com o seu contador de instruções igual a F0, qual será o conteúdo do registrador 0 quando a máquina finalmente executar a instrução de parada na posição FC?

- 15.** Se o computador do Apêndice C executar uma instrução a cada microsegundo, quanto tempo ele levará para completar o programa do Problema 14?
- 16.** Suponha que as células de memória dos endereços de 20 a 28 do computador descrito no Apêndice C contenham os seguintes padrões hexadecimais de bits:

Endereço	Conteúdo
20	12
21	20
22	32

23	30
24	B0
25	21
26	20
27	C0
28	00

Pressupondo que a máquina inicie com o seu contador de instruções igual a 20:

- Quais serão os padrões de bits existentes nos registradores 0, 1 e 2 quando o computador parar?
- Qual será o padrão de bits existente na célula de memória de endereço 30 quando o computador parar?
- Qual será o padrão de bits existente na célula de memória de endereço B0 quando o computador parar?

- 17.** Suponha que as células de memória dos endereços de AF a B1 do computador descrito no Apêndice C contenham os seguintes padrões hexadecimais de bits:

Endereço	Conteúdo
AF	B0
B0	B0
B1	AF

O que acontecerá se iniciarmos a máquina com o seu contador de instruções igual a AF?

- 18.** Suponha que as posições de memória dos endereços de 00 a 05 do computador descrito no Apêndice C contenham os seguintes padrões hexadecimais de bits:

Endereço	Conteúdo
00	25
01	B0
02	35
03	04
04	C0
05	00

Se iniciarmos o computador com seu contador de instruções igual a 00, quando o computador irá parar?

- 19.** Para cada caso a seguir, escreva um pequeno programa na linguagem de máquina descrita no Apêndice C para executar as ações solicitadas. Trabalhe com a hipótese de que cada programa estará na memória a partir do endereço 00.

- a. Mover o valor contido na posição de memória 8D para a posição de memória B3.
- b. Trocar entre si os valores armazenados nas posições de memória 8D e B3.
- c. Se o valor armazenado nas posições de memória 45 for 00, então colocar o valor CC na posição de memória 88; caso contrário, colocar o valor DD na posição de memória 88.
- 20.** Um jogo popular entre os usuários de computador é o *core wars*, uma variação do jogo batalha naval. (O termo *core* tem sua origem nos primórdios da tecnologia de memórias, quando os 0s e os 1s eram representados por campos magnéticos em pequenos anéis feitos de material magnético.) O jogo é disputado entre dois programas adversários, cada qual armazenado em diferentes posições da mesma memória do computador. Admite-se que o computador alterne o processamento dos dois programas, executando ora uma instrução de um, ora do outro. O objetivo de cada programa é destruir o outro, gravando dados estranhos sobre o programa do adversário, sem que, entretanto, os programas conheçam a posição um do outro.
- a. Escreva um programa na linguagem de máquina do Apêndice C, que aborde o jogo em uma maneira defensiva e que seja o menor possível.
- b. Escreva um programa, na linguagem de máquina do Apêndice C, que tente evitar qualquer ataque do programa adversário movendo-se para outras posições diferentes. Mais precisamente, instrua o seu programa para começar na posição 00, fazer uma cópia de si mesmo na posição 70 e então desviar para esta nova cópia.
- c. Estenda o programa do item (b) de modo que ele possa continuar se mudando para novas posições de memória. Em particular, faça o seu programa se mover para a posição 70, depois para E0 ($=70 + 70$) e, em seguida, para 60 ($=70 + 70 + 70$) etc.
- 21.** Escreva um programa na linguagem de máquina do Apêndice C para computar a soma dos valores armazenados nas posições de memória A1, A2, A3 e A4 e codificados em complemento de dois. O programa deverá armazenar o total na posição de memória A5.
- 22.** Suponha que as posições de memória dos endereços de 00 a 05 do computador descrito no Apêndice C contenham os seguintes padrões hexadecimais de bits:
- | Endereço | Conteúdo |
|----------|----------|
| 00 | 20 |
| 01 | C0 |
| 02 | 30 |
| 03 | 04 |
| 04 | 00 |
| 05 | 00 |
- O que acontecerá se iniciarmos o computador com o 00 no seu contador de instruções?
- 23.** O que acontecerá se as posições de memória dos endereços 06 e 07 do computador descrito no Apêndice C contiverem os padrões de bits B0 e 06, respectivamente, e se o computador for iniciado com o valor 06 no seu contador de instruções?
- 24.** Suponha que o seguinte programa, escrito na linguagem de máquina do Apêndice C, seja armazenado na memória principal a partir do endereço 30 (hexadecimal). Que tarefa o programa executará quando for processado?
- | |
|------|
| 2003 |
| 2101 |
| 2200 |
| 2310 |
| 1400 |
| 3410 |
| 5221 |
| 5331 |
| 3239 |
| 333B |
| B248 |
| B038 |
| C000 |
- 25.** Resuma os passos envolvidos quando a máquina descrita no Apêndice C executa uma instrução com o código de operação B. Expresse a sua resposta com um conjunto de diretrizes, como se você estivesse dizendo à UCP o quê fazer.
- *26.** Resuma os passos envolvidos quando a máquina descrita no Apêndice C executa uma instrução com o código de operação 5. Expresse a sua resposta com um conjunto de diretrizes, como se você estivesse dizendo à UCP o quê fazer.

- *27.** Resuma os passos envolvidos quando a máquina descrita no Apêndice C executa uma instrução com o código de operação 6. Expressse a sua resposta com um conjunto de diretrizes, como se você estivesse dizendo à UCP o quê fazer.
- *28.** Suponha que os registradores 4 e 5 do computador descrito no Apêndice C contenham os padrões hexadecimais de bits 3C e C8, respectivamente. Qual será o padrão de *bits* contido no registrador 0 após a execução das seguintes instruções?
- 5045
 - 6045
 - 7045
 - 8045
 - 9045
- *29.** Utilizando a linguagem de máquina descrita no Apêndice C, escreva programas que executem as seguintes tarefas:
- Copiar para a posição de memória BB o padrão de *bits* armazenado na posição 66.
 - Alterar os quatro *bits* menos significativos da posição de memória de endereço 34 para 0s, mantendo inalterados os demais *bits*.
 - Copiar os quatro *bits* menos significativos da posição de memória de endereço A5 para os quatro *bits* menos significativos do endereço A6, mantendo inalterados os demais *bits* deste.
 - Copiar os quatro *bits* menos significativos da posição de memória de endereço A5 para os quatro *bits* mais significativos de A5. (Assim, os primeiros quatro *bits* em A5 ficarão iguais aos seus quatro últimos.)
- *30.** Efetue as operações indicadas:
- $\begin{array}{r} 111000 \\ \text{AND } 101001 \end{array}$
 - $\begin{array}{r} 000100 \\ \text{AND } 010101 \end{array}$
 - $\begin{array}{r} 111011 \\ \text{AND } 110101 \end{array}$
 - $\begin{array}{r} 000100 \\ \text{OR } 101001 \end{array}$
 - $\begin{array}{r} 111011 \\ \text{OR } 110101 \end{array}$
 - $\begin{array}{r} 000100 \\ \text{XOR } 101001 \end{array}$
 - $\begin{array}{r} 111011 \\ \text{XOR } 010101 \end{array}$
 - $\begin{array}{r} 000100 \\ \text{XOR } 101001 \end{array}$
 - $\begin{array}{r} 111011 \\ \text{XOR } 010101 \end{array}$
- *31.** Identifique a máscara e a operação lógica necessárias à realização dos seguintes objetivos:
- Colocar 0s nos quatro *bits* centrais de um padrão de oito, mantendo inalterados os demais *bits*.
 - Obter o complemento de um padrão de oito *bits*.
 - Obter o complemento do *bit* mais significativo de um padrão de oito *bits*, sem alterar os demais.
 - Colocar um 1 no *bit* mais significativo de um padrão de oito, sem alterar os demais.
 - Colocar 1s em todos os *bits* de um padrão de oito *bits*, exceto no seu *bit* mais significativo, o qual deverá permanecer inalterado.
- *32.** Identifique uma operação lógica (bem como a máscara correspondente) que, quando executada sobre uma cadeia de entrada de oito *bits*, produzirá uma cadeia de saída formada apenas de 0s, se e somente se a cadeia de entrada for 10000001.
- *33.** Descreva uma sequência de operações lógicas (bem como as máscaras correspondentes) que, quando executadas sobre uma cadeia de entrada de oito *bits*, produzirá um byte de saída formado apenas de 0s somente se a cadeia de entrada for iniciada e terminada por 1s. Caso contrário, a saída deverá ter pelo menos um dos *bits* igual a 1.
- *34.** Indique qual será o resultado obtido ao executar um deslocamento circular de quatro *bits* à esquerda sobre os seguintes padrões de *bits*:
- 10101
 - 11110000
 - 001
 - 101000
 - 00001
- *35.** Qual será o resultado obtido ao executar um deslocamento circular de um *bit* à direita sobre os seguintes bytes, representados em notação hexadecimal (forneça suas respostas na notação hexadecimal):
- 3F
 - 0D
 - FF
 - 77
- *36.** Que instrução na linguagem de máquina descrita no Apêndice C poderia ser usada para fazer um deslocamento circular à direita de três *bits* no registrador B?
- *37.** Escreva um programa, na linguagem de máquina do Apêndice C, que reverta o conteúdo da posição de memória de endereço 8C.
- *38.** Uma impressora que imprime 40 caracteres em um segundo pode manter este ritmo de transferência para uma cadeia de caracteres ASCII (cada qual

com seu *bit* de paridade), chegando serialmente a uma velocidade de 300 bps? E se fosse a 1200 bps?

- *39. Suponha que uma pessoa digite 30 palavras em um minuto. (Considere uma palavra como cinco caracteres.) Se um computador executar uma instrução a cada microsegundo (milésimo de segundo), quantas instruções serão executadas durante o tempo decorrido entre a digitação de dois caracteres sucessivos?
- *40. Quantos *bits* por segundo um teclado deve transmitir para sustentar um ritmo de digitação de 30 palavras por minuto? (Suponha que cada caractere esteja codificado em ASCII, juntamente com o seu *bit* de paridade, e que cada palavra consista em cinco caracteres.)
- *41. Um sistema de comunicação é capaz de transmitir qualquer seqüência de oito estados diferentes a uma taxa de, no máximo, 300 estados por segundo. Com que velocidade, medida em *bits* por segundo, este sistema poderia ser usado para transferir informação?
- *42. Suponha que o computador descrito no Apêndice C se comunique com uma impressora utilizando a técnica de entrada/saída mapeada na memória. Suponha também que o endereço FF seja utilizado para enviar caracteres à impressora, e o endereço FE, para receber informações sobre o estado desta. Em particular, suponha que o *bit* menos significativo do endereço FE indique se a impressora está pronta ou não para receber mais um caractere (0 indica *não-pronto* e 1 indica *pronto*). Iniciando no endereço 00, escreva uma rotina, em linguagem de máquina, que aguarde até que a impressora esteja pronta para receber outro caractere, enviando então para a impressora o caractere representado pelo padrão de *bits* contido no registrador 5.

- *43. Escreva um programa, na linguagem de máquina descrita no Apêndice C, que preencha com 0s todas as posições de memória dos endereços de A0 a C0. Tal programa deverá ser suficientemente pequeno para caber nas posições de memória de endereços de 00 a 13 (hexadecimal).
- *44. Suponha que um computador com 20GB de espaço disponível para armazenamento em disco rígido receba dados por meio de uma conexão telefônica, a uma velocidade de transmissão de 14.400 bps. A esta velocidade, quanto tempo transcorrerá até que todo o espaço disponível seja preenchido?
- *45. Suponha que uma linha de comunicação esteja sendo utilizada para transmitir dados serialmente, a uma velocidade de 14.400 bps. Se houver uma interferência com 0,01 segundo de duração, quantos *bits* de dados serão afetados por ela?
- *46. Suponha que haja 32 processadores disponíveis, cada um capaz de calcular a soma de dois números, compostos de vários dígitos, em um milionésimo de segundo. Descreva como as técnicas de processamento simultâneo podem ser aplicadas para calcular a soma de 64 números em apenas seis milionésimos de segundo. Quanto tempo seria necessário para um único processador calcular esta mesma soma?
- *47. Faça um resumo das diferenças entre as arquiteturas CISC e RISC.
- *48. Identifique dois métodos para aumentar a vazão de processamento em um computador.
- *49. Descreva de que forma a média de um conjunto de números pode ser calculada mais rapidamente com uma máquina com multiprocessamento do que com outra com apenas um processador.

Questões sociais

As seguintes questões procuram auxiliar o leitor no entendimento de alguns assuntos éticos, sociais e legais no campo da computação. O objetivo não é meramente o de fornecer respostas a tais questões. O leitor também deve justificá-las e verificar se as justificativas apresentadas preservam sua consistência de uma questão para outra.

1. Suponha que um fabricante de computadores desenvolva uma nova arquitetura de máquina. Até que ponto deve ser permitido a esta companhia manter a propriedade de tal arquitetura? Qual seria a melhor política a adotar, tendo em vista o bem da sociedade?

2. Em um certo sentido, o ano de 1923 marca o nascimento da então chamada *obsolescência planejada*. Foi o ano em que a General Motors, liderada por Alfred Sloan, introduziu o conceito de modelos anuais. A idéia era aumentar as vendas mudando o estilo, em vez de necessariamente introduzir um automóvel melhor. Sloan é lembrado por dizer: “Queremos fazer você ficar insatisfeito com o seu carro atual, para que compre um novo”. Até que ponto essa estratégia de mercado é utilizada atualmente na indústria dos computadores?
3. Normalmente, nossos pensamentos giram em torno de como a tecnologia computacional mudou a sociedade. Todavia, muitos argumentam que tal tecnologia muitas vezes impediu que mudanças ocorressem, ao permitir que sistemas antigos permanecessem, e, até se fortalecessem, em alguns casos. Por exemplo, o domínio da bolsa de valores de Nova Iorque ou o papel do governo norte-americano na sociedade sobreviveriam sem a tecnologia computacional? Até que ponto seria possível manter uma autoridade centralizada sem o uso da tecnologia computacional? Até que ponto estaríamos melhor ou pior sem tal tecnologia?
4. É ético para uma pessoa assumir a atitude de desconhecimento quanto aos detalhes internos de uma máquina, considerando que um outro indivíduo a construirá, fará sua manutenção e resolverá qualquer problema que venha a surgir? Sua resposta depende do fato de a máquina ser um computador, um automóvel, uma usina nuclear ou uma torradeira?
5. Suponha que um fabricante produza uma pastilha (circuito eletrônico) e depois descubra a existência de uma falha de projeto. Suponha que, mais adiante, o fabricante decida não substituir tais pastilhas já colocadas no mercado e resolva manter em segredo a falha, argumentando que nenhuma delas está sendo utilizada em aplicações em que a falha possa causar consequências. Alguém poderia ser prejudicado por essa decisão do fabricante?
6. O avanço da tecnologia é uma cura para as doenças do coração ou é a fonte de vida sedentária que leva às doenças do coração?
7. Em “Walden”, Henry David Thoreau argumenta que nos tornamos ferramentas de nossas ferramentas, isto é, em vez de nos beneficiarmos com as ferramentas que possuímos, gastamos nosso tempo obtendo-as e mantendo-as. Até que ponto isto é verdade em relação à computação? Por exemplo, se você possui um computador pessoal, quanto tempo passou juntando dinheiro para poder adquiri-lo, aprendendo a usá-lo, mantendo-o, atualizando-o e se preocupando com ele em comparação ao tempo que você passou se beneficiando com ele? Quando você o utiliza, está usando bem o seu tempo? Você é mais ativo socialmente com ou sem o seu computador pessoal?
8. É fácil imaginar desastres financeiros ou em sistemas de navegação que podem ocorrer como resultado de erros aritméticos devidos a estouro ou problemas de truncamento. Que consequências podem resultar nos sistemas de armazenamento de imagem devido à perda de detalhes (talvez em áreas como reconhecimento ou diagnóstico médico)?

Leituras adicionais

- Carpinelli, J. D. *Computer Systems Organization and Architecture*. Reading, MA: Addison-Wesley, 2001.
- Hamacher, V. C., Z. G. Vranesic, and S. G. Zaky. *Computer Organization*, 5th ed. New York: McGraw-Hill, 2002.
- Knuth, D. E. *The Art of Computer Programming*, vol. 1, 3rd ed. Reading, MA: Addison-Wesley Longman, 1998.
- Messmer, H. *The Indispensable PC Hardware Book* 4th ed. Boston: Addison-Wesley, 2002.
- Patterson, D. A., and J. L. Hennessy. *Computer Organization and Design*. 2nd ed. San Francisco: Morgan Kaufmann, 1997.
- Tanenbaum, A. S. *Structured Computer Organization*. 4th ed. Upper Saddle River, NJ: Prentice Hall, 1999.

P A R T E 2

Software

Na primeira parte, discutimos os principais componentes de um computador. Desses, os que são materiais denominam-se *hardware*. Entretanto, os programas que o *hardware* executa são imateriais e denominados *software*. Nesta Parte 2 do livro, dirigiremos a nossa atenção para tópicos relacionados a *software*, o que nos conduzirá ao âmago da Ciência da Computação, o estudo de algoritmos. Em particular, investigaremos a descoberta, a representação e a comunicação de algoritmos.

Começaremos discutindo os sistemas operacionais no Capítulo 3. Tais sistemas são pacotes de *software* grandes e complexos que controlam as atividades globais de um computador ou de um grupo de computadores conectados em rede. Assim, nosso estudo de sistemas operacionais abrange o tópico de redes de computadores em geral e a Internet em particular. No Capítulo 4, estudaremos os algoritmos, com ênfase no seu descobrimento e em suas representações. No Capítulo 5, analisaremos como os algoritmos são comunicados às máquinas por meio do processo de programação e investigaremos as características das linguagens de programação populares. Finalmente, no Capítulo 6, estudaremos todo o processo de desenvolvimento de programas, no contexto da Engenharia de *Software*.

CAPÍTULO 3

Sistemas operacionais e redes

As aplicações atuais dos computadores freqüentemente exigem que uma única máquina execute atividades que possam competir pela posse dos recursos da máquina. Por exemplo, um computador pode ser conectado a vários terminais ou estações de trabalho, atendendo simultaneamente a diversos usuários. Até mesmo em sistemas de um único usuário, como os computadores pessoais, o usuário pode requerer atividades entremeadas, tais como tocar música em um CD enquanto edita um documento e talvez imprima outro documento. Este tipo de utilização exige muita coordenação para evitar que atividades independentes interfiram umas nas outras, garantindo ainda uma comunicação eficiente e confiável entre atividades interdependentes. Tal coordenação é efetuada por um programa denominado sistema operacional.

Problemas similares de coordenação e de comunicação surgem quando diferentes máquinas são conectadas, formando uma rede de computadores. A resolução de tais problemas é uma extensão natural do tema estudado nos sistemas operacionais. Neste capítulo, discutimos os conceitos fundamentais relativos a sistemas operacionais e a redes de computadores.

- 3.1 A evolução dos sistemas operacionais**
Sistemas com um único processador
Sistemas com multiprocessamento
- 3.2 Arquitetura dos sistemas operacionais**
Uma inspeção no *software*
Componentes de um sistema operacional
Como iniciar o sistema operacional
- 3.3 A coordenação das atividades da máquina**
O conceito de processo
Administração de processos
O modelo cliente-servidor
- * 3.4 O tratamento da competição entre processos**
Semáforos
Enlace mortal (*deadlock*)
- 3.5 Redes**
Fundamentos de redes
A Internet
- * 3.6 Protocolos de redes**
Controle dos privilégios de transmissão
A abordagem em camadas para o *software* da Internet
O protocolo TCP/IP
- 3.7 Segurança**

*Os asteriscos indicam sugestões de seções consideradas opcionais.

3.1 A evolução dos sistemas operacionais

Abordamos o estudo dos sistemas operacionais e das redes sob uma perspectiva histórica, partindo dos antigos sistemas com um único processador até chegar aos sistemas com multiprocessamento mais modernos.

Sistemas com um único processador

As máquinas com um único processador, das décadas de 1940 e 1950, não eram flexíveis nem muito eficientes. A execução de programas exigia uma preparação considerável de equipamento, incluindo a montagem de fitas e a colocação física de cartões perfurados em leitoras de cartão, o posicionamento de chaves, e assim por diante. A execução de cada programa, chamada *trabalho*, era conduzida como atividade estanque. Quando vários usuários desejavam compartilhar uma mesma máquina, era comum o emprego de folhas de reserva de horário. Durante o período alocado a um usuário, a máquina ficava totalmente sob sua responsabilidade e controle. A sessão normalmente começava com a instalação do programa, seguida de curtos períodos de execução do mesmo e muitas vezes era finalizada com um esforço desesperado de fazer alguma coisa a mais (“vai levar só mais um minuto”), enquanto o usuário seguinte já começava impacientemente a sua instalação.

Em ambientes assim, os sistemas operacionais começaram a surgir como ambientes de *software* destinados a simplificar a instalação dos programas do usuário e a tornar suave a transição entre um trabalho e outro. Um primeiro progresso consistiu na separação entre usuários e equipamentos, eliminando a movimentação física de pessoas para dentro e para fora da sala do computador. Para isso, um operador de computadores era contratado para executar a operação propriamente dita da máquina. Aos que desejasse usar o computador, era solicitado que os programas a serem executados fossem submetidos ao operador, juntamente com os dados necessários e com eventuais instruções especiais sobre as requisições do programa, devendo o usuário retornar posteriormente para receber os resultados. O operador, por sua vez, transferia esse material para o armazenamento em massa da máquina, de onde o sistema operacional poderia acessá-lo, para promover sua execução. Esse foi o início do chamado **processamento em lote** (*batch processing*) — a execução de diversos trabalhos coletados em um lote, sem qualquer interação com o usuário.

Nos sistemas de processamento em lote, os trabalhos presentes no armazenamento em massa aguardam para ser executados em uma **fila de trabalhos** (Figura 3.1). Uma **fila** é um sistema de organização de armazenamento na qual objetos (no nosso caso, trabalhos) são ordenados de modo que o primeiro a entrar seja executado em primeiro lugar (**first-in, first-out — FIFO**), isto é, os objetos são retirados da fila na mesma ordem em que chegaram. Todavia, a maioria das filas de trabalhos na realidade não segue rigorosamente a disciplina FIFO, uma vez que, em sua maioria, os sistemas operacionais levam em conta as prioridades dos trabalhos. Como resultado, um trabalho de alta prioridade pode vir a ser executado antes de um outro, apesar de este último ter, eventualmente, chegado antes.

Nos primeiros processamentos em lote, cada trabalho era acompanhado de um conjunto de instruções explicativas, destinadas a orientar acerca da preparação da máquina para a execução do trabalho. Tais instruções eram codificadas em uma linguagem de controle de trabalhos (*job control language — JCL*) e mantidas na fila junto ao trabalho. Quando este era selecionado para ser executado, o sistema operacional imprimia essas instruções, para que fossem lidas e

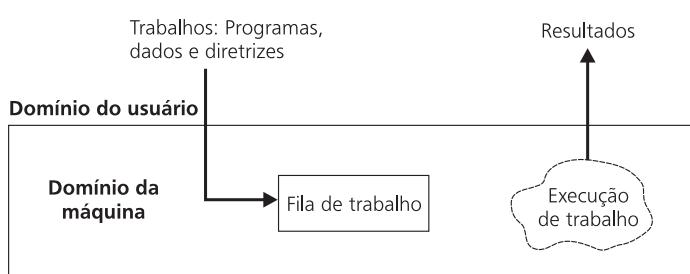


Figura 3.1 Processamento em lote.

executadas pelo operador. As instruções que necessitavam da ação direta do operador geralmente referiam-se a equipamentos desligados. Como atualmente as atividades dessa natureza são raras, as linguagens de controle de trabalho se transformaram em formas de comunicação com o sistema operacional, e não mais com o operador. De fato, a função de operador está ficando cada vez mais obsoleta. As instituições modernas, em vez de contratar operadores para fazer funcionar as máquinas manualmente, contratam administradores de sistema para efetuar a gestão do sistema de computação, obter e supervisionar a instalação de novos equipamentos e softwares, estabelecer de forma justa regulamentos locais, como a restrição e o compartilhamento de espaços no disco entre os usuários, e coordenar esforços para solucionar problemas que possam surgir no sistema.

A principal desvantagem dos tradicionais processamentos em lotes é a falta de interação do usuário com o programa, uma vez que este último seja submetido à fila de trabalhos. Este procedimento é aceitável para algumas aplicações, como é o caso do processamento de folhas de pagamento, em que os dados e todas as decisões do processo estão estabelecidos *a priori*. Contudo, não é aceitável quando o usuário necessita interagir com o programa durante a sua execução. Como exemplo, pode-se citar os sistemas de reserva de passagens ou similares, pois as reservas e os cancelamentos devem ser notificados assim que ocorrerem. Outro exemplo corresponde aos sistemas de processamento de texto, pois neles os documentos são escritos e reescritos dinamicamente. Um outro exemplo ainda são os jogos para computadores, para os quais a interação com a máquina é a característica mais significativa.

Para satisfazer a essas necessidades, foram desenvolvidos novos sistemas operacionais, que permitiam a execução de programa realizando um diálogo com o usuário por meio de terminais remotos ou estações de trabalho — característica conhecida como **processo interativo** (Figura 3.2). Esses sistemas interativos exigiam que o tempo gasto pela máquina para realizar as suas tarefas fosse compatível com as atividades no ambiente da mesma. (Usar um processador de textos seria frustrante se a máquina não pudesse acompanhar a digitação.) O serviço de computação proporcionado dessa maneira sincronizada ficou conhecido como **processamento em tempo real**.

Se os sistemas interativos permitissem atender a apenas um usuário de cada vez, o processamento em tempo real não deveria apresentar qualquer problema. Entretanto, os computadores eram caros e, por isso, cada um deveria servir simultaneamente a mais de um usuário. Por outro lado, por ser comum o fato de vários usuários solicitarem serviços interativos ao computador ao mesmo tempo, as características exigidas de um sistema de tempo real passaram a constituir um obstáculo. Se o sistema operacional, neste ambiente multiusuário, insistisse em executar somente um *trabalho* de cada vez, apenas um único usuário acabaria recebendo um atendimento satisfatório em tempo real.

Uma solução para este problema poderia ser um sistema operacional que revezasse a execução dos vários trabalhos por processo denominado **partilhamento de tempo** (*time-sharing*), que é uma técnica de dividir o tempo em intervalos ou fatias (*time slices*) e restringir a execução, dentro de cada fração de tempo, a um trabalho de cada vez. Ao término de cada intervalo, o trabalho corrente é retirado do processamento e um outro é posto em execução durante a próxima fatia de tempo. Dessa maneira, revezando rapidamente, a execução dos trabalhos, cria-se a ilusão de que vários trabalhos estão sendo executados simultaneamente. Dependendo dos tipos de trabalhos que estivessem sendo executados, os antigos sistemas de tempo partilhado chegavam a atender simultaneamente até cerca de 30 usuários, com uma resposta aceitável em tempo real.

Atualmente, a técnica de tempo partilhado é utilizada tanto em sistemas com um único usuário como em ambientes multiusuários, embora o primeiro geralmente seja denominado **multitarefa** (*multitasking*), referindo-se à ilu-

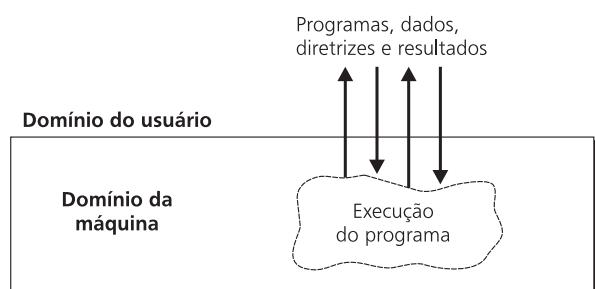


Figura 3.2 Processamento interativo.

são que propicia de haver mais de uma tarefa sendo executada ao mesmo tempo. Independentemente do número de usuários do ambiente, constatou-se que o conceito de tempo partilhado promovia o aumento da eficiência global de uma máquina. Esta constatação se mostrou particularmente surpreendente ao se levar em conta o considerável processamento adicional exigido para a implementação do revezamento que caracteriza a técnica de tempo partilhado. Entretanto, na sua ausência, um computador acaba gastando mais tempo enquanto espera que seus dispositivos periféricos completem suas tarefas, ou que um usuário faça sua próxima solicitação ao sistema. Em ambientes de tempo partilhado, este tempo pode ser cedido a alguma outra tarefa. Assim, enquanto uma tarefa espera pela sua vez de utilizar o processador, a outra pode prosseguir a sua execução. Como resultado, em um ambiente de tempo partilhado, um conjunto de tarefas pode ser concluído em tempo menor do que se fosse executado de modo seqüencial.

Sistemas com multiprocessamento

Mais recentemente, a necessidade de compartilhar informações e recursos entre diferentes máquinas suscitou o desejo de unir as máquinas para o intercâmbio de informações. Para preencher essa necessidade, popularizaram-se os sistemas com computadores interconectados, conhecidos como **redes** de computadores. De fato, o conceito de uma máquina central grande, que servisse a muitos usuários, foi substituído pelo conceito de muitas máquinas pequenas, conectadas por uma rede na qual os usuários compartilham recursos espalhados pela rede — como serviços de impressão, pacotes de *software*, equipamentos de armazenamento de dados e de informação. O principal exemplo é a **Internet**, uma rede de redes que hoje une milhões de computadores do mundo todo. Estudaremos a Internet com mais pormenores nas Seções 3.5 e 3.6.

Muitos dos problemas de coordenação que ocorrem em projetos de redes são iguais ou semelhantes aos enfrentados pelos sistemas operacionais. De fato, o *software* de controle de rede pode ser visto como um sistema operacional projetado para grandes redes. Sob este ponto de vista, o desenvolvimento de *software* de redes é uma extensão natural do campo dos sistemas operacionais. Enquanto as redes mais antigas foram construídas como um conjunto de computadores individuais, levemente interligados, cada qual sob o controle do seu próprio sistema operacional, as pesquisas recentes na área de redes estão se concentrando nos sistemas estruturados como grandes redes, cujos recursos são igualmente compartilhados entre as tarefas atribuídas à rede. Essas tarefas são designadas para ser executadas nos processadores da rede, de acordo com a necessidade, sem levar em consideração a posição física real de tais processadores. Um exemplo é o sistema de servidor de nomes utilizado na Internet, que estudaremos na Seção 3.5. Tal sistema permite que uma ampla gama de máquinas, espalhadas pelo mundo, possa trabalhar em conjunto para traduzir a forma humana e mnemônica de endereços da Internet para sua forma numérica, compatível com a rede.

As redes representam apenas um exemplo dos projetos de multiprocessadores que estão inspirando o desenvolvimento dos sistemas operacionais modernos. Enquanto uma rede produz um sistema com multiprocessamento mediante a combinação de máquinas, em que cada uma contém apenas um único processador, outros sistemas com multiprocessamento são projetados como computadores únicos, porém com mais de um processador. Um sistema operacional para tais computadores não apenas coordenará a competição entre as várias tarefas que são de fato executadas simultaneamente, mas também controlará a alocação de tarefas aos diversos processadores. Este processo envolve problemas de **balanceamento de carga** (alocação dinâmica de tarefas a vários processos, de modo a garantir que os processadores sejam utilizados eficientemente), bem como de **escalação** (divisão das tarefas em várias subtarefas, cujo número seja compatível com o número de processadores disponíveis na máquina).

Vimos então que o desenvolvimento de sistemas com multiprocessamento criou dimensões adicionais no estudo de sistemas operacionais, uma área que deverá se manter em franca atividade nos anos vindouros.



QUESTÕES/EXERCÍCIOS

1. Identifique exemplos práticos de filas. Em cada caso, indique alguma situação em que seja violada a estrutura FIFO.
2. Quais dos seguintes processos requerem processamento em tempo real?
 - a. Imprimir etiquetas para correspondência
 - b. Jogar no computador
 - c. Exibir letras na tela do monitor à medida que vão sendo digitadas
 - d. Executar um programa de previsão do estado da economia no próximo ano
3. Qual é a diferença entre processamento em tempo real e processamento interativo?
4. Qual é a diferença entre tempo partilhado e multitarefa?

3.2 Arquitetura dos sistemas operacionais

Para entender a composição de um sistema operacional comum, consideremos inicialmente o espectro completo de *softwares* encontrados em um sistema computacional comum. Então, concentraremos a atenção no sistema operacional.

Uma inspeção no software

Abordamos a nossa inspeção dos *softwares* encontrados em um sistema computacional comum apresentando um esquema para classificar o *software*. Essa classificação invariavelmente separa módulos semelhantes de *software* em diferentes classes, da mesma forma como os fusos horários definem a diferença de uma hora no relógio, mesmo para comunidades geograficamente próximas entre as quais não haja diferenças significativas no horário solar. Além disso, no caso da classificação de *software*, a dinâmica e a falta de uma autoridade definitiva no assunto conduzem a classificações e terminologias contraditórias. Por exemplo, os usuários do sistema operacional Windows da Microsoft encontram um grupo de programas chamados acessórios que inclui os *softwares* das nossas classes de aplicação e utilitários. Logo, a classificação seguinte deveria ser vista mais como uma forma de enfrentar um assunto complexo do que como uma taxonomia universalmente aceita.

Primeiramente, dividimos o *software* de um computador em duas grandes categorias: **software de aplicação** e **software de sistema**. (Figura 3.3) Software de aplicação consiste em programas que executam tarefas particulares de utilização da máquina. Um computador utilizado para manter o cadastro de uma companhia industrial contém softwares de aplicação diferentes dos utilizados por um engenheiro eletricista. Exemplos de *software* de aplicação incluem planilhas eletrônicas, sis-

Uniformidade benéfica ou monopólio prejudicial?

Uma vez que o sistema operacional de um computador estabelece a base na qual as comunicações com o computador são feitas, parece razoável que o uso de um sistema operacional padrão para máquinas muito diversas seja uma boa idéia. Esse padrão significaria que o conhecimento da utilização aprendido em um computador poderia ser facilmente adaptado a outras máquinas. Além disso, os desenvolvedores de *software* de aplicação não teriam de construir produtos compatíveis com múltiplos projetos de sistema operacional. Contudo, esses argumentos omitem muitos aspectos da sociedade atual. Mais especificamente, o produtor de um sistema operacional universal teria uma força tremenda no mercado, a qual, se fosse mal utilizada, poderia ser potencialmente mais prejudicial do que benéfica aos usuários de computador. Muitas dessas questões têm sido documentadas pelas partes envolvidas no processo antitruste que o governo dos EUA move contra a Microsoft desde 1998.

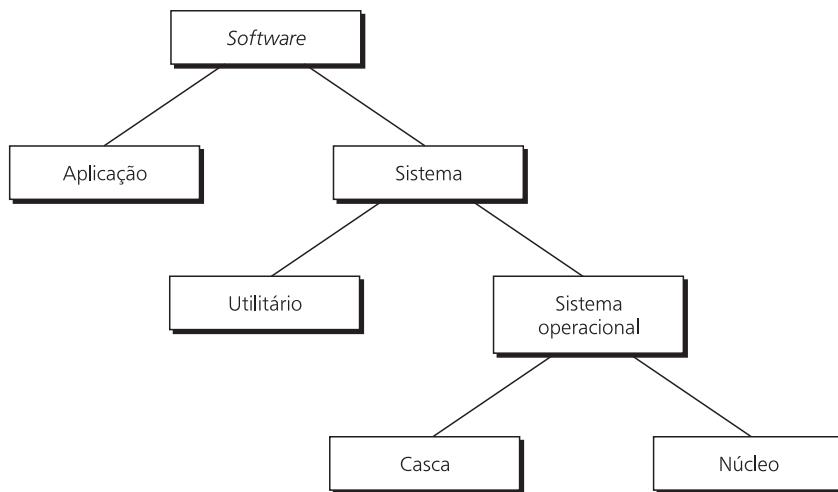


Figura 3.3
Classificação de software.

temas de bancos de dados, de edição de textos e de contabilidade, *software* de desenvolvimento de programas e jogos.

Em contraste com o *software* de aplicação, um *software* de sistema executa as tarefas comuns nos sistemas computacionais em geral. Até certo ponto, o *software* de sistema desenha o ambiente em que o *software* de aplicação se desenvolve, lembrando muito a maneira como a infra-estrutura de uma nação proporciona os fundamentos dos quais os cidadãos dependem para a concretização de seu modo de viver.

Na classe de *software* de sistema, podem ser consideradas duas categorias, uma das quais consiste no próprio sistema operacional, e a outra, em módulos de *software* do tipo conhecido como **utilitário**. A maioria dos softwares utilitários de uma instalação consiste em programas que executam tarefas essenciais à instalação embora não incluídas no sistema operacional. De certo modo, softwares utilitários consistem em módulos de *software* que ampliam as capacidades do sistema operacional. Por exemplo, a capacidade de efetuar a formatação de um disco ou a cópia de um arquivo em geral não são implementadas pelo próprio sistema operacional, mas por programas utilitários. Outros exemplos de softwares utilitários incluem os programas para comunicação por modem nas linhas telefônicas, *software* para compactar e descompactar arquivos e sistemas para gerenciar as comunicações em redes.

É fácil verificar que, implementando certos programas na forma de *software* utilitário, o sistema operacional resulta mais simples do que seria se tivesse de compatibilizar todos os recursos básicos exigidos pelo sistema computacional. Além disso, as rotinas implementadas como *software* utilitário podem ser mais facilmente personalizadas de acordo com as necessidades de uma instalação específica. Não é incomum encontrar companhias ou indivíduos que alteraram ou ampliaram algum dos softwares utilitários que acompanhavam originalmente o sistema operacional.

A distinção entre *software* de aplicação e *software* utilitário não é muito nítida. Sob o nosso ponto de vista, a distinção é se o pacote é parte da infra-estrutura de *software*. Assim, uma nova aplicação pode evoluir para um utilitário, caso se torne uma ferramenta fundamental. A distinção entre o *software* utilitário e o sistema operacional também é vaga. Alguns sistemas consideram os softwares que executam serviços básicos, como listar os arquivos do sistema de armazenamento em massa, como softwares utilitários; outros os incluem como partes do sistema operacional.

Componentes de um sistema operacional

A parte do sistema operacional que define a interface entre o sistema operacional e seus usuários é chamada **casca** (*shell*), cujo trabalho é de prover uma comunicação natural com o(s) usuário(s) da

máquina. As cascas modernas executam tal função por meio de uma **interface gráfica com o usuário** (*graphical user interface — GUI*) na qual os objetos a serem manipulados, tais como arquivos e programas, são representados pictoricamente por meio de ícones na tela do monitor. Essas interfaces permitem que os usuários emitam comandos ao sistema operacional, apontando e clicando os ícones da tela por meio de um dispositivo manual chamado *mouse*. As cascas mais antigas se comunicavam com os usuários via teclado e monitor, por mensagens textuais.

Embora a casca de um sistema operacional represente um papel importante na definição da funcionalidade de um computador, ela é somente uma interface entre o usuário e o verdadeiro coração do sistema operacional (Figura 3.4). Esta distinção entre a casca e as partes internas do sistema operacional é enfatizada pelo fato de alguns sistemas permitirem ao usuário selecionar, entre diversas cascas, aquela que lhe for mais adequada. Os usuários do sistema operacional UNIX, por exemplo, podem selecionar diversas cascas, incluindo o Borne shell, o C shell e o Korn shell. Versões mais antigas do Windows da Microsoft eram essencialmente cascas de substituição para o MS-DOS. Em casos como esse, o sistema operacional permanece o mesmo, exceto quanto ao modo como se comunica com os usuários da máquina.

Um componente fundamental das interfaces gráficas atuais é o **gerente de janelas**, que aloca blocos na tela chamados janelas e controla que aplicação está associada com cada janela. Quando a aplicação quer exibir alguma coisa na janela, ela notifica o gerente, que então coloca o padrão desejado na janela atribuída à aplicação. Entretanto, quando um botão do *mouse* é acionado, é o gerente de janelas que calcula a localização do *mouse* na tela e notifica a aplicaçãopropriada.

Em contraste com a casca de um sistema operacional, a sua parte interna geralmente é chamada **núcleo** (*kernel*), o qual contém os componentes de *software* que executam as funções mais básicas necessárias ao funcionamento de cada instalação computacional específica. Um destes módulos básicos é o **gerente de arquivos**, cuja função é coordenar o uso dos recursos de armazenamento em massa do computador. Mais precisamente, o gerente de arquivos controla todos os arquivos armazenados no dispositivo de armazenamento em massa, mantendo as informações sobre a localização de cada arquivo, os usuários autorizados a acessar os diversos arquivos e as áreas disponíveis no armazenamento em massa, para novos arquivos ou para a extensão de arquivos já existentes.

Para auxiliar os usuários, a maioria dos gerentes de arquivos permite que estes sejam agrupados em conjuntos chamados **pastas** ou **diretórios**. Essa abordagem permite ao usuário organizar seus arquivos de acordo com as respectivas finalidades, agrupando em cada diretório arquivos referentes a um mesmo assunto. Além disso, é possível criar uma organização hie-

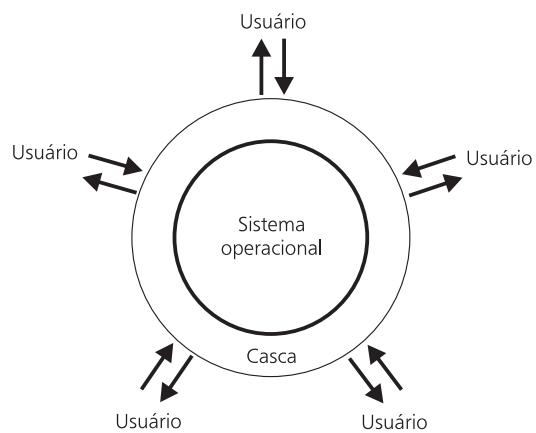


Figura 3.4 A casca como interface entre os usuários e o sistema operacional.

Linux

Para os entusiastas da computação que querem fazer experiências com os componentes internos de um sistema operacional, existe o Linux. Ele é um sistema operacional originalmente projetado por Linus Torvalds, quando era estudante na Universidade de Helsinqui. É um produto não proprietário, portanto, disponível juntamente com o seu código-fonte (veja o Capítulo 5) e sua documentação, sem custo. Uma vez que é gratuitamente disponibilizado na forma de programa-fonte, ele se tornou popular entre os que encaram a computação como passatempo, os estudantes de sistemas operacionais e os programadores em geral. Também se tornou popular como substituto de sistemas operacionais comerciais no mercado. Sem dúvida, o Linux é reconhecido como um dos sistemas operacionais mais confiáveis da atualidade. Você pode aprender mais sobre ele no sítio da Web em <http://www.linux.org>.

rárquica, possibilitando que cada diretório por sua vez possa conter subdiretórios. Por exemplo, um usuário pode criar um diretório chamado *Registros* que contenha subdiretórios chamados *RegistrosFinanceiros*, *RegistrosMédicos* e *RegistrosDomésticos*. Em cada um destes subdiretórios podem ser armazenados arquivos que pertençam à categoria correspondente. Uma seqüência de aninhamentos de níveis de diretórios é denominada **caminho** (*path*). Os caminhos normalmente são indicados listando os diretórios separados por barras. Por exemplo, *animais/pré-históricos/dinossauros* representa o caminho iniciado no diretório chamado *animais*, passando por seu subdiretório chamado *pré-históricos* e terminando no subdiretório *dinossauros*.

Qualquer acesso a arquivos, por parte de algum módulo de *software*, é efetuado através do gerente de arquivos. O procedimento inicia com a solicitação ao gerente para fazer acesso ao arquivo. Este procedimento é conhecido como “abrir o arquivo”. Se o gerente de arquivos aceitar o pedido, ele fornecerá a informação necessária para encontrar e manipular o arquivo. Essa informação é mantida em uma área da memória principal denominada **descriptor de arquivo**. É com base na informação contida nesse descriptor de arquivo que operações elementares individuais são executadas sobre o arquivo.

Outro componente do núcleo consiste em uma coleção de **dirigentes de dispositivo** (*device drivers*), que são as unidades de *software* que se comunicam com os controladores (ou, às vezes, diretamente com os dispositivos periféricos) para efetuar as operações nos dispositivos ligados à máquina. Cada dirigente é projetado para um tipo particular de dispositivo (como impressora, acionador de disco, unidade de fita magnética ou monitor) e traduz as requisições genéricas em passos mais técnicos exigidos pela dispositivo atribuído àquele dirigente. Por exemplo, um dirigente de dispositivo para uma impressora contém o *software* de leitura e decodificação da palavra de estado da impressora, bem como todos os detalhes do processo de comunicação. Assim, os outros componentes de *software* não precisam lidar com as tecnicidades quando desejam imprimir um arquivo. Em vez disso, eles podem meramente pedir ao dirigente de dispositivo para imprimir o arquivo e deixar que ele tome conta dos detalhes. Dessa maneira, o projeto das outras unidades de *software* pode ser independente das características únicas de um dispositivo em particular. O resultado é um sistema operacional genérico que pode ser personalizado para dispositivos particulares ao instalar os dirigentes apropriados.

Um outro componente do núcleo de um sistema operacional é o **gerente de memória**, que se encarrega de coordenar a utilização da memória principal da máquina. Essa função é mínima nos ambientes nos quais a máquina executa apenas uma tarefa a cada instante. Nesses casos, o programa que desempenha a tarefa é colocado na memória principal, executado e então substituído por outro, correspondente à próxima tarefa. Porém, em ambientes multiusuário ou multitarefa, nos quais a máquina se encarrega de várias atividades ao mesmo tempo, os deveres do gerente de memória são mais complexos. Nestes casos, muitos programas e blocos de dados devem coexistir na memória principal, cada um em uma área própria, determinada pelo gerente de memória. Conforme as necessidades das diferentes atividades, o gerente de memória vai providenciando as áreas necessárias e mantendo um mapa das regiões de memória que não estão mais ocupadas.

A tarefa do gerente de memória torna-se mais complexa quando a área total de memória principal solicitada excede o espaço realmente disponível na máquina. Neste caso, o gerente de memória pode criar a ilusão de espaço adicional alternando os programas e os dados entre a memória principal e o sistema de armazenamento em massa. Suponha, por exemplo, que seja solicitada uma memória de 256 MB, mas apenas 128 MB estejam de fato disponíveis. Para criar a ilusão de um espaço de memória maior, o gerente divide o espaço solicitado em unidades chamadas **páginas**, cujo conteúdo ele guarda no sistema de armazenamento em massa (uma página comum possui tamanho não superior a alguns kilobytes). Uma vez que diferentes páginas são necessárias na memória principal, o gerente deve trocá-las por outras que não estão mais em uso, e assim as outras unidades de *software* executam como se houvesse 256 MB de memória principal na máquina. Este espaço ilusório de memória é chamado **memória virtual**.

No núcleo de um sistema operacional também estão situados o **escalador** (*scheduler*) e o **despachante** (*dispatcher*), que estudaremos na próxima seção. Por ora, mencionamos apenas que, em um sistema de tempo partilhado, o escalador determina quais atividades serão executadas e o despachante controla a distribuição de fatias de tempo para tais atividades.

Como iniciar o sistema operacional

Vimos como um sistema operacional se comunica com os usuários e como os seus componentes trabalham em conjunto para coordenar a execução de atividades dentro da máquina. Entretanto, ainda não analisamos como se inicia a execução de um sistema operacional. Isto é feito por meio de um procedimento conhecido como **booting**, executado pela máquina todas as vezes que ela é ligada. Para compreender tal procedimento, deve-se, antes de mais nada, compreender a razão de ele ser executado.

Um processador é projetado de forma tal que, todas as vezes que for ligado, o conteúdo do seu contador de instruções seja devidamente preenchido com um endereço predeterminado. É nesse endereço que o processador encontra o programa a ser executado. Para assegurar que tal programa esteja sempre presente, a área de memória normalmente é projetada de modo que seu conteúdo seja permanente. Essa memória é conhecida como **memória para leitura apenas** (*read-only memory — ROM*). Uma vez que os padrões de bits sejam instalados na ROM, o que se faz mediante um processo de gravação análogo ao da solda de fusíveis em uma pastilha, a informação se conserva indefinidamente, mesmo com o desligamento da máquina.

No caso de computadores pequenos, como os utilizados como dispositivos de controle em fornos de microondas, em sistemas de ignição de automóveis e em receptores estéreo de rádio, é possível implementar áreas significativamente grandes da memória principal em tecnologia ROM, uma vez que, nestes casos, o objetivo principal não é a flexibilidade, já que o programa a ser executado por tais dispositivos será sempre o mesmo, todas as vezes que for acionado. Contudo, isto não ocorre em computadores de propósito geral, de modo que não é prático realizar grandes áreas de memória principal em ROM nessas máquinas. De fato, em máquinas de propósito geral, a maior parte da memória é volátil, o que significa que o seu conteúdo é perdido sempre que a máquina for desligada.

Para ser possível dar partida em uma máquina de propósito geral, a sua área de memória ROM é pré-programada com uma rotina chamada **bootstrap**, executada automaticamente toda vez que a máquina é ligada. Ela faz o processador transferir o conteúdo de uma área predeterminada do disco para uma região volátil da memória principal (Figura 3.5). Na maioria dos casos, este material é o sistema operacional. Uma vez transferido o sistema operacional para a memória principal, a rotina *bootstrap* prepara o processador para executá-lo e lhe transfere o controle da máquina. A partir de então, todas as atividades da máquina passam a ser controladas pelo sistema operacional.

Na maioria dos computadores pessoais, a rotina *bootstrap* é projetada para primeiro tentar extrair o sistema operacional do disco flexível. Se nenhum disco estiver inserido na máquina, então o sistema operacional será extraído do disco rígido. Contudo, se um disco flexível estiver inserido, mas não conter

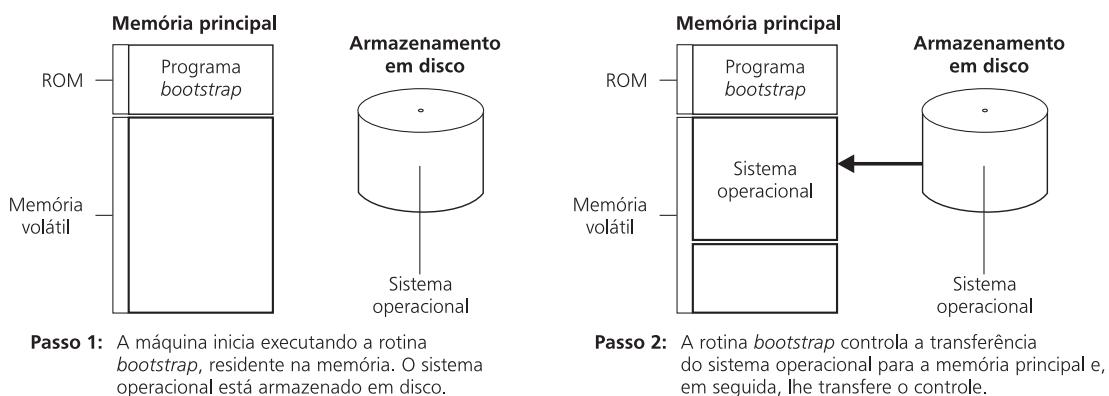


Figura 3.5 O processo de *booting*.

o sistema operacional, a rotina *bootstrap* fará uma pausa e emitirá uma mensagem de erro para o operador da máquina. Você provavelmente já experimentou esse fenômeno ao ligar um computador pessoal em que havia um disco de dados no acionador.



QUESTÕES/EXERCÍCIOS

1. Liste os componentes de um sistema operacional comum e resuma a função de cada um em uma única frase.
2. Qual é a diferença entre *software* de aplicação e *software* utilitário?
3. O que é memória virtual?
4. Resuma o procedimento de *booting*.

3.3 A coordenação das atividades da máquina

Nesta seção, examinaremos como um sistema operacional coordena a execução dos *softwares* de aplicação, dos utilitários e dos diversos módulos internos do próprio sistema operacional. Iniciamos com o conceito de processo.

O conceito de processo

Um dos conceitos mais fundamentais dos sistemas operacionais modernos é a distinção entre o programa e a atividade de executá-lo. O programa é apenas um conjunto estático de diretrizes e sua execução é uma atividade dinâmica, cujas propriedades mudam à medida que o tempo avança. Esta atividade é conhecida como **processo**. Um processo leva em conta a situação corrente da atividade, conhecida como **estado do processo**. Este estado inclui a posição do programa atualmente em execução (o valor do Contador de Instruções), bem como os valores contidos nos outros registradores do processador e as posições associadas de memória. Em termos gerais, o estado do processo fornece uma fotografia da situação da máquina em um dado momento. Em diferentes instantes da execução de um programa (nos diversos momentos de um processo), serão observadas diferentes fotografias do estado do processo.

Para enfatizar a distinção entre um programa e um processo, note-se que um único programa pode ser associado a mais de um processo em um mesmo instante. Por exemplo, em um sistema multiusuário de tempo partilhado, dois usuários podem editar documentos separados ao mesmo tempo. As duas atividades utilizam um mesmo programa editor de textos, mas cada uma caracteriza um processo separado, com seu próprio conjunto de dados e sua própria taxa de progresso. Nesta situação, o sistema operacional pode manter na memória principal uma só cópia do programa editor e permitir que cada processo o utilize à sua maneira, durante a fatia de tempo que lhe couber.

Em uma típica instalação de tempo partilhado, é natural que vários processos compitam pelas fatias de tempo. Esses processos englobam a execução de programas de aplicação e utilitários, bem como porções do sistema operacional, que tem a tarefa de coordenar todos esses processos. Essa atividade de coordenação inclui garantir que cada processo tenha acesso aos recursos de que necessita (dispositivos periféricos, área na memória principal, acesso a dados e acesso ao processador), que processos independentes não interfiram uns com os outros e que processos que se intercomunicam tenham a possibilidade de trocar informação entre si.

Administração de processos

As tarefas associadas à coordenação de processos são manuseadas pelo escalador e pelo despachante no interior do núcleo do sistema operacional. Assim, o escalador mantém um registro dos processos presentes no sistema computacional, inclui novos processos nesse conjunto e remove processos que já completaram sua missão. Para cuidar de todos os processos, o escalador mantém na memória principal um conjunto de dados em uma estrutura denominada **tabela de processos**. Cada vez que a máquina recebe uma nova tarefa, o escalador cria para ela um processo, acrescentando uma nova linha à tabela de processos. Esta linha contém diversos indicadores: a área de memória designada para o processo (obtida por meio do gerente de memória), a prioridade do processo e um indicador de que ele está pronto para ser executado ou à espera de algum evento. Diz-se que um processo está **pronto** para ser executado quando está em um estado a partir do qual sua atividade possa prosseguir; o processo estará em **estado de espera** se seu progresso estiver sendo bloqueado até que seja registrada a ocorrência de algum evento externo, tal como a conclusão de um acesso ao disco ou o recebimento de uma mensagem, enviada por algum outro processo.

O despachante é o módulo do núcleo do sistema operacional cuja função é a de assegurar que os processos escalados sejam de fato executados. Em um sistema de tempo partilhado, esta tarefa é realizada dividindo-se o tempo físico em pequenas **fatias** (*time slice*), geralmente de cerca de 50 milissegundos. A atenção do processador é revezada entre os processos, a cada qual é concedido um intervalo de tempo não superior à duração de uma fatia (Figura 3.6). O procedimento de alternar o processador de um processo para outro é denominado **chaveamento de processos**.

Cada vez que um processo inicia o uso de sua fatia de tempo, o despachante dispara um circuito temporizador, encarregado de medir a próxima fatia, ao término da qual o temporizador gera um sinal denominado **interrupção**. O processador reage a este sinal de uma forma muito parecida ao modo como alguém reage quando interrompido durante a execução de alguma tarefa: interrompe o que estiver fazendo, registra o ponto da tarefa no qual foi interrompido e então passa a atender ao evento que provocou a interrupção. Quando o processador recebe uma interrupção, completa o seu ciclo de máquina corrente, guarda a posição em que se encontra (retornaremos logo mais a este assunto) e começa a executar um programa, chamado **rotina de tratamento de interrupção**, o qual deve ter sido depositado previamente em uma região predeterminada da memória principal.

Em nosso ambiente de tempo partilhado, o programa de tratamento de interrupção faz parte do próprio despachante. Assim, como efeito do sinal de interrupção, tem-se o bloqueio da continuidade do processo em andamento, devolvendo-se o controle ao despachante. Neste momento, o despachante permite que o escalador atualize a tabela de processos (por exemplo, a prioridade do processo que acaba de esgotar a sua fatia de tempo poderá ser reduzida e as prioridades dos demais, aumentadas). O despa-

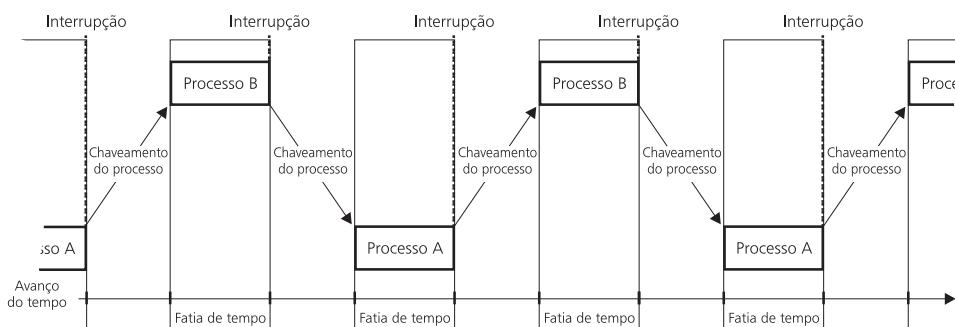


Figura 3.6 Partilhamento de tempo entre os processos A e B.

chante então seleciona o processo de maior prioridade dentre os que se encontrarem prontos, reinicia a operação do temporizador e permite que o processo selecionado inicie a sua fatia de tempo.

Fundamental em um sistema de tempo partilhado é a capacidade de parar um processo para continuá-lo mais tarde. Caso ocorra uma interrupção durante a leitura de um livro, a capacidade do leitor de continuar a leitura mais tarde depende de sua habilidade de relembrar o ponto em que parou, bem como de reter a informação acumulada até tal ponto. Em suma, deve ser capaz de recriar o ambiente existente imediatamente antes da ocorrência da interrupção. Esse ambiente é denominado *estado* do processo. Lembre-se que este estado inclui o valor do contador de instruções, o conteúdo dos registradores e os das posições relevantes de memória. As máquinas projetadas para operar sistemas de tempo partilhado incluem recursos para guardar tal estado a cada ocorrência de interrupção. Também possuem instruções em linguagem de máquina para recarregar um estado anteriormente armazenado. Tais características simplificam a tarefa, de responsabilidade do despachante, de alternar os processos e ilustram até que ponto o projeto das máquinas modernas pode ser influenciado pelas necessidades dos sistemas operacionais.

Às vezes, a fatia de tempo de um processo termina antes do tempo determinado pelo temporizador. Por exemplo, se um processo executar uma operação de E/S, solicitando dados de um disco, sua fatia de tempo será truncada pelo sistema, uma vez que, de outra forma, tal processo desperdiçaria o tempo restante dessa fatia, aguardando que o controlador terminasse de executar a operação solicitada. Neste caso, o escalador atualizará a tabela de processos, marcando o processo corrente como em estado de espera, e o despachante fornecerá uma nova fatia a outro processo que já esteja pronto para ser executado. Depois (talvez várias centenas de milissegundos mais tarde), quando o controlador indicar que aquela operação de E/S foi completada, o escalador reclassificará o processo como pronto para a execução, habilitando-o assim a competir novamente por outra fatia de tempo.

O modelo cliente-servidor

As diversas unidades internas de um sistema operacional funcionam como processos independentes que, em um sistema de tempo partilhado, competem por fatias de tempo, sob a supervisão do despachante. Tais processos se intercomunicam para coordenar suas atividades. Por exemplo, para escalar um novo processo, o escalador solicita espaço de memória ao gerente de memória. Para acessar um arquivo em disco, o processo deve primeiro obter informações do gerente de arquivos.

A troca de mensagens entre processos é chamada **comunicação entre processos** e representa uma área extensa de pesquisa. De fato, a comunicação entre processos pode ser feita de várias formas. Uma delas, chamada **modelo cliente-servidor** (Figura 3.7) tornou-se muito comum no campo das redes de computadores. O modelo define as funções básicas executadas pelos componentes como a de **cliente**, o qual envia suas solicitações para outras unidades, e a de **servidor**, que satisfaz as solicitações recebidas dos clientes. Por exemplo, o gerente de arquivos de um sistema operacional pode funcionar como um servidor, fornecendo acessos a arquivos conforme as solicitações dos seus clientes.

O uso do modelo cliente-servidor em projetos de *software* leva a uma padronização dos tipos de comunicação existentes no sistema. Um cliente apenas faz solicitações aos servidores e espera as suas

respostas, enquanto um servidor apenas executa os serviços solicitados e envia as respostas correspondentes aos clientes. O papel de um servidor é o mesmo para clientes que residem na mesma máquina ou em máquinas distantes ligadas na rede. A distinção fica com o *software* que controla a comunicação — não nos clientes e servidores.

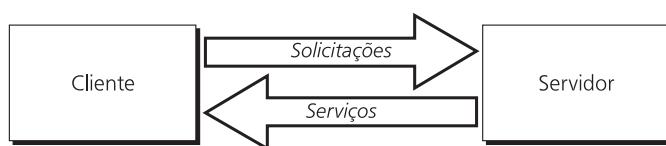


Figura 3.7 O modelo cliente-servidor.

Logo, se os componentes de um sistema de *software* forem projetados como clientes e servidores, eles poderão desempenhar as suas tarefas independentemente de residirem na mesma máquina ou em máquinas separadas por grandes distâncias (Figura 3.8). Assim, desde que a rede forneça meios para enviar solicitações e respostas, uma coleção de clientes e servidores pode ser distribuída na rede na configuração que for mais conveniente.

O desejo de estabelecer um sistema uniforme de troca de mensagens que possa suportar essa configuração distribuída nas redes de computadores é a meta subjacente do conjunto de padrões e especificações conhecido como CORBA (Common Object Request Broker Architecture — arquitetura para facilitar as requisições comuns aos objetos). Em síntese, o CORBA provê um padrão para a comunicação em rede entre unidades de *software* conhecidas como objetos (como clientes e servidores). Ele foi desenvolvido pelo Object Management Group, um consórcio de fabricantes de *hardware* e *software* e de usuários interessados em promover e expandir o escopo da tecnologia orientada a objetos — assunto que introduziremos no Capítulo 5 e que voltará a ser discutido nos capítulos seguintes.



QUESTÕES/EXERCÍCIOS

1. Resuma as diferenças entre programa e processo.
2. Resuma os passos executados pelo processador quando ocorre uma interrupção.
3. Em um sistema de tempo partilhado, de que forma os processos de maior prioridade são executados antes dos demais?
4. Em um sistema de tempo partilhado, se cada fatia tiver a duração de 50 milissegundos e cada chaveamento de processo gastar 1 microsegundo, quantos processos a máquina atenderá em um único segundo?
5. Se, na máquina do exercício 4, cada processo sempre esgotar toda a sua fatia, qual percentagem de tempo da máquina será realmente empregada para executar processos? Qual seria esta parcela se cada processo executasse um pedido de E/S 1 microsegundo após o início da sua fatia?
6. Identifique na sociedade algumas relações que estejam de acordo com o modelo cliente-servidor.

3.4 O tratamento da competição entre processos

Uma importante tarefa de um sistema operacional é a alocação dos recursos da máquina aos processos no sistema. Aqui usamos o termo *recurso* em sentido amplo, incluindo os dispositivos periféricos da máquina, bem como elementos internos da própria máquina. O gerente de arquivos autoriza o acesso a arquivos existentes, aloca espaço em disco para a construção de arquivos novos; o gerente de memória aloca espaço de memória; o escalador, espaço na tabela de processos; e o despachante, fatias de tempo. Como acontece com muitos problemas em sistemas computacionais, esta tarefa de alocação pode

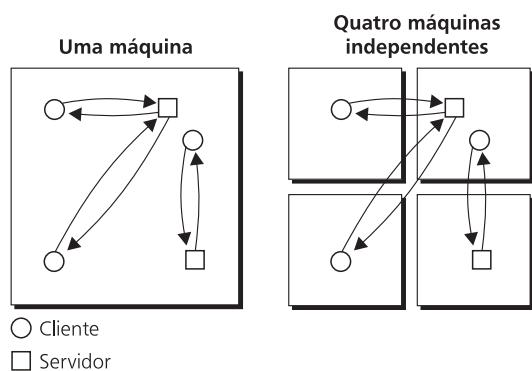


Figura 3.8 Estrutura idêntica de comunicação entre os clientes e os servidores, operando na mesma máquina e distribuída entre máquinas diferentes.

parecer simples à primeira vista. Por trás dela, entretanto, estão presentes vários problemas que, se não forem devidamente tratados, podem levar à ocorrência de falhas no sistema. É bom lembrar que uma máquina não pensa por si mesma, mas apenas segue diretrizes. Assim, para construir sistemas operacionais confiáveis, é necessário desenvolver algoritmos que levem em conta cada detalhe possível do sistema, independentemente insignificante ele possa parecer.

Semáforos

Consideremos uma máquina com uma única impressora, em que esteja sendo executado um sistema operacional de tempo partilhado. Se um processo precisar imprimir seus resultados, deverá solicitar ao sistema operacional acesso ao dirigente da impressora. Nesse caso, o sistema operacional deve decidir se o pedido deve ser atendido ou não, conforme a impressora esteja livre ou ocupada por algum processo. Se não estiver ocupada, o sistema operacional atenderá à solicitação e permitirá que o processo solicitante prossiga sua execução; caso contrário, o pedido será negado e o processo, marcado como à espera de que a impressora fique disponível. De fato, se for permitido que dois processos tenham acesso simultâneo à impressora, a listagem embaralhada assim obtida será inútil para ambos.

A alocação de acesso exige que o sistema operacional mantenha atualizada a informação do estado de alocação da impressora. Uma solução possível consiste em empregar uma variável sinalizadora que, neste contexto, pode ser materializada na forma de um *bit* de memória, cujos estados frequentemente denominamos *ligado* e *desligado* (*set* e *clear*), em vez de 1 e 0. Um sinalizador desligado indica que a impressora está disponível, enquanto um ligado indica que ela está alocada no momento. Aparentemente, essa solução parece não esconder qualquer problema imprevisto. No início da operação, o sistema operacional simplesmente desliga o sinalizador, e a partir daí, toda vez que for recebido um pedido de acesso à impressora, verifica o valor assumido pelo sinalizador. Se este estiver desligado, o pedido será atendido e o sistema operacional deverá então ligar o sinalizador. Se este já estiver ligado, o sistema operacional colocará o processo requisitante em estado de espera. Cada vez que um processo terminar de utilizar a impressora, o sistema operacional alocará a um processo em espera ou, se nenhum processo estiver nesse estado, simplesmente desligará o sinalizador.

Embora esta solução pareça boa à primeira vista, ela apresenta um problema. A tarefa de testar e ligar o sinalizador requer a execução de diversas instruções de máquina. Portanto, é possível que a tarefa seja interrompida logo após ser detectada a situação de sinalizador desligado, porém antes de haver tempo para o mesmo seja religado. Por isso, pode ocorrer a situação descrita a seguir.

Suponha que, neste instante, a impressora esteja disponível e sendo solicitada por um processo. O seu sinalizador é verificado e, por se encontrar no estado desligado, indica que a impressora está disponível. Suponha que, por coincidência, neste momento o processo seja interrompido e que outro processo inicie a utilização de sua fatia de tempo, e também solicite o uso da impressora. Novamente, o sinalizador será verificado, mas o mesmo ainda se encontra desligado, já que o processo anterior fora interrompido antes que o sistema operacional pudesse ligá-lo. Como consequência, o sistema operacional permitirá que o segundo processo utilize a impressora. Mais tarde, o primeiro processo retomará a execução a partir do ponto em que havia sido interrompido, isto é, imediatamente depois da instrução que executava quando o sistema operacional detectou o desligamento do sinalizador. Desse modo, o sistema operacional permitirá ao primeiro processo o acesso à impressora, e, portanto, os dois processos estarão agora utilizando a mesma impressora, ao mesmo tempo.

O problema, neste ponto, é que a tarefa de testar e, provavelmente, de ligar o sinalizador, uma vez iniciada, deve ser completamente executada, sem interrupções. Uma solução é utilizar as instruções de habilitação (*enable*) ou de desabilitação (*disable*) de interrupção, disponíveis na maioria das linguagens de máquina. Quando executada, uma instrução de desabilitação faz com que futuras interrupções sejam bloqueadas, enquanto uma instrução de habilitação faz com que a UCP volte a responder aos sinais de interrupção. Assim, se o sistema operacional iniciar a rotina de teste do sinalizador com uma instrução de desabilitação de interrupção e terminar com uma de habilitação, uma vez iniciada essa rotina, nenhuma outra atividade poderá interrompê-la.

Outra abordagem é usar a instrução de *test-and-set*, disponível em muitas linguagens de máquina. Esta instrução permite ao processador obter o valor do sinalizador, anotar o valor recebido e então ligar o sinalizador, tudo dentro de uma única instrução de máquina. A vantagem é que, como a UCP sempre completa uma instrução antes de reconhecer uma interrupção, a tarefa de testar e ligar o sinalizador não será interrompida quando for implementada como uma única instrução.

Um sinalizador adequadamente implementado, conforme descrito, é conhecido com o nome de **semáforo**, em referência aos sinais ferroviários utilizados para controlar o acesso aos trilhos. De fato, os semáforos são usados em sistemas de *software* do mesmo modo como em ferrovias. Em lugar dos segmentos de trilhos, que podem conter somente um trem por vez, está a seqüência de instruções que podem ser executadas apenas por um processo de cada vez. Tal seqüência é chamada de **região crítica**. A exigência de que apenas um processo possa executar uma região crítica é conhecida como **exclusão mútua**. Em resumo, uma maneira comum de obter a exclusão mútua em uma região crítica é controlá-la por um semáforo. Para entrar na região crítica, um processo deve encontrar o semáforo desligado e então ligá-lo antes de entrar; depois, logo que sair, deve desligar o semáforo. Se o semáforo for encontrado ligado, o processo que tentou entrar na região crítica deverá esperar até que o semáforo seja desligado.

Enlace mortal (*deadlock*)

Outro problema que pode surgir durante a alocação de recursos é o **enlace mortal**, situação na qual dois ou mais processos ficam impedidos de prosseguir suas execuções, devido ao fato de cada um estar aguardando acesso a recursos já alocados ao outro. Por exemplo, um processo tem acesso à impressora, mas está esperando pelo dispositivo de fita, enquanto outro processo tem acesso ao dispositivo de fita, mas está aguardando a impressora. Outro exemplo acontece quando processos criam novos processos para executar subtarefas. Se o escalador não dispuser de espaço na tabela de processos e cada processo no sistema precisar criar um processo adicional antes de poder completar sua tarefa, nenhum processo poderá continuar. A ocorrência de tais situações, assim como a de diversas outras configurações (Figura 3.9), pode reduzir dramaticamente o desempenho de um sistema.

A análise do enlace mortal revelou que ele não ocorre, a menos que sejam satisfeitas simultaneamente as três condições seguintes:

1. Há competição por recursos não-compartilháveis.
2. Os recursos são solicitados de forma parcial, ou seja, já de posse de alguns recursos, um processo volta a solicitar mais recursos em instante posterior.
3. Uma vez alocado, um recurso não pode ser retomado pelo sistema, a não ser que o processo o libere espontaneamente.

Isoladas essas três condições, torna-se possível resolver o problema do enlace mortal pela simples eliminação de qualquer uma delas. Em geral, as técnicas que tratam a terceira condição pertencem à categoria conhecida como esquemas de detecção e correção de enlaces mortais. Nestes casos, a ocorrência do enlace é considerada tão remota que nenhum esforço é feito para evitá-la. Pelo contrário, a forma empregada é a de localizar a sua ocorrência e então corrigir a situação retro-

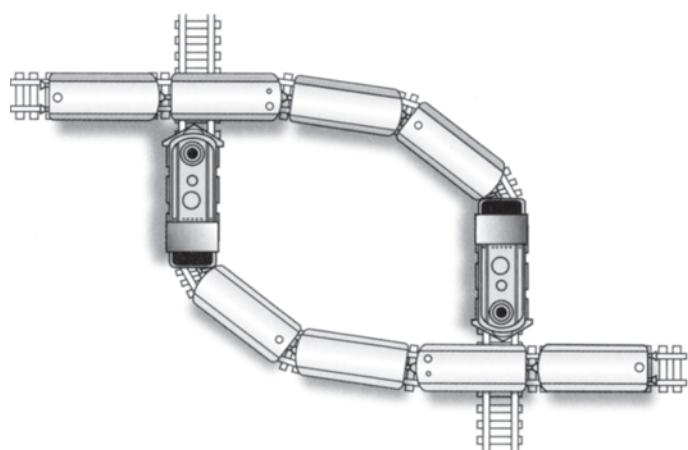


Figura 3.9 Um enlace mortal resultante de competição por intersecções não-compartilháveis de transporte ferroviário.

mando compulsoriamente alguns dos recursos alocados. O nosso exemplo da tabela de processos completamente lotada se encaixa nesta situação. Um administrador de sistema normalmente dimensiona uma tabela de processos suficientemente grande para cada instalação. Entretanto, se os enlaces mortais ocorrerem por causa de alguma tabela lotada, o administrador precisa apenas lançar mão de seu privilégio de *superusuário* para remover (o termo técnico é *kill* — matar) alguns dos processos que lotam esta tabela, garantindo assim que os demais possam continuar suas tarefas.

As técnicas de tratamento das duas primeiras condições são conhecidas como esquemas para evitar os enlaces mortais. Por exemplo, elimina-se a segunda condição exigindo que cada processo solicite todos os recursos necessários de uma só vez. Outra técnica, talvez mais criativa, elimina a primeira condição, sem remover diretamente a competição, mas convertendo recursos não-compartilháveis em compartilháveis. Por exemplo, suponha que o recurso em questão seja uma impressora e vários processos solicitem o seu uso. Toda vez que um processo solicitar a impressora, o sistema operacional atenderá o pedido. Contudo, em lugar de conectar o processo ao dirigente da impressora, o sistema operacional o conecta a um dirigente de dispositivo que armazena a informação a ser impressa em um disco, em vez de enviá-la à impressora. Assim, sob esta configuração, tudo se passa como se cada processo tivesse acesso à impressora e, portanto, cada um continua normalmente a sua execução. Mais tarde, quando a impressora estiver disponível, o sistema operacional transferirá os dados do disco para ela. Desta maneira, o sistema operacional fez um recurso não-compartilhável comportar-se como se fosse compartilhável, criando a ilusão de haver mais de uma impressora. Esta técnica de armazenar dados para uma saída posterior em uma ocasião mais oportuna é chamada de **spooling**, e é muito comum em sistemas de todos os tamanhos.

Naturalmente, outros problemas podem surgir quando existe competição dos processos pelos recursos de uma máquina. Por exemplo, um gerente de arquivos, em geral, concede a vários processos o acesso a um mesmo arquivo, desde que se trate de acessos apenas para a leitura do arquivo. Nesse caso, ocorrerão conflitos se mais de um processo tentar alterar um mesmo arquivo ao mesmo tempo. Assim, um gerente de arquivos permite acesso ao arquivo de acordo com as necessidades dos processos, permitindo que vários leiam os dados, mas que somente um grave informações em um determinado momento. Outros sistemas dividem o arquivo de forma que diferentes processos possam alterar diferentes partes do mesmo arquivo concomitantemente. Todavia, ainda há problemas que devem ser solucionados. Por exemplo, de que maneira esses processos, que possuem autorização para ler dados de um arquivo, devem ser notificados quando um outro processo estiver alterando o conteúdo do arquivo?



QUESTÕES/EXERCÍCIOS

1. Suponha que os processos A e B compartilhem tempo na mesma máquina e que ambos necessitem de um mesmo recurso não-compartilhável por pequenos períodos (por exemplo, cada processo imprime uma série de pequenos relatórios independentes). Cada processo pode, repetidamente, acessar um recurso, liberá-lo e depois solicitá-lo novamente. Descubra uma situação adversa quando se resolve controlar os acessos aos recursos da seguinte maneira:

Comece atribuindo o valor 0 a um sinalizador. Se o processo A solicitar o recurso e o sinalizador contiver 0, atenda o pedido. Caso contrário, faça o processo A esperar. Se o processo B solicitar o recurso e o sinalizador for 1, atenda o pedido. Caso contrário, faça o processo B esperar. Todas as vezes que o processo A liberar o recurso, altere o sinalizador para 1. Todas as vezes que o processo B terminar de utilizar o recurso, altere o sinalizador para 0.

2. Suponha que uma estrada de dupla pista se reduza a uma pista única ao atravessar um túnel. Para coordenar o uso do túnel, foi instalado o seguinte sistema de sinalização:

Ao entrar por qualquer extremidade do túnel, um carro faz com que seja ligado um sinal luminoso vermelho nas duas aberturas do túnel. No momento em que o carro sair, esse sinal

será desligado. Se, ao aproximar-se do túnel, outro carro encontrar ligado o sinal vermelho, deverá aguardar até que ele se apague antes de entrar no túnel. Qual é o defeito deste esquema?

3. Suponha que as soluções seguintes tenham sido propostas para remover o enlace mortal que ocorre em uma ponte de pista única quando dois carros se defrontam. Das condições de ocorrência de enlaces mortais descritas no texto, identifique qual delas é removida quando se emprega cada uma das soluções propostas a seguir:
 - a. Não deixar um carro subir na ponte enquanto esta não estiver vazia.
 - b. Se dois carros se defrontarem, fazer um deles voltar atrás.
 - c. Acrescentar uma nova pista à ponte.
4. Suponha que, em um sistema de tempo partilhado, representemos cada processo por um ponto e que desenhemos uma seta de um ponto a outro se o processo representado pelo primeiro ponto estiver esperando por um recurso ocupado pelo segundo. O quadro resultante é chamado pelos matemáticos de *grafo orientado*. Qual propriedade do grafo orientado representa um enlace mortal no sistema?

3.5 Redes

As primeiras redes de computadores eram constituídas de máquinas independentes que, basicamente, apenas efetuavam transferências de arquivos através de conexões telefônicas temporárias, usando os softwares utilitários dos sistemas operacionais. Atualmente, a interação de computadores por meio de redes tornou-se comum e multifacetada. Muitos sistemas de software modernos, tais como de recuperação de informação global, de contabilidade e estoque de grandes empresas e mesmo alguns jogos de computador, são projetados como **sistemas distribuídos**, o que significa que consistem em unidades em execução em diferentes computadores de uma rede. Já o software subjacente necessário para suportar tais aplicações cresceu desde um simples pacote utilitário até um sistema expansível de software de rede, que proporciona uma infra-estrutura em toda a rede. Em um certo sentido, então, os softwares de rede vêm evoluindo para um sistema operacional de redes. No restante deste capítulo, consideraremos alguns tópicos associados a um contexto expansível de software de sistema.

Fundamentos de redes

Cada rede de computadores pode ser classificada em uma das duas grandes categorias: **redes locais** (*local area network* — LANs) e **redes de longa distância** (*wide area networks* — WANs). Uma rede local em geral é formada por um conjunto de computadores, localizados em um único edifício ou em um complexo de edifícios. Por exemplo, os computadores existentes em um *campus* universitário ou em uma indústria podem ser interconectados por uma rede local. Uma rede de longa distância une máquinas que podem estar em pontos opostos de uma cidade, ou mesmo do mundo.

Outra dicotomia existente nas redes é baseada na propriedade do projeto, se de domínio público ou de uma única corporação. Uma rede do primeiro tipo é chamada **rede aberta** e uma do segundo, **rede fechada** ou **rede proprietária**. A Internet é um sistema aberto. De fato, a comunicação através da Internet é controlada por uma coleção aberta de padrões conhecida como TCP/IP, conjunto de protocolos que discutiremos na próxima seção. A empresa Novell, por sua vez, é um grande fornecedor de software de rede, desenvolvido por ela e de sua propriedade. Assim, os sistemas de rede instalados e mantidos pela Novell são sistemas fechados.

Outro modo de classificar redes está baseado em sua topologia e refere-se ao padrão com que as máquinas são conectadas. A Figura 3.10 representa algumas topologias mais conhecidas: (a) em anel, na qual as máquinas são conectadas circularmente; (b) em via, em que são conectadas por uma linha comum de comunicação chamada *via*; (c) em estrela, na qual uma das máquinas serve como uma central,

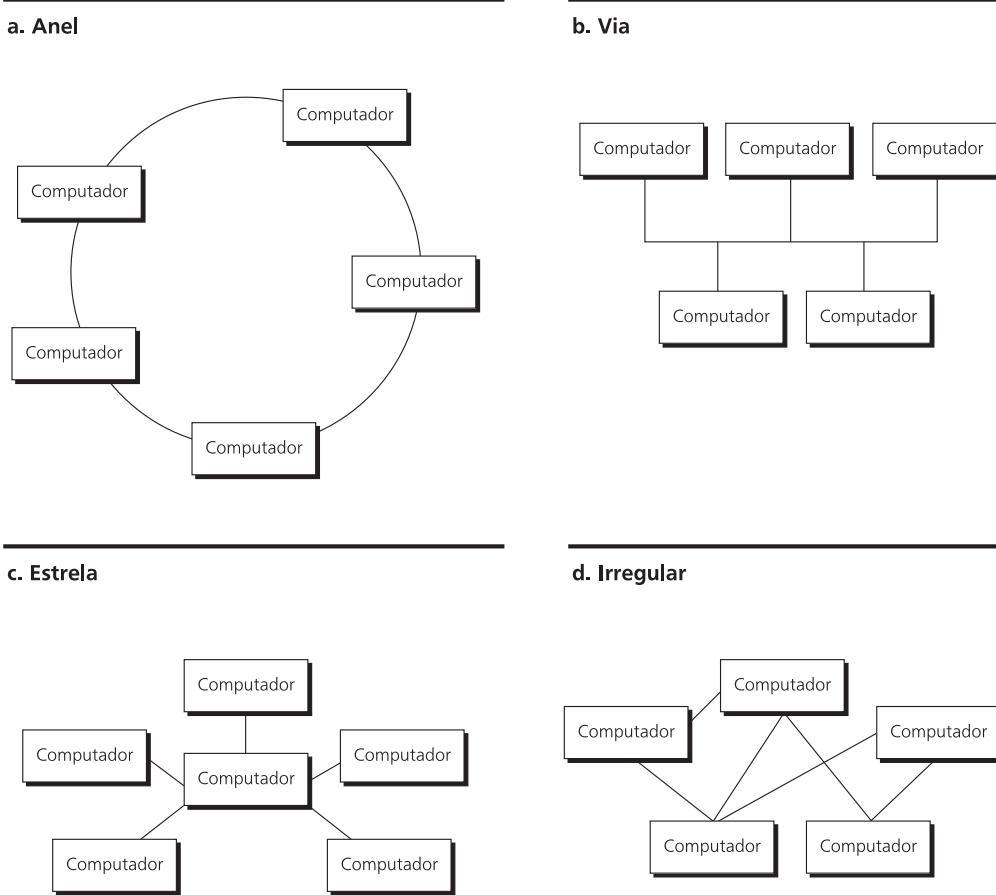


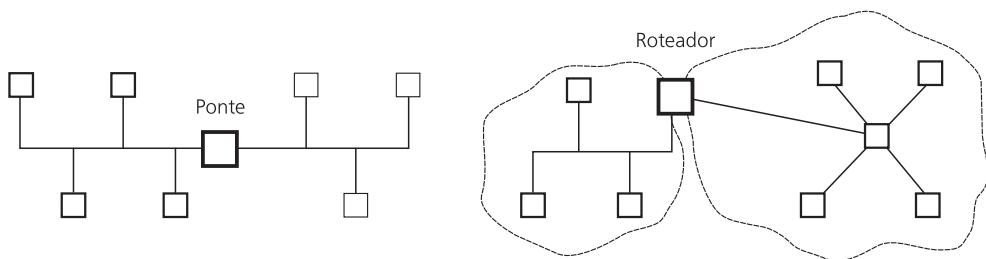
Figura 3.10 Topologias de redes.

à qual todas as outras são conectadas; (d) irregular, na qual as máquinas são conectadas de uma maneira arbitrária. A topologia irregular é comum em redes de longa distância, enquanto as em anel e em via em geral são empregadas em ambientes locais, cujas redes têm um único responsável.

É importante lembrar que a conexão entre as máquinas de uma rede não necessita ser física. As redes sem fio estão se tornando a cada dia mais comuns. Por exemplo, a tecnologia tradicional de transmissão de rádio, na qual o sinal se propaga em todas as direções, pode ser usada para implementar uma rede com topologia de via. Nesse caso, a via não é um cabo físico, mas um segmento de freqüências do espectro de transmissão.

Às vezes, torna-se necessário interligar duas redes existentes. Quando elas são compatíveis, isto pode ser feito meramente conectando as redes por meio de um dispositivo de acoplamento chamado **ponte** (*bridge*). Por exemplo, no caso de duas redes com topologia de via, é possível usar uma ponte para conectar as duas vias existentes de tal forma que os sinais de uma sejam retransmitidos na outra (Figura 3.11a). O importante é que quando duas redes estão conectadas por uma ponte, o resultado é simplesmente uma única rede maior.

Contudo, muitas vezes é necessário unir duas redes cujas características não são compatíveis. Por exemplo, as características de uma rede em estrela não são compatíveis com uma em via. Nesses



a. Uma ponte conecta duas redes com topologia de via para formar uma rede maior.

b. Um roteador conecta uma rede com tecnologia de via com outra com topologia em estrela para formar uma *internet* que consiste em duas redes.

Figura 3.11 A distinção entre uma ponte e um roteador.

casos, as duas redes devem ser conectadas de uma maneira que construa uma rede de redes, que é conhecida como **internet**, ou seja, as redes originais mantêm a sua originalidade e continuam a funcionar como redes independentes. Elas são meramente ligadas de forma que as mensagens de uma sejam transferidas para a outra, permitindo assim que uma máquina de uma rede envie mensagem a uma máquina da outra.

A conexão entre duas redes para formar uma *internet* é feita por uma máquina chamada **roteador** (router), que é um computador pertencente às duas redes, que propaga as mensagens de uma para a outra (Figura 3.11b). Note que a tarefa de um roteador é significativamente maior do que a de uma ponte. De fato, o roteador deve fazer as conversões de idiossincrasias das duas redes originais.

A Internet

O exemplo mais notável de *internet* é a Internet (note que o I é maiúsculo), que se originou de uma pesquisa iniciada em 1973 pela Defense Advanced Research Project Agency (DARPA). A meta desse programa era desenvolver a capacidade de interconectar várias redes de computadores, de forma que eles funcionassem como uma única rede confiável. Hoje, a Internet é uma combinação de redes locais e redes de longa distância que envolve milhões de máquinas.

Topologia da Internet Conceitualmente, a Internet pode ser vista como um conjunto de **domínios**, em que cada um é constituído de uma rede ou uma pequena *internet*, operada por uma única entidade, como uma universidade, uma empresa, ou uma instituição governamental. Cada domínio é um sistema autônomo, que pode ser configurado conforme determinado pelas autoridades locais, talvez até mesmo na forma de um conjunto de redes de longa distância, de âmbito mundial. Para estabelecer um domínio, a entidade deve registrá-lo no ICANN (Internet Corporation of Assigned Names and Numbers — pronuncia-se Ai-can), uma corporação sem fins lucrativos para coordenar a identificação de domínios, como veremos a seguir.

Uma vez registrado, ele pode ser anexado à Internet por meio de um roteador que conecte uma das redes do domínio a uma rede da Internet. Esse roteador é considerado o **portão** (gateway) do domínio, já que representa a porta do domínio ao mundo exterior. O “mundo exterior” às vezes é chamado **nuvem** (cloud), em referência ao fato de que a estrutura da Internet fora do portão do domínio está fora de seu controle e é irrelevante à sua operação. Qualquer mensagem transmitida a um destino dentro do domínio é tratada dentro dele; qualquer mensagem transmitida a um destino fora do domínio é direcionada para o portão, de onde é enviada para a nuvem.

Se alguém “ficar” no portão e “olhar” a nuvem, poderá identificar diversas estruturas. Sem dúvida, a Internet tem crescido de uma maneira imprevisível, uma vez que vários domínios encontram pontos nos quais se conectam à nuvem. Uma estrutura popular, contudo, reúne os portões de diversos domínios para formar uma rede regional de portões. Por exemplo, um grupo de universidades pode decidir juntar seus recursos para construir esse tipo de rede. Essa rede regional, por sua vez, pode ser conectada a uma rede mais global a que outras redes regionais se interliguem. Dessa maneira, a porção da nuvem assume uma estrutura hierárquica (Figura 3.12).

Indivíduos que desejam acessar a Internet podem registrar, implementar e manter os seus próprios domínios. Contudo, é mais comum para um indivíduo ter acesso à Internet por meio de um domínio estabelecido pela organização à qual ele pertence, ou contratando um provedor de serviços (Internet Services Provider — ISP) para se conectar ao domínio estabelecido pelo provedor. Na maioria dos casos, a sua conexão ao provedor é temporária, feita através de linha telefônica.

Endereçamento na Internet À cada máquina na Internet é atribuído um único endereço chamado **endereço IP**, usado para identificá-la nas comunicações. Cada endereço IP é um padrão de 32 bits que consiste em duas partes: a identificação do domínio ao qual a máquina pertence e uma identificação da máquina dentro desse domínio. A parte do endereço que identifica o domínio, o **identificador da rede**, é atribuído pelo ICANN quando o domínio é estabelecido e registrado. Assim, é por meio desse processo de registro que se garante que cada domínio na Internet possui um único identificador na rede. A parte do endereço que identifica uma máquina particular dentro do domínio é chamada **endereço do hospedeiro** (o termo **hospedeiro** se refere a uma máquina da rede, em reconhecimento ao papel de hospedar as solicitações de outras máquinas). O endereço do hospedeiro é atribuído pela autoridade local do domínio — geralmente uma pessoa cuja função é identificada como administrador de rede ou administrador de sistema. Por exemplo, o identificador da rede da companhia de publicações Addison Wesley é 192.207.177 (os identificadores de redes tradicionalmente são escritos em notação decimal com pontos; veja o Exercício 8 no final da Seção 1.4). No entanto, uma máquina dentro deste domínio terá um endereço como 192.207.177.133, cujo último byte é o endereço do hospedeiro.

Endereços no formato de padrões de bits são difíceis para os seres humanos memorizarem. Por isso, a cada domínio é atribuído um endereço mnemônico, conhecido como **nome do domínio**. Por exemplo, o nome do domínio de Addison Wesley é aw.com. Essa classificação é chamada **domínio de alto nível** (*top-level domain* — TLD). Existem muitos TLDs, incluindo edu para instituições de educação, gov para instituições governamentais, org para organizações sem fins lucrativos, museum para museus, info para uso irrestrito e net, que foi originalmente dedicado aos provedores, mas hoje é usado em larga escala. Além desses TLDs gerais, existem TLDs de duas letras para os países (chamados TLD de código do país) como au para Austrália e ca para Canadá.

Uma vez que um domínio tenha um nome mnemônico, sua autoridade local poderá estendê-lo para obter nomes mnemônicos para as máquinas do domínio. Por exemplo, uma máquina individual dentro do domínio aw.com pode ser identificada como ssenterprise.aw.com.

Devemos enfatizar que a notação decimal com pontos, usada nos endereços mnemônicos, nada tem a ver com a usada para representar endereços no formato de padrões de bits. Em vez disso, as seções de um endereço mnemônico

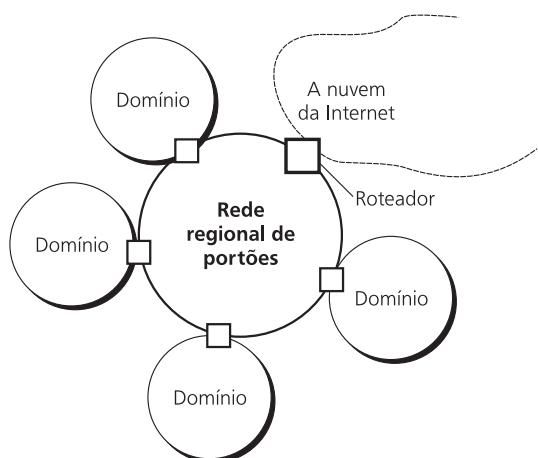


Figura 3.12 Uma abordagem típica de conexão à Internet.

identificam a localização da máquina dentro de um sistema de classificação hierárquica. No nosso exemplo, o endereço `ssenterprise.aw.com` indica uma máquina conhecida como `ssenterprise` dentro da instituição `aw` que, por sua vez, está dentro da classe (ou TLD) de instituições comerciais `.com`. No caso de domínios grandes, uma autoridade local pode dividir seu domínio em subdomínios, e os endereços mnemônicos serão mais longos. Por exemplo, suponha que a Nowhere University tenha como nome do domínio `nowhereu.edu` e resolva dividir seu domínio em subdomínios. Então, uma máquina da Nowhere University poderia ter um endereço como: `r2d2.compsc.nowhereu.edu`, o que significa que a máquina `r2d2` está no subdomínio `compsc`, que, por sua vez, está no domínio `nowhereu` dentro da classe de domínios educacionais `.edu`.

A autoridade local de cada domínio é responsável pela manutenção de um diretório, que contém os endereços mnemônicos e seus endereços numéricos IP correspondentes para as máquinas dentro do domínio. Este diretório é implementado por um computador interno ao domínio, na forma de um servidor, chamado **servidor de nomes**, que fornece a informação dos endereços. Juntos, todos os servidores de nomes que compõem a Internet formam um amplo diretório, distribuído por toda a Internet, usado para converter endereços mnemônicos para os endereços numéricos correspondentes. Se alguém desejar enviar uma mensagem para um destinatário identificado em forma mnemônica, este sistema de servidores de nomes converterá este endereço mnemônico para o seu endereço correspondente no padrão de bits compatível com o software de Internet. Normalmente, isso é feito em uma fração de segundo.

Correio eletrônico Para poder trocar mensagens entre usuários individuais da Internet (um sistema conhecido como **e-mail**, abreviatura de *electronic mail*, ou seja, correio eletrônico), a autoridade local designa uma máquina do domínio para manipular as atividades de correio eletrônico do mesmo. Esta máquina é conhecida como **servidor de correio** do domínio. Cada mensagem de *e-mail* enviada de dentro do domínio é inicialmente dirigida ao servidor de correio, que então a envia ao destinatário. De maneira semelhante, toda mensagem de *e-mail* endereçada a uma pessoa do domínio é recebida pelo servidor de correio do domínio, onde é mantida até que a pessoa solicite a leitura de suas novas mensagens.

Com o papel do servidor de correio em mente, é fácil entender a estrutura de um endereço de correspondência individual. Ele consiste em uma cadeia de símbolos que identificam a pessoa, seguida do sinal `@`, seguido do nome mnemônico do servidor de correio que deve receber a correspondência. Assim, o endereço eletrônico de um indivíduo na Addison Wesley tem o seguinte aspecto: `shakespeare@mailroom.aw.com`. Isso significa que a máquina conhecida como `mailroom` no domínio `aw.com` é o servidor de correio que manipula os *e-mails* da pessoa `shakespeare`. É prática comum os domínios serem projetados de modo a que o nome do servidor de correio não precise aparecer no endereço de *e-mail*. Nesse caso, o endereço de *e-mail* consiste meramente em uma cadeia de símbolos que identificam o receptor seguida do sinal `@`, seguido pelo nome mnemônico do domínio. Assim, o endereço acima seria reduzido para `shakespeare@aw.com`.

A World Wide Web Além de ser um meio de comunicação via correio eletrônico, a Internet tornou-se também um meio de divulgação de documentos multimídia, conhecidos como **hipertextos**, os quais contêm unidades de texto, imagens, som e vídeo e podem estar ligados a outros documentos. (O termo **hipermídia** às vezes é usado em reconhecimento ao fato de que o hipertexto se expandiu para incluir outros formatos que não o texto.). O leitor de hipertexto pode fazer acessos a outros documentos associados simplesmente posicionando o *mouse* e apertando o seu botão. Por exemplo, suponhamos que a sentença “Foi excelente o desempenho da orquestra, interpretando o ‘*Bolero*’, de Maurice Ravel” apareça em um hipertexto, e que o nome *Maurice Ravel* esteja conectado a outros documentos — que talvez fornecam informações sobre o compositor. O leitor poderá examinar esse material associado simplesmente apontando o nome *Maurice Ravel* e apertando o botão do *mouse*. Além disso, se as ligações apropriadas estiverem instaladas, o leitor poderá ouvir uma gravação de áudio do concerto, selecionando o nome *Bolero*.

Desta maneira, o leitor dos hipertextos pode examinar documentos inter-relacionados, ou seguir uma sequência de raciocínios, de um documento para outro. Como as partes dos vários documentos

podem estar vinculadas a outros documentos, forma-se uma teia de informações. Quando implementados em rede de computadores, os documentos podem residir em várias máquinas, formando uma ampla teia. De modo semelhante, a teia que se desenvolveu na Internet engloba o mundo inteiro e é conhecida como **World Wide Web** (abreviada por Web). Um documento hipertexto na Web é comumente chamado de **página da Web**. Uma coleção de páginas fortemente relacionadas (normalmente armazenada em uma única localidade) é chamada **sítio da Web** (*Web site*).

Os pacotes de *software* que ajudam os leitores de hipertextos a percorrer os entrelaçamentos dos mesmos ou são programas que operam como clientes ou, então, programas que operaram como servidores. O cliente reside no computador do leitor e é encarregado de obter materiais solicitados pelo usuário e apresentá-los de uma maneira organizada. É o cliente que fornece ao usuário a interface necessária para percorrer a teia. Por isso, o cliente freqüentemente é chamado navegador (*browser* — aquele que folheia). O servidor de hipertexto reside no computador que contém documentos a serem consultados. Sua tarefa é permitir o acesso a esses documentos, conforme a solicitação dos clientes. Em resumo, o usuário obtém acesso aos hipertextos por meio de um navegador existente na sua máquina, o qual atende às solicitações do leitor, requisitando os serviços fornecidos pelos servidores de hipertextos espalhados pela Internet.

Para localizar e recuperar documentos na Web, cada documento recebe um único endereço chamado **Localizador Uniforme de Recursos** (*Uniform Resource Locator* — URL). Cada URL contém a informação necessária para um navegador para contatar o servidor apropriado e solicitar o documento desejado. Um URL comum está ilustrado na Figura 3.13. Às vezes, um URL pode não identificar especificamente o documento, consistindo apenas no protocolo e no endereço que certamente descreve a informação disponível nessa máquina. Esse URL menor fornece os meios de contatar uma organização. Por exemplo, o URL <http://www.aw.com> levará à *home page* da Addison Wesley, que contém ligações com numerosos documentos relacionados à companhia e a seus produtos.

Um hipertexto é semelhante a um documento textual tradicional, em que o texto é representado caractere por caractere, usando um sistema de codificação como o ASCII ou o Unicode. A diferença é que o hipertexto também contém símbolos especiais, chamados marcadores, que descrevem como o documento deve aparecer na tela do computador e que itens do documento estão ligados a outros documentos. Esse sistema é conhecido como **Hypertext Markup Language** — **HTML** (linguagem de marcação de hipertextos). Assim, através do HTML, um autor de uma página da Web descreve toda a informação que um navegador necessita para apresentar a página na tela do usuário e para encontrar qualquer documento relacionado com a página corrente.

Ethernet

Ethernet é um conjunto de padrões que implementam uma rede local com topologia de via. Seu nome derivou do projeto da Ethernet original, no qual as máquinas eram conectadas por um cabo coaxial denominado Ether. Desenvolvida originalmente nos anos 1970 e agora padronizada pela IEEE como parte da família de padrões IEEE-802, a Ethernet é o método mais comum de ligar computadores pessoais em rede. De fato, os controladores Ethernet para PC são disponíveis e fáceis de instalar.

Atualmente, existem várias versões de Ethernet, que refletem os avanços na tecnologia e nas taxas maiores de transferência. No entanto, todos mantêm os traços comuns que caracterizam a família Ethernet. Dentre eles, estão o formato em que os dados são compactados para ser transmitidos, o uso da codificação Manchester (um método para representar 0s e 1s, no qual o 0 é representado por um sinal descendente e o 1, por um ascendente) para efetuar a transmissão dos bits e o uso da CSMA/CD para controlar o direito de transmissão.

A versão HTML de uma página extremamente simples é mostrada na Figura 3.14. Ela consiste em duas seções — um cabeçalho e um corpo. O cabeçalho contém a informação preliminar semelhante à encontrada no início de um memorando interno, a data e o assunto do memorando. O corpo contém o material a ser apresentado na tela do usuário. Nesse caso, a página consiste meramente na frase: “Minha Página Web”, exibida prominentemente na tela. (O texto “Minha Página Web” é marcado como cabeçalho de nível um, indicado pela marcação h1.) Aprenderemos mais sobre a HTML quando discutirmos arquivos-texto no Capítulo 8.

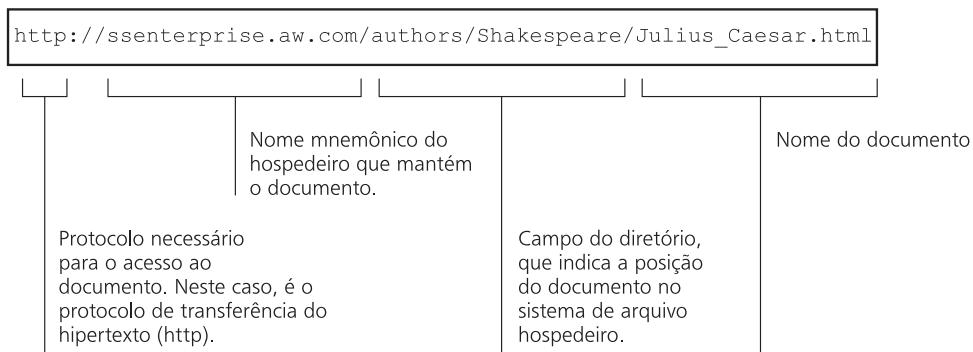


Figura 3.13 Um URL (localizador uniforme de recursos) comum.

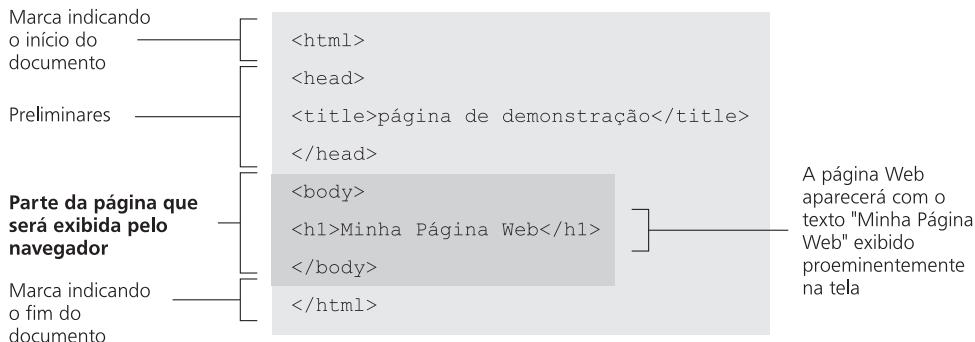
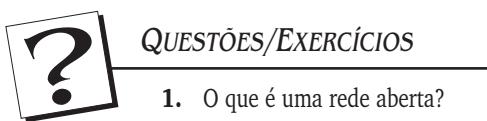


Figura 3.14 Uma página da Web expressa em HTML.



1. O que é uma rede aberta?
2. O que é um roteador?
3. Quais são os componentes de um endereço completo de uma máquina na Internet?
4. O que é um URL? E um navegador?
5. Em uma rede local com topologia em anel, qual a desvantagem de restringir a uma única direção a transferência das mensagens?

3.6 Protocolos de redes

Os conjuntos de regras que administram a comunicação entre os diferentes componentes de um sistema computacional são denominados **protocolos**, em alusão aos protocolos usados na sociedade para administrar as relações humanas. Por meio dos protocolos da rede, é se definem os detalhes de

cada atividade, incluindo o modo como são enviadas as mensagens, a maneira como a autorização para transmitir mensagens é delegada às máquinas e a forma como são manipuladas as tarefas de compactar e descompactar mensagens para a transmissão. Consideremos primeiramente os protocolos que controlam a autorização de uma máquina para transmitir suas próprias mensagens pela rede.

Controle dos privilégios de transmissão

Um método para coordenar as autorizações de transmissão de mensagens é o **protocolo token-ring** (anel de símbolos) desenvolvido pela IBM em 1970 e que continua sendo um protocolo popular para redes com topologia em anel. Nele, cada máquina só transmite mensagens à sua vizinha à direita e só recebe mensagens da vizinha à esquerda, conforme ilustrado na Figura 3.15. A mensagem de uma máquina para outra deve ser enviada, dentro da rede, sempre no sentido horário, até que alcance o seu destino. Quando ela atinge o seu objetivo, a máquina receptora mantém uma cópia da mensagem e envia outra para percorrer o anel. Quando a cópia enviada atingir a máquina que a originou, esta interpretará que a mensagem alcançou o seu destino, devendo portanto ser retirada do anel. Naturalmente, este sistema depende da cooperação entre essas máquinas. Se cada uma insistir em transmitir constantemente apenas suas mensagens em vez de também enviar as das outras, nenhum trabalho útil será realizado.

Para resolver esse problema, se passa através do anel de comunicação um padrão de bits, chamado **símbolo**. As máquinas que apresentarem este mesmo símbolo recebem a autorização para transmitir suas próprias mensagens; sem ele, a máquina fica autorizada apenas a retransmitir as mensagens que recebe. Normalmente, cada máquina apenas passa o símbolo da esquerda para a direita, fazendo o mesmo com as mensagens. Entretanto, se ao receber o símbolo a máquina tiver suas próprias mensagens para serem enviadas à rede, ela as transmitirá, mas reterá o símbolo. Quando as mensagens tiverem completado o seu ciclo ao redor do anel, a máquina devolverá o símbolo para a próxima do anel. Do mesmo modo, quando a próxima máquina receber o símbolo, poderá devolvê-lo imediatamente ou transmitir primeiro a sua própria mensagem, antes de devolver o símbolo para a máquina seguinte. Desta maneira, cada máquina da rede tem igual oportunidade de transmitir suas próprias mensagens enquanto o símbolo permanecer circulando no anel.

Outro protocolo para coordenar a autorização de transmissão foi criado pela Ethernet, que é uma versão popular de uma rede com topologia de via. Na Ethernet, a autorização para transmitir mensagens é controlada por um protocolo conhecido como **CSMA/CD** (*Carrier Sense, Multiple Access with Collision Detection*), o qual estabelece que toda mensagem transmitida por uma máquina deve ser retransmitida para todas as máquinas da via (Figura 3.16). Cada uma pode monitorar todas as mensagens, mas retém apenas aquelas que lhe estiverem endereçadas. Para transmitir uma mensagem, a máquina espera até detectar silêncio na via, quando começará a transmitir, enquanto continua monitorando a via. Se outra máquina também começar a transmitir, ambas detectarão o sinal e aguardarão durante um breve período aleatório, antes de tentar novamente a transmissão. O resultado é um sistema semelhante ao usado por um pequeno grupo de pessoas em uma conversação. Se duas pessoas começam a falar ao mesmo tempo, ambas param. A diferença é que elas podem pedir desculpas, como: “Desculpe, o que você ia dizer?”, ou “Nada, nada. Por favor, fale primeiro!”, enquanto com o protocolo CSMA/CD cada máquina simplesmente tenta de novo.

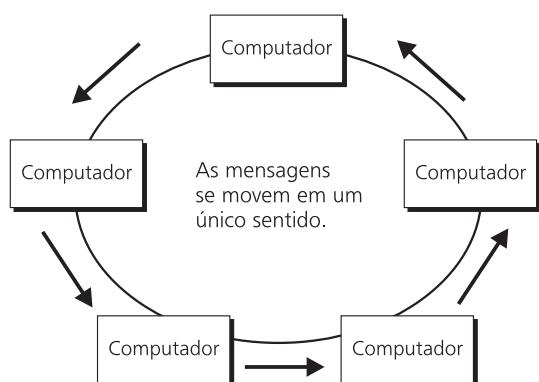


Figura 3.15 Comunicação com topologia em anel.

A abordagem em camadas para o software da Internet

A principal tarefa de um *software* de rede é prover a infra-estrutura necessária para transmitir mensagens de uma máquina para a outra. Na Internet, essa atividade de troca de mensagens é realizada por meio de uma hierarquia de unidades de *software* que executam tarefas similares às que seriam feitas se você fosse mandar um presente em um pacote a um amigo que reside na costa leste dos EUA, supondo que você esteja na costa oeste (Figura 3.17). Mais especificamente, você iria embrulhar o presente e escrever o endereço apropriado no lado de fora do pacote. Então, o levaria a uma empresa prestadora de serviços de entrega, como o correio. Esta poderia colocar o pacote juntamente com outros em um contêiner maior e entregá-lo a uma companhia aérea cujos serviços foram previamente contratados. Esta colocaria o contêiner em um avião e o levaria até a cidade destino, talvez com paradas intermediárias no decorrer da viagem. No destino final, a companhia retiraria o contêiner do avião e o levaria a um escritório da entregadora. Por sua vez, esta retiraria o pacote do contêiner e o entregaria a seu amigo.

Em resumo, o transporte do presente é realizado por uma hierarquia de três níveis: (1) o nível do usuário (referente a você e seu amigo), (2) a companhia entregadora e (3) a companhia aérea. Cada nível utiliza o nível inferior mais próximo como uma ferramenta abstrata. (Você não se preocupa com os detalhes da companhia entregadora, e esta não se preocupa com as operações internas da companhia aérea.) Em cada nível de hierarquia, podem ser identificados remetentes e destinatários, sendo que os destinatários exercem um papel oposto ao dos seus parceiros remetentes.

É o que ocorre com o *software* que controla a comunicação na Internet, exceto pelo fato de haver quatro níveis, e por cada um destes níveis, chamado camada, ser formado por um conjunto de rotinas

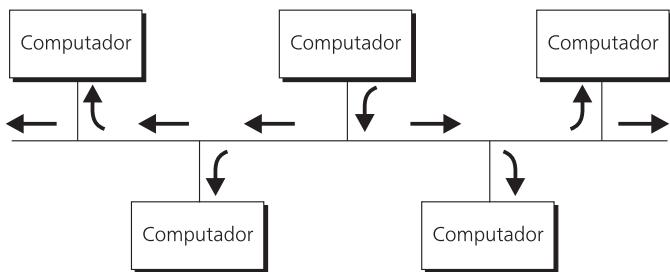


Figura 3.16 Comunicação em uma topologia de via.

Javascript, Applets, CGI e Java Servlets

Convém ao usuário da World Wide Web ter um entendimento básico das ações que acontecem quando uma página é acessada. Muitas páginas Web atualmente são acompanhadas por animação e som. Essas ações geralmente são controladas por unidades de programa executadas no computador do usuário. O servidor Web transfere suas unidades de programa à máquina do usuário juntamente com o documento HTML que define a página Web. Duas linguagens populares para desenvolver tais unidades de programa são o Javascript (desenvolvida pela Netscape Communications, Inc) e o Java (desenvolvido pela Sun Microsystems). No caso do Java, as unidades de programa são chamadas *applets*.

Por outro lado, há casos em que, quando o usuário está acessando uma página, ele pode causar a execução de um programa na máquina que contém o servidor que forneceu a página. Por exemplo, o usuário pode fazer um pedido de compra ou iniciar uma pesquisa em um sítio de busca. (Um sítio de busca contém um banco de dados de ligações para páginas da Internet.) Existem duas técnicas populares para implementar essas ações iniciadas pelo usuário. Uma é para o navegador do usuário iniciar a execução do programa seguindo um conjunto de protocolos conhecido como CGI (Common Gateway Interface). A outra é para o navegador do usuário iniciar a execução de um programa Java conhecido como Servlet.

Em síntese, o acesso a páginas Web pode resultar na execução de programas na máquina do usuário e na do servidor. Nos dois casos, a execução do programa em uma máquina é iniciada pela outra. Por sua vez, o uso de tais características representa uma possível ameaça à segurança. A máquina do usuário deve assegurar que as unidades de programa que acompanham as páginas da Web não realizem ações maliciosas, e a máquina do servidor, que os clientes em visita às páginas da Web não poderão ter acesso generalizado à máquina, maior do que o pretendido.

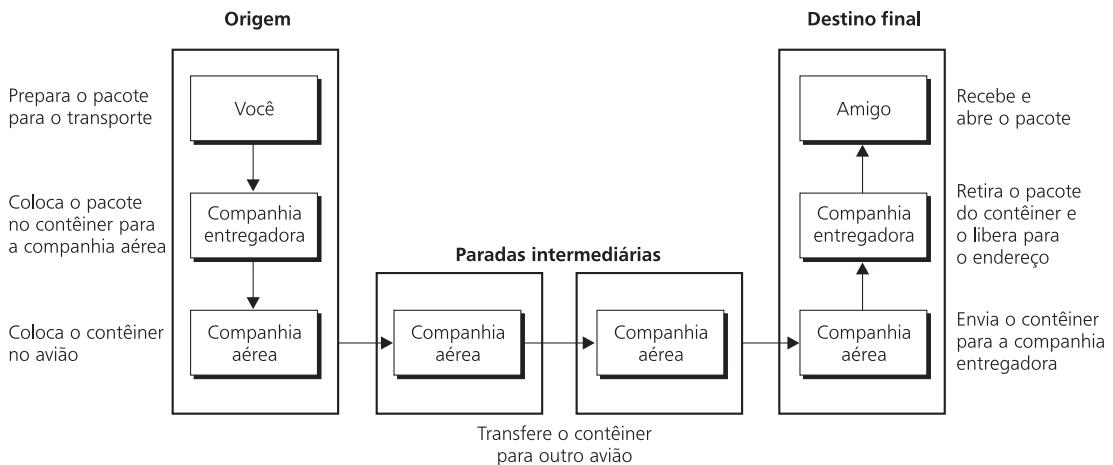


Figura 3.17 Exemplo de transporte de pacote.

de *software*, em vez de pessoas e negócios. As quatro camadas são conhecidas como de **aplicação, transporte, rede e ligação** (Figura 3.18). Todas elas estão presentes em cada máquina da Internet. Uma mensagem geralmente se origina na camada de aplicação. De lá, é passada para as camadas de transporte e de rede, onde é preparada para a transmissão, e finalmente é transmitida pela camada de ligação. A mensagem é recebida pela camada de ligação no destinatário e passa de volta pela hierarquia até ser entregue à camada de aplicação do destinatário.

Vamos investigar esse processo mais profundamente, rastreando a mensagem em seu caminho no sistema (Figura 3.19). Começaremos nossa jornada na camada de aplicação.

A camada de aplicação consiste em unidades de *software* que necessitam da comunicação pela Internet para desempenhar as suas tarefas. Embora os nomes sejam similares, essa camada não se responde ao *software* classificado como de aplicação apresentado na Seção 3.2. De fato, muitas unidades

consideradas parte da camada de aplicação da Internet seriam consideradas *software* utilitário, de acordo com o esquema de classificação da Seção 3.2. Um exemplo é uma coleção de rotinas para a transferência de arquivos pela Internet que use um protocolo conhecido como **Protocolo de Transferência de Arquivos** (File Transfer Protocol — FTP). Essas rotinas são comumente implementadas como um pacote completo de *software* conhecido como FTP, em referência ao protocolo subjacente. Outro exemplo é o pacote *telnet*, desenvolvido como meio para permitir que uma pessoa tenha acesso a uma máquina na rede com se ela fosse a usuária local da máquina. FTP e *telnet* foram pensados originalmente como *software* de aplicação de acordo com o critério da Seção 3.2, mas atualmente fazem parte da infra-estrutura encontrada na maioria dos computadores pessoais. De fato, essas unidades de *software* agora são usadas como ferramentas abstratas na construção de aplicações maiores, como os navegadores da Web. Nesse sentido, elas se

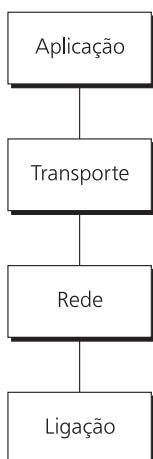


Figura 3.18 As camadas de *software* da Internet.

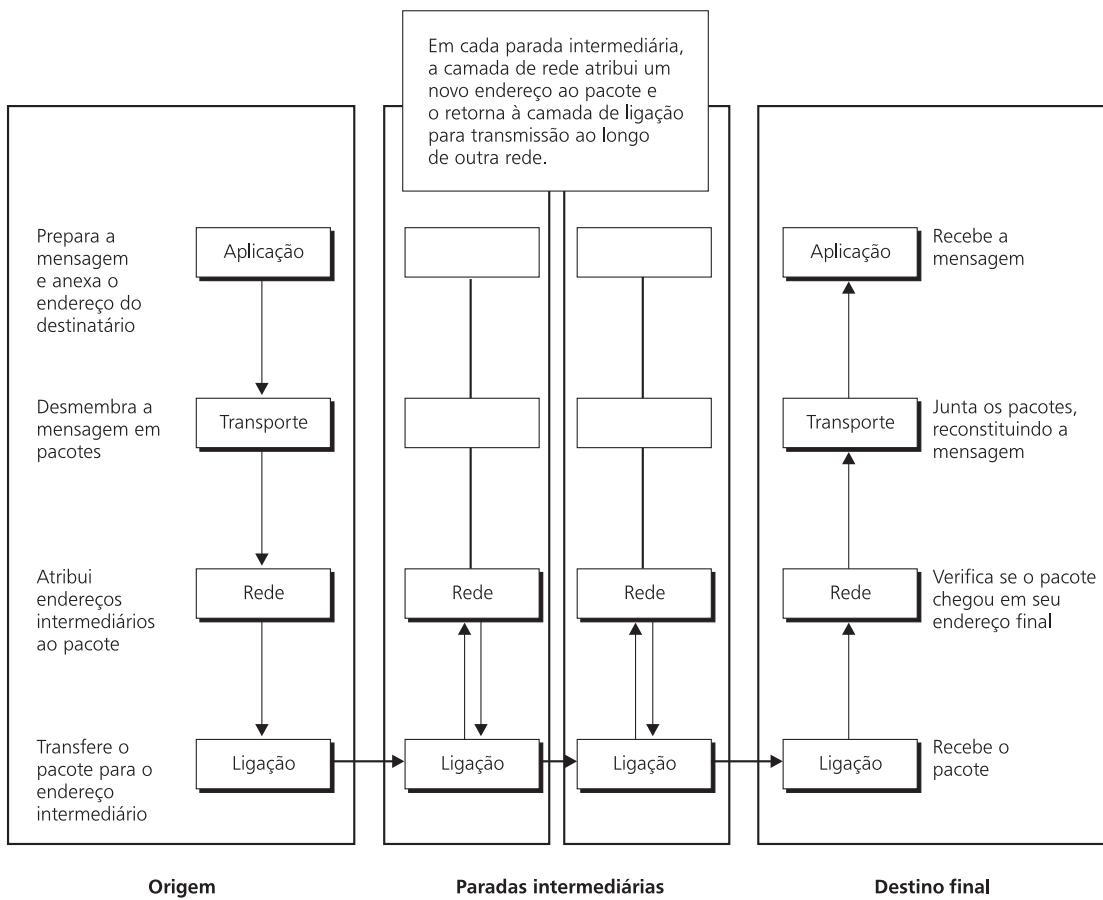


Figura 3.19 Acompanhamento de uma mensagem pela Internet.

tornaram *software* utilitário. Outro exemplo desse fenômeno é a coleção de rotinas de *software* que implementam o **Protocolo Simples de Transferência de Correio** (*Simple Mail Transfer Protocol* — SMTP), que atualmente é um pacote utilitário usado pelos servidores de correio quando transferem correspondência eletrônica.

A camada de aplicação usa a de transporte para enviar e receber mensagens pela Internet de um modo muito parecido à maneira como você usaria uma companhia entregadora para enviar e receber pacotes. Assim como é sua responsabilidade fornecer um endereço compatível com as especificações da companhia entregadora, é responsabilidade da camada de aplicação fornecer um endereço compatível com a camada de transporte. Com vistas a cumprir esta exigência, é que a camada de aplicação solicita ao servidor de nomes da Internet que converta endereços mnemônicos usados por pessoas em endereços IP compatíveis com a Internet.

A principal tarefa da camada de transporte é aceitar as mensagens da camada de aplicação e assegurar que elas estejam formatadas adequadamente à transmissão pela Internet. Para isso, a camada de transporte divide as mensagens longas em pequenos segmentos, que são transmitidos como unidades individuais. Essa divisão é necessária porque uma única mensagem longa pode obstruir o fluxo de outras mensagens nos pontos da Internet onde numerosas mensagens cruzam seus caminhos. Sem dúvida, pe-

queenos segmentos de mensagem podem se entrelaçar nesses pontos, enquanto uma mensagem longa força os outros a esperar até que ela passe (similar aos carros que esperam um longo trem passar pelo cruzamento).

A camada de transporte adiciona números de sequência a esses pequenos segmentos que ela produz, de modo que eles podem ser reagrupados no destino da mensagem. Então, anexa o endereço de destino a cada segmento e entrega esses segmentos endereçados, conhecidos como **pacotes** (*packets*) à camada de rede. A partir desse ponto, os pacotes são tratados individualmente, sem relação com as mensagens, até que atinjam a camada de transporte do destino. É possível que os pacotes de uma mensagem sigam caminhos diferentes ao longo da Internet.

A camada de rede é responsável por verificar se os pacotes recebidos são repassados de uma rede da Internet para outra até que atinjam seu destino. Assim, é a camada de rede que deve lidar com a topologia da Internet. Se o caminho de um pacote específico através da Internet deve passar por muitas redes individuais, é a camada de rede em cada parada intermediária que determina a direção na qual o pacote dever ser enviado. Isso é feito acrescentando-se um endereço destinatário intermediário a cada pacote, de acordo com as seguintes normas: “Se o destino final do pacote estiver dentro da rede corrente, o endereço anexado será uma duplicata do endereço final do destinatário; caso contrário, o endereço anexado será o do roteador da rede corrente, por onde o pacote deve passar para ser transferido para uma rede adjacente.” Deste modo, o pacote destinado a uma máquina dentro da rede corrente será enviado a ela, e o pacote destinado a uma máquina externa a ela continuará a sua jornada rede a rede.

Uma vez determinado o destino intermediário do pacote, a camada de rede anexa esse endereço ao pacote e o entrega à camada de ligação.

A responsabilidade da camada de ligação é a de lidar com os detalhes da comunicação com a rede específica em que a máquina se encontra. Se esta rede for do tipo anel de símbolos, a camada de ligação terá de aguardar o recebimento do símbolo antes de realizar sua transmissão. Se a rede usar um protocolo CSMA/CD, a camada de ligação terá de esperar por um silêncio na via antes de começar a transmitir. Além disso, cada rede individual dentro da Internet possui o seu próprio sistema de endereços, que é independente do sistema usado pela Internet. Afinal de contas, muitas dessas redes já funcionavam por si próprias bem antes de se associarem à Internet. Assim, a camada de ligação terá de traduzir os endereços da Internet anexados aos pacotes para um sistema local apropriado de endereços.

Cada vez que um pacote é transmitido, ele é recebido pela camada de ligação da máquina receptora. É então enviado à camada de rede, onde o destinatário final é comparado com a localização corrente. Se não coincidirem, a camada de rede anexará um novo endereço intermediário ao pacote e o devolverá à camada de ligação para retransmissão. Dessa maneira, cada pacote salta de uma máquina para outra a seu modo, até o seu destino final. Note que apenas as camadas de ligação e de rede estão envolvidas nas paradas intermediárias (veja novamente a Figura 3.19).

Se a camada de rede constatar que um pacote recebido alcançou seu destino final, ela o enviará à camada de transporte. Ao receber os pacotes da camada de rede, a de transporte extrai os segmentos e reconstitui a mensagem original de acordo com os números de sequência que foram fornecidos pela camada de transporte na máquina de origem. Uma vez que a mensagem esteja completa, será enviada à camada de aplicação — completando assim o processo de transmissão de mensagens.

A determinação de qual aplicação deve receber uma nova mensagem é uma tarefa importante da camada de transporte. Isto é feito atribuindo um número único de porta (nada a ver com as portas de E/S discutidas no Capítulo 2) às várias unidades de aplicação e exigindo que a aplicação que envia a mensagem coloque o número apropriado da porta no endereço da mensagem antes que esta inicie a sua jornada. Assim, quando a mensagem tiver sido recebida pela camada de transporte do destino, esta necessitará apenas enviar a mensagem ao software de aplicação instalado no número designado da porta.

Os usuários da Internet raramente precisam se preocupar com números de porta, uma vez que as aplicações comuns possuem números aceitos universalmente. Por exemplo, se um navegador Web for solicitado recuperar o documento cujo URL é <http://www.zoo.org/animals/frog.html> ele saberá que o contato com o servidor HTTP em www.zoo.org é feito pela porta de número 80. De modo

semelhante, quando transfere um arquivo, um cliente FTP sabe que deve procurar o servidor FTP na porta de número 21.

Em resumo, as comunicações na Internet envolvem a interação de quatro camadas de *software*. A camada de aplicação lida com mensagens e endereços do ponto de vista da aplicação. A de transporte converte essas mensagens em pacotes compatíveis com a Internet e reconstitui as mensagens recebidas antes de entregá-las à aplicação apropriada. A camada de rede lida com o direcionamento dos pacotes através da Internet. A de ligação trata da transmissão propriamente dita dos pacotes de uma máquina para outra em uma mesma rede. Com toda essa atividade, é espantoso que o tempo de resposta da Internet seja medido em milissegundos. De fato, a maioria das transações parece ocorrer instantaneamente.

O protocolo TCP/IP

A demanda por redes abertas gerou uma necessidade de padronização, por meio da qual os fabricantes pudessem construir equipamentos e softwares compatíveis com os produtos de outros fabricantes. Um desses padrões é o modelo de referência da Open System Interconnection (OSI), produzido pela International Organization for Standardization — ISO. Este padrão se baseia em uma hierarquia formada por sete níveis, em vez dos quatro adotados pela Internet. Tornou-se um modelo muito citado, pois conta com o apoio de uma organização internacional, porém não foi prontamente implementado, pois, antes de ser concluído, o protocolo TCP/IP já tinha sido desenvolvido, implementado, amplamente divulgado e tido a sua confiabilidade comprovada, tornando-se o sistema de protocolos da Internet.

O pacote TCP/IP é um conjunto de protocolos que implementa a hierarquia de quatro níveis descrita. De fato, os protocolos **TCP (Transmission Control Protocol)** e **IP (Internet Protocol)** são os nomes de apenas dois dos protocolos dessa coleção. Assim, o fato de todo o conjunto ser denominado protocolo TCP/IP é, no mínimo, impróprio. Mais precisamente, o protocolo TCP define uma versão da camada de transporte. Dizemos uma versão porque o protocolo TCP/IP apresenta dois modos de implementar a camada de transporte, sendo o segundo definido pelo protocolo **UDP (User Datagram Protocol)**. Isso se assemelha ao fato de você poder escolher uma determinada companhia entregadora, dentre as várias disponíveis, sendo que cada uma oferece o mesmo serviço básico, porém com suas características próprias. Assim, dependendo da qualidade de serviço almejada, o software da camada de aplicação pode optar pelo envio dos dados ou pela versão TCP ou UDP da camada de transporte (Figura 3.20).

Há duas diferenças básicas entre o TCP e o UDP. A primeira é que, antes de enviar uma mensagem solicitada pela camada de aplicação, a camada de transporte baseada em TCP envia uma mensagem à camada de transporte do destinatário, informando que uma mensagem está para ser enviada. Espera então que esta seja reconhecida, antes de enviar a mensagem da camada de aplicação. Desta forma, pode-se dizer que a camada de transporte do TCP estabelece uma conexão antes de enviar a mensagem. A camada de transporte baseada no UDP não estabelece tal conexão. Ela apenas envia a mensagem ao endereço dado, sem ao menos saber se a máquina destinatária se encontra operante. Por isso, o UDP é considerado um protocolo sem conexões.

A segunda diferença básica entre os protocolos TCP e UDP é que a camada de transporte do TCP trabalha, tanto na origem como no destino, por meio de reconhecimento e retransmissão de segmentos, para certificar-se de que todos os segmentos da mensagem tenham sido transferidos com sucesso aos seus destinos. Por isso, o TCP é reconhecido como um protocolo confiável, enquanto o UDP, que não oferece tal serviço de retransmissão, é considerado não-confiável. Isso não significa que o UDP não seja uma boa escolha. Na verdade, uma camada

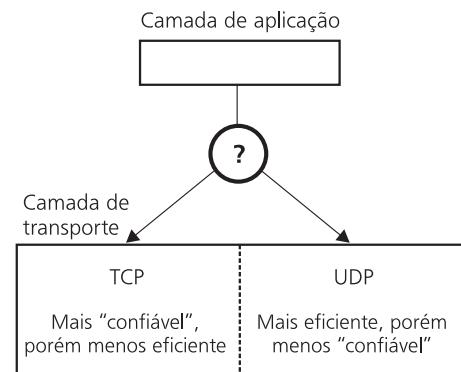


Figura 3.20 A escolha entre TCP e UDP.

POP3 versus IMAP

Os usuários da Internet que obtêm serviços de correio eletrônico via conexão telefônica discada devem ter ouvido falar e até talvez devem ter tido de escolher entre

POP3 e IMAP. Esses são os protocolos normais pelos quais um usuário de um computador remoto (geralmente um computador de mesa ou portátil) pode ter acesso às mensagens recebidas por um servidor de correio e guardadas na caixa postal do usuário. POP3 é abreviação de Post Office Protocol – Version 3 e é o mais simples dos dois. Com o POP3, o usuário pode transferir (“fazer download”) mensagens para seu computador pessoal,

onde elas podem ser lidas, armazenadas em várias pastas, editadas e manipuladas segundo as necessidades do usuário. Essa atividade acontece na máquina local do usuário, utilizando seu sistema de armazenamento em massa. Se, depois de ter lido e processado seu correio matinal em um computador, mais tarde o usuário em outra máquina fizer contato com o servidor de correio, tudo o que estará disponível para ele se resume a uma lista de mensagens em sua caixa postal. IMAP, abreviação de Internet Mail Access Protocol, permite ao usuário criar pastas e organizar mensagens na mesma máquina do servidor de correio. Dessa maneira, um usuário que precise ter acesso a sua correspondência de diferentes computadores pode manter os registros de correio acessíveis a qualquer computador remoto que venha a utilizar. Assim, o IMAP exige um serviço de mais alto nível do provedor da Internet que mantém o servidor de correio, que por sua vez pode cobrar uma taxa mais alta para serviços de IMAP do que de POP3.

de transporte baseada em UDP é mais aerodinâmica do que uma baseada em TCP, porém se uma aplicação for preparada para manipular as possíveis consequências do uso do UDP, esta opção poderá ser a melhor. Por exemplo, o correio eletrônico normalmente é enviado usando TCP, mas as comunicações feitas entre os servidores de nomes quando traduzem endereços na forma mnemônica para a forma IP usam UDP.

O protocolo IP é o padrão da Internet para a sua camada de rede. Uma de suas características é que, toda vez que uma camada de rede de IP enviar um pacote à camada de ligação, ela anexará um contador de saltos (*hop count*), número indicativo do tempo limite de permanência estabelecido para este pacote na rede. Este valor indica o número máximo de vezes que esse pacote poderá ser retransmitido de um nó a outro, durante a sua tentativa de encontrar o seu caminho pela Internet. Toda vez que a camada de rede do protocolo IP remeter um pacote, ela diminuirá o contador correspondente em uma unidade. Com base neste esquema, a camada de rede pode proteger a Internet, evitando que pacotes possam circular eternamente dentro do sistema. Embora a Internet continue crescendo diariamente, um contador de saltos com valor inicial 64 se mostra mais do que suficiente para garantir que um pacote possa encontrar a sua trajetória através do labirinto das redes locais, de longa distância e dos roteadores existentes.



QUESTÕES/EXERCÍCIOS

- Quais as camadas da hierarquia do *software* da Internet usadas para retransmitir uma mensagem recém-chegada para outra máquina?
- Cite algumas diferenças entre uma camada de transporte baseada no protocolo TCP e outra baseada no UDP.
- De que forma o *software* da Internet assegura que as mensagens não fiquem sendo eternamente retransmitidas de uma máquina para outra?
- O que impede um computador hospedeiro da Internet de fazer cópias de todas as mensagens que passam por ele?

3.7 Segurança

Quando uma máquina é conectada a uma rede, ela se torna acessível a muitos usuários em potencial. Os problemas encontrados caem em duas categorias: acesso não-autorizado à informação e vandalismo. Uma abordagem para resolver o problema do acesso não-autorizado é o uso de senhas para con-

trolar o acesso à máquina e itens de dados particulares. Infelizmente, elas podem ser obtidas por diversos meios. Alguns usuários de computador simplesmente compartilham suas senhas com os amigos — uma prática de ética questionável. Em outros casos, as senhas são roubadas. Um meio de furtar senhas é explorar as falhas no sistema operacional da máquina para encontrar os registros de senhas. Outra forma é escrever um programa que simule o processo de *login** local, de maneira que os usuários, pensando que estão se comunicando com o sistema operacional, digitam suas senhas, que então são guardadas pelo programa. Ainda outro meio de se obter senhas é tentar as escolhas óbvias e ver se elas funcionam. Por exemplo, os usuários que temem esquecer suas senhas podem usar seus próprios nomes como senhas. Datas como a de aniversário também são escolhas comuns.

Em um esforço para impedir esse jogo de adivinhação de senhas, os sistemas operacionais podem ser projetados para denunciar qualquer avalanche de senhas incorretas. Muitos também indicam a data em que a última conexão foi realizada, cada vez que uma nova sessão é iniciada. Isto permite ao usuário detectar qualquer uso não-autorizado de sua conta. Uma defesa mais sofisticada contra os adivinhadores de senha é criar a ilusão de sucesso (chamada *trapdoor*) quando senhas falsas são fornecidas e prosseguir dando informações sem importância ao intruso enquanto se tenta localizar a sua origem.

Outra abordagem para proteger os dados de acesso não-autorizado é criptografá-los; a idéia é que mesmo se eles forem obtidos por um intruso, a informação permanecerá segura. Para isso, diversas técnicas de criptografia foram desenvolvidas, sendo a mais popular na Internet a **criptografia de chave pública**. Ela permite que muitas pessoas enviem mensagens com segurança a um receptor central. A criptografia de chave pública envolve o uso de dois valores chamados chaves. Uma delas, conhecida como chave pública, é usada para a codificação de mensagens e é conhecida por todos autorizados a gerar mensagens; a outra, conhecida como chave privada, é necessária para decodificar as mensagens e conhecida apenas pela pessoa que deve receber as mensagens. O conhecimento da chave pública não permite a decodificação de mensagens. Assim, haverá poucos danos se ela cair em mãos desautorizadas. Ela dá à pessoa não-autorizada a habilidade de gerar mensagens, mas não a de decodificar as mensagens interceptadas. O conhecimento da chave privada é obviamente mais poderoso, mas ela é inherentemente mais segura do que a chave pública, pois é mantida por apenas uma pessoa. Investigaremos um sistema particular de criptografia de chave pública no Capítulo 11.

Além disso, existem muitas questões legais quanto ao acesso não-autorizado à informação, e algumas delas lidam com a distinção entre parte autorizada e não-autorizada. Por exemplo, um empregador está autorizado a monitorar as comunicações de seus funcionários? Até que ponto um provedor da Inter-

Camada segura de encaixes

Suponha que você esteja fornecendo seu número do cartão de crédito ao comprar uma mercadoria pela World Wide Web. Como você pode se assegurar de que o sítio que você contatou de fato é o da empresa correta? Suponha que uma firma de corretagem receba uma solicitação para vender o estoque de seu cliente. Como saber também se a solicitação veio do cliente e não de um impostor? A camada segura de encaixes (Secure Sockets Layer – SSL) é um sistema de protocolos que usa técnicas de chave pública para resolver estas questões. Ela também permite que as mensagens entre clientes e servidores sejam criptografadas. Para se comunicar via SSL, cliente e servidor devem se registrar previamente em um terceiro sítio, chamado autoridade certificadora. O cliente então contata o servidor usando o protocolo SSL, que faz com que cliente e servidor contatem a autoridade certificadora para confirmar a autenticidade de cada um antes de qualquer comunicação. Durante esse processo, eles também obtêm as chaves de criptografia necessárias para codificar a sua comunicação.

A maioria dos navegadores Web suporta SSL de forma transparente ao usuário. Uma vez que o usuário está registrado em uma autoridade certificadora, o SSL é ativado automaticamente quando ele contata o servidor, com um URL iniciado por https em vez de http do protocolo tradicional de comunicação por hipertexto.

*N. de T. Contração de *Logical In* — processo que inicia a sessão de um usuário em uma rede.

net tem acesso autorizado à informação que está sendo comunicada aos seus clientes? Até que ponto um provedor é responsável pelo conteúdo da comunicação entre seus clientes? Tais questões vêm desafiando a comunidade jurídica atual.

Nos Estados Unidos, muitas dessas questões são remetidas ao Electronic Communication Privacy Act (ECPA) de 1986, o qual se originou na legislação para controlar a escuta telefônica. Embora a lei seja extensa, seu escopo é capturado em alguns pequenos trechos. Em particular, ela declara que:

Exceto nos casos especificamente mencionados neste capítulo, qualquer pessoa que intencionalmente interceptar, procurar interceptar, ou que mandar outra pessoa interceptar ou procurar interceptar qualquer comunicação oral, por fio ou eletrônica... deve ser punida conforme a subseção (4) ou sujeita à ação conforme a subseção (5)

e

... qualquer pessoa ou entidade que forneça serviço de comunicação eletrônica ao público não deve intencionalmente divulgar o conteúdo de qualquer comunicação nesse serviço a qualquer pessoa ou entidade que não o endereçado ou receptor de tal comunicação ou o agente do endereçado ou do receptor.

Em resumo, o ECPA confirma o direito individual à comunicação privada — é ilegal a pessoa desautorizada se meter em comunicação alheia, e é ilegal um provedor da Internet liberar a informação que diga respeito à comunicação entre seus clientes. Contudo, a lei também declara o seguinte:

É permitido a um funcionário, empregado ou agente da Comissão Federal de Comunicações em seu trabalho normal, na conclusão do monitoramento responsável exercido pela Comissão para a garantia do Capítulo 5 do título 47 da Constituição, interceptar uma comunicação oral, por fio ou eletrônica ou comunicação oral transmitida por rádio, ou abrir e usar a informação obtida.

Assim, o ECPA explicitamente dá à Comissão Federal de Comunicações o direito de monitorar a comunicação eletrônica sob certas circunstâncias. Isso leva a alguns tópicos complicados. Primeiro, para que a Comissão possa exercer o seu direito garantido pelo ECPA, os sistemas de comunicação devem ser construídos e programados de forma que a comunicação possa ser monitorada. O estabelecimento dessa capacidade é considerado no Communications Assistance for Law Enforcement Act — CALEA. Ele exige que as companhias de telecomunicações modifiquem os seus equipamentos para se ajustarem à lei. Contudo, a implementação desse ato mostrou-se complexa e cara, resultando em um adiamento do prazo para a sua efetivação.

Um tópico ainda mais controverso envolve o choque entre o direito da Comissão de monitorar as comunicações e o direito do público de usar criptografia. Afinal de contas, se as mensagens monitoradas são bem criptografadas, então a intromissão na comunicação pelas agências legais de nada adianta. Os governos nos Estados Unidos, Canadá e Europa estão considerando sistemas que exigiriam o registro de chaves de criptografia (ou talvez chaves para as chaves). Entretanto, vivemos em um mundo onde a espionagem feita por uma corporação se tornou tão significativa quanto a militar. Assim, é compreensível que a exigência de registrar chaves de criptografia causaria desconforto a corporações e cidadãos idôneos. Até que ponto o sistema de registro é seguro?

O problema do vandalismo é exemplificado pela ocorrência de danos causados por vírus de computador e vermes de rede. Em geral, o **vírus** é um trecho de programa que se anexa a outros programas do sistema. Por exemplo, ele pode se inserir como prefixo de um programa do sistema, e assim, toda vez que esse programa hospedeiro for ativado, o vírus será automaticamente executado antes. Este vírus, por sua vez, pode praticar atos maliciosos prontamente constatáveis, ou apenas procurar outros programas, para neles instalar outras réplicas de si mesmo. Se um programa infectado for transferido para um novo computador, por meio da rede ou de disquetes, o vírus nele contido começará a infectar os programas existentes neste outro computador, assim que o programa recém-transferido for ativado. Desta maneira, o vírus se transmite de um computador para outro. Em alguns casos, os vírus são planejados inicialmente

para ser apenas propagados para outros programas, até que ocorra alguma condição predeterminada, como, por exemplo, a chegada de uma data especificada, para só então executar outros atos de vandalismo prejudiciais. Isto aumenta a probabilidade de o vírus se instalar em muitas máquinas, antes de ter sua presença detectada.

O termo **verme** (*worm*) normalmente se refere a um programa autônomo, que se transmite pela rede, instalando-se nas máquinas por onde passa e enviando outras cópias de si mesmo através da rede. Como no caso dos vírus, estes programas podem ser projetados apenas para se reproduzir, ou então para também executar atividades de vandalismo.

Uma abordagem que pode ser aplicada aos problemas de acesso não-autorizado e de vandalismo é a instalação de um *software* que filtra o tráfego que entra ou passa pela máquina. Por exemplo, um *software* de aplicação pode ser modificado para perscrutar todas as informações que chegam e registrar as mensagens que contenham certas palavras; a camada de transporte, para registrar todo o tráfego de ou para uma certa porta, ou a camada de rede pode ser modificada de maneira a filtrar todas as mensagens de certos endereços IP. Softwares desse tipo são chamados **firewall** (parede de fogo), no sentido em que formam uma barreira protetora que isola a região de um lado do perigo do outro lado. Freqüentemente, os *firewalls* são colocados no portão do domínio de tal forma que o domínio inteiro é protegido do perigo oculto na Internet. Em outros casos, são colocados nas máquinas individuais para formar um nível de proteção único a uma necessidade particular.

Com o crescimento da popularidade das redes, o risco de danos causados por acesso não-autorizado ou vandalismo também cresce. Falhas de segurança na Internet têm levado a prejuízos financeiros significativos, a riscos à segurança nacional e a numerosas questões éticas e legais. Se a Internet se tornará um ambiente de comunicação seguro, só vendo. O estado da arte atual poderia ser caracterizado pela recomendação: “usuário, tome cuidado”.

A equipe para responder às emergências da computação

Em novembro de 1988, um verme na Internet causou uma interrupção significativa dos serviços. Conseqüentemente, a Defense Advanced Research Projects Agency (DARPA) dos EUA formou o Computer Emergency Response Team – CERT, cujo Centro de Coordenação está localizado na Carnegie-Mellon University. O CERT é o “observador” de segurança da Internet. Dentre as suas atribuições, estão a investigação de problemas de segurança, a emissão de alertas de segurança e a implementação de campanhas de esclarecimento ao público para melhorar a segurança na Internet. O Centro de Coordenação do CERT mantém um sítio na Web no endereço <http://www.cert.org>, no qual divulga as suas atividades.



QUESTÕES/EXERCÍCIOS

1. Tecnicamente, o termo *dados* se refere à representação da informação, enquanto *informação* se refere ao significado subjacente. O uso de senhas protege os dados ou a informação? O uso de criptografia protege os dados ou a informação?
2. Quais são os pontos básicos do ECPA?
3. Explique como o CALEA demonstrou que a aplicação de leis que exigem ação pode não ser efetiva.

Problemas de revisão de capítulo

(Os problemas marcados com asteriscos se referem às seções opcionais.)

1. Cite quatro atividades de um sistema operacional comum.
2. Resuma a diferença entre processamento em lotes e processamento interativo.

3. Qual é a diferença entre processamento interativo e processamento em tempo real?
4. O que é um sistema operacional multitarefa?
5. Qual a informação contida em uma tabela de processos de um sistema operacional?
6. Qual a diferença entre um processo pronto e um processo em espera?
7. Qual a diferença entre memória virtual e memória principal em um computador?
8. Quais complicações poderão surgir em um sistema de tempo partilhado se dois processos solicitarem acesso concomitante ao mesmo arquivo? Há casos em que o gerente de arquivos deve atender ou rejeitar tal solicitação?
9. Defina carga equilibrada e escalação (*scaling*) no contexto das arquiteturas de multiprocessadores.
10. Resuma o processo de *booting* de um computador.
11. Suponha que um sistema operacional de tempo partilhado divida o tempo de processamento em intervalos de 50 milissegundos cada. Se normalmente forem gastos, em média, 8 milissegundos para posicionar o cabeçote de leitura/gravação de um disco na trilha desejada e outros 17 milissegundos para que os dados passem por este cabeçote, que fração do intervalo de tempo será gasta em espera, a cada operação de leitura de um disco? Se a máquina for capaz de executar dez instruções a cada microsegundo, quantas poderiam ser executadas durante este período de espera? (É por essa razão que um sistema com tempo partilhado normalmente permite que um processo seja executado, enquanto outro aguarda a finalização de serviços solicitados a um dispositivo periférico.)
12. Cite cinco recursos que o sistema operacional multitarefa se responsabiliza por coordenar.
13. Diz-se que um processo é limitado por entrada/saída* se solicitar muitas operações de transferência de dados, enquanto que processos que executam principalmente operações no processador e na memória do sistema são considerados limitados por processamento.** Se dois processos, um deles limitado por entrada/saída e o outro limitado por processamento, estiverem competindo por uma fatia de tempo para utilizar o processador, a qual deverá ser dada a prioridade? Por quê?
14. Em um sistema que executa dois processos em ambiente de tempo partilhado, pode-se obter maior vazão (*throughput*) se ambos forem limitados por entrada/saída (*I/O-bound*) (veja Problema 13) ou se um deles for limitado por entrada/saída e o outro, por processamento? Por quê?
15. Escreva um conjunto de diretrizes que estabeleça as ações que o despachante de um sistema operacional deve realizar quando a fatia de tempo de um processo termina.
16. Identifique os componentes do estado de um processo.
17. Identifique uma situação, em um sistema de tempo partilhado, na qual um processo não consome todo o intervalo de tempo a ele designado.
18. Cite, em ordem cronológica, os eventos mais importantes que ocorrem quando um processo é interrompido.
19. Responda às questões abaixo considerando o sistema operacional que você usa:
 - a. Como você pede ao sistema operacional para copiar um arquivo?
 - b. Como você pede ao sistema operacional para mostrar o diretório de um disco?
 - c. Como você pede ao sistema operacional para executar um programa?
20. Responda às questões abaixo considerando o sistema operacional que você usa:
 - a. Como o sistema operacional restringe o acesso apenas aos usuários autorizados?
 - b. Como você pede ao sistema operacional para mostrar o conteúdo corrente da tabela de processos?
 - c. Como você pede ao sistema operacional para não permitir que outros usuários da máquina tenham acesso a seus arquivos?
21. Descreva o modelo cliente-servidor.

*N. de T. Em inglês, *I/O bound*.

**N. de T. Em inglês, *compute-bound*.

- 22.** O que é CORBA?
- 23.** Identifique duas formas de classificação para as redes de computadores.
- 24.** Descreva os componentes do seguinte endereço eletrônico:
`kermit@frogs.animals.com`
- 25.** Defina os seguintes conceitos:
- Servidor de nomes
 - Domínio
 - Roteador
 - Hospedeiro
- 26.** Suponha que o endereço de um hospedeiro na Internet seja 134.48.4.123. Qual é o endereço de 32 bits na notação hexadecimal?
- 27.** Qual a diferença entre uma rede aberta e uma fechada?
- 28.** Defina os seguintes conceitos:
- Hipertexto
 - HTML
 - Navegador
- 29.** O que é a World Wide Web?
- 30.** Identifique os componentes do seguinte URL e descreva o significado de cada um:
`http://frogs.animals.com/animals/moviestars/kermit.html`
- 31.** No contexto de redes de computadores, qual a diferença entre um verme e um vírus?
- 32.** Quais são os tópicos referentes à segurança e à privacidade na Internet?
- 33.** Que são o ECPA e o CALEA?
- *34.** Explique um uso importante para a instrução *test-and-set*, encontrada em muitas linguagens de máquina. Por que é importante que toda a operação *test-and-set* seja implementada como uma única instrução?
- *35.** Um banqueiro, com \$100.000, empresta a dois clientes \$50.000 cada um. Mais tarde, estes dois clientes avisam que antes de reembolsar os seus empréstimos, cada um precisa de outro empréstimo de \$10.000 para completar a transação relativa aos empréstimos anteriores. O banqueiro soluciona este enlace mortal obtendo empréstimos de outra fonte e repassa este capital adicional (com um acréscimo na

- taxa de juros) aos dois clientes. Qual das três condições para enlace mortal foi removida pelo banqueiro?
- *36.** Estudantes que desejam se matricular em Modelagem Ferroviária II na universidade local necessitam da autorização do instrutor e devem pagar uma taxa pelo uso do laboratório. As duas exigências devem ser cumpridas, independentemente, em qualquer ordem e em diferentes locais do *campus*. O número de matrículas é limitado a 20 estudantes. Este limite é exigido tanto pelo instrutor, que concederá autorização de matrícula a apenas 20 estudantes, quanto pela tesouraria da universidade, que receberá a taxa de matrícula de somente 20 estudantes. Suponha que este sistema de inscrição tenha resultado, com sucesso, no registro de 19 estudantes para o curso, mas que a última vaga esteja sendo reivindicada simultaneamente por dois estudantes, um dos quais obteve somente a permissão do instrutor e o outro somente pagou a taxa. Qual condição de enlace mortal é removida em cada uma das seguintes propostas de solução para o problema?
- Os dois estudantes são admitidos no curso.
 - O tamanho de classe é reduzido para 19, assim, nenhum dos dois será inscrito no curso.
 - É negada a permissão para os dois estudantes que reivindicam a última vaga e um terceiro é designado para ocupá-la.
 - Decide-se que a única exigência para entrada no curso é o pagamento da taxa. Assim, o estudante que pagou ocupa a última vaga e a entrada é negada ao que obteve somente a permissão de instrutor.
- *37.** Explique como pode ocorrer um enlace mortal quando dois peões se encontram em um jogo de xadrez. Quais recursos não-compartilháveis estão envolvidos? Como este enlace mortal normalmente é eliminado?
- *38.** Suponha que cada recurso não-compartilhável de um sistema computacional seja classificado como de nível 1, nível 2 ou nível 3. Além disso, suponha que cada processo no sistema seja obrigado a solicitar os recursos de que necessite de acordo com esta classificação. Assim, terá de requisitar todos os recursos de nível 1 antes de solicitar os de nível 2. Assim que tiver recebido os

de nível 1, poderá requisitar todos os de nível 2, e assim por diante. Há possibilidade de ocorrer um enlace mortal neste sistema? Justifique sua resposta.

- *39. Cada braço de um robô está programado para retirar conjuntos montados sobre a esteira de uma linha de montagem, testá-los quanto às suas tolerâncias e depositá-los em uma de duas caixas, de acordo com os resultados desse teste. As peças chegam, uma de cada vez, guardando entre si um intervalo suficiente de tempo. Para impedir que os dois braços tentem agarrar uma mesma peça, os computadores que os controlam compartilham uma mesma posição de memória. Se um braço estiver disponível quando houver a aproximação de um conjunto, o computador de controle deste braço lerá o valor contido na posição de memória comum. Se este valor for diferente de zero, o braço deixará passar o conjunto. Caso contrário, o computador de controle irá colocar um valor diferente de zero nessa posição de memória, instruir o braço a apanhar o conjunto e recolocar o valor 0 nessa posição de memória após completar a ação. Que seqüência de eventos poderia conduzir a uma disputa entre os dois braços?
- *40. O seguinte problema “filósofos durante a janta”, foi originalmente proposto por E. W. Dijkstra e atualmente faz parte do folclore da Ciência da Computação.
Cinco filósofos estão sentados ao redor de uma mesa circular. À frente de cada um, está um prato de macarrão. Existem cinco garfos sobre a mesa, cada um entre dois pratos. Cada filósofo deseja alternar entre pensar e comer. Para comer, o filósofo requer a posse dos dois garfos que estão adjacentes a seu prato. Identifique a possibilidade de enlace mortal e de inanição presentes no problema do jantar dos filósofos.
- *41. Suponha que cada computador em uma rede em anel seja programado para transmitir simultaneamente, nas duas direções, as mensagens originadas naquele nó que estejam endereçadas a todos os outros nós pertencentes à rede. Além disso, suponha que isto seja feito obtendo-se inicialmente acesso à comunicação com a máquina situada à esquerda do nó, bloqueando-se o uso desse caminho enquanto o acesso à trajetória de comunicação com a máquina à direita do nó não for obtido, e para só então transmitir a mensagem.

Identifique o enlace mortal que ocorrerá se todas as máquinas da rede tentarem emitir tal mensagem simultaneamente.

- *42. Identifique o uso de uma fila no processo de *spooling* — envio de saídas — para uma impressora.
- *43. O cruzamento de duas ruas pode ser considerado como um recurso não-compartilhável, pelo qual competem os carros que dele se aproximam. Um semáforo é usado, em vez de um sistema operacional, para controlar a alocação desse recurso. Se o equipamento for capaz de quantificar o fluxo de tráfego que chega de cada direção e programado para sempre conceder sinal verde ao tráfego mais pesado, o tráfego mais leve poderia sofrer do que se chama inanição. Qual o significado desse termo? O que poderia acontecer em um sistema computacional multiusuário, no qual as rotinas são atendidas segundo a sua prioridade, se a competição pelos recursos fosse sempre resolvida estritamente com base na prioridade?
- *44. Quais problemas poderão ocorrer em um sistema de tempo partilhado se o despachante sempre designar intervalos de tempo de acordo com um sistema de prioridades, segundo o qual a prioridade de cada tarefa seja sempre a mesma? (Sugestão: de acordo com a relação entre a prioridade da rotina que acabou de utilizar o intervalo de tempo a que tinha direito, e a das rotinas que estão aguardando, qual deverá ser a rotina a receber o próximo intervalo de processamento?)
- *45. Qual a semelhança entre enlace mortal e inanição? (Consulte o Problema 44.) Qual a diferença entre esses dois conceitos?
- *46. Que problema surgirá se, em um sistema de tempo partilhado, a duração dos intervalos de tempo for sendo reduzida progressivamente? E se tais intervalos fossem aumentados gradativamente?
- *47. Com o desenvolvimento da Ciência da Computação, as linguagens de máquina têm sido estendidas para prover instruções especializadas. Três dessas instruções de máquina que foram introduzidas na Seção 3.4 são usadas extensivamente pelos sistemas operacionais. Quais são elas?

- *48. O que é o modelo de referência OSI?
- *49. Em uma rede baseada em topologia de via, esta é um recurso não-compartilhável cuja posse deve ser disputada pelas máquinas antes de transmitirem as suas mensagens. Como o enlace mortal é controlado nesse contexto?
- *50. Protocolos à base de *símbolos* também podem ser usados para controlar a permissão de transmissão em redes que não tenham topologia de anel. Projete um protocolo à base de *símbolos* para efetuar esse controle em uma rede local com topologia de via.
- *51. Descreva os passos seguidos por uma máquina que deseja transmitir uma mensagem em uma rede usando o protocolo CSMA/CD.
- *52. Enumere as quatro camadas da hierarquia de *software* da Internet e identifique a tarefa que cada uma executa.
- *53. Em que sentido o protocolo TCP poderia ser considerado superior ao UDP na implementação da camada de transporte? Em que situação tal avaliação se inverte?
- *54. O que significa dizer que UDP é um protocolo isento de conexões?

Questões sociais

As seguintes questões procuram auxiliar o leitor no entendimento de alguns assuntos éticos, sociais e legais no campo da computação. O objetivo não é meramente o de fornecer respostas a tais questões. O leitor também deve justificá-las e verificar se as justificativas apresentadas preservam sua consistência de uma questão para outra.

1. Suponha que você esteja usando um sistema operacional multiusuário, que lhe permita ver os nomes dos arquivos pertencentes aos outros usuários, bem como o seu conteúdo, desde que não estejam protegidos. Olhar tais informações sem autorização seria semelhante a entrar sem permissão em uma casa destrancada, ou seria mais parecido com a leitura de material afixado em um lugar público, como uma sala de espera de um consultório médico?
2. Quando você tem acesso a um sistema multiusuário, que responsabilidades assume ao escolher a sua senha?
3. A possibilidade de conectar computadores em rede tem popularizado o conceito de trabalho em casa. Quais são os pontos positivos e negativos desse movimento? Ele afetará o consumo de recursos naturais? Fortalecerá os laços familiares? Reduzirá a “política de escritório”? Os que trabalham em casa terão as mesmas oportunidades de ascensão profissional do que os outros? Os laços comunitários serão enfraquecidos? O contato pessoal reduzido entre os pares terá efeito positivo ou negativo?
4. A Internet está rapidamente se tornando uma alternativa à compra nas lojas. Que efeitos essa mudança de hábito de compras têm na comunidade? E nas ruas de comércio? E nas pequenas lojas, como livrarias e armazéns, onde gostamos de olhar as mercadorias sem comprá-las? Até que ponto a compra pelo menor preço possível é boa ou ruim? Existe alguma obrigação moral de pagar mais caro para incentivar a produção local? É ético comparar produtos em uma loja local e depois comprá-los a preço mais baixo via Internet? Quais são as consequências a longo prazo desse comportamento?
5. Até que ponto o governo deve controlar o acesso dos cidadãos à Internet (ou qualquer rede internacional) em nome da segurança nacional? Quais são as questões de segurança que podem ocorrer?
6. Boletins eletrônicos permitem que os usuários de redes de computadores enviem (em geral, anonimamente) mensagens e leiam mensagens enviadas por outros. O administrador desses boletins deve ser considerado responsável pelo seu conteúdo? Uma companhia telefônica pode ser considerada responsável pelo conteúdo das conversações telefônicas? O gerente de um supermercado

pode ser considerado responsável pelas informações contidas em um mural comunitário, afixado na loja?

7. O uso da Internet deveria ser monitorado? Deveria ser regulamentado? Nesse caso, por quem?
8. Quanto tempo você gasta na Internet? Esse tempo é bem utilizado? O acesso à Internet alterou as suas atividades sociais? Você acha mais fácil conversar via Internet do que pessoalmente?
9. Quando você compra um pacote de *software* para um computador pessoal, o desenvolvedor geralmente pede que você se registre de modo a ser notificado quando futuras versões estiverem disponíveis. Esse processo de registro tem sido feito cada vez mais através da Internet. Normalmente, você informa o seu nome, endereço e talvez como ficou conhecendo o produto; então o *software* automaticamente transfere esses dados ao desenvolvedor. Quais questões éticas poderão surgir se o programa de registro enviar informações adicionais ao desenvolvedor durante o processo de registro? Por exemplo, o *software* poderia vasculhar o seu sistema e informar que outros pacotes existem ali.
10. Quando você visita um sítio da Web, ele tem a capacidade de gravar dados, chamados *cookies*, em seu computador, indicando que você o visitou. Esses dados podem então ser utilizados para identificar o retorno de visitantes. Eles também podem guardar quais sítios visitados podem ser acessados por outros sítios. Um sítio Web deve ter essa capacidade? Deve ser permitido a um sítio Web gravar *cookies* no seu computador sem o seu conhecimento? Quais são os possíveis benefícios dos *cookies*? Que problemas podem surgir com o seu uso?
11. Uma lei contrária a um ato impede a sua ocorrência ou meramente o torna ilegal? Se as corporações forem obrigadas a registrar suas chaves de criptografia, elas ficarão seguras ou meramente protegidas pela lei? Qual é a diferença?
12. Em geral, a etiqueta leva a evitar que se ligue a um amigo em seu local de trabalho para tratar de assuntos pessoais ou sociais, tais como a combinação para sair em um fim de semana. De maneira semelhante, hesitamos em ligar para um cliente em sua casa para informar sobre um novo produto. Também enviamos convite de casamento para as residências e anúncios de simpósios de negócios para os endereços de trabalho. É adequado enviar *e-mails* pessoais a um amigo usando o servidor de correio de seu local de trabalho?

Leituras adicionais

- Comer, D. E. *Computer Networks and Internet*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2001.
Kurose, J. F. and K. W. Ross. *Computer Networking*. Reading, MA: Addison-Wesley Longman, 2001.
Nutt, G. *Operating Systems: A Modern Approach*, 2nd ed. Boston: Addison-Wesley, 2002.
Rosener, J. *CyberLaw, The Law of the Internet*. New York: Springer, 1997.
Shay, W. A. *Understanding Data Communications and Networks*. Boston: PWS, 1994.
Silberschatz, A., P. B. Galvin and G. Gagne *Operating System Concepts*, 6th ed. New York: Wiley, 2001.
Spinello, R. A. and H. T. Tavani. *Readings in CyberEthics*. Boston, MA: Jones and Bartlett, 2001.
Stevens, W. R. *TCP/IP Illustrated*, vol. 1. Reading, MA: Addison-Wesley, 1994.
Tanenbaum, A. S. *Modern Operating Systems*. 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2001.

4

C A P Í T U L O

Algoritmos

Vimos que, antes de um computador executar uma tarefa, deve-se fornecer-lhe um algoritmo que o instrua exatamente acerca do que deve ser feito; portanto, o estudo de algoritmos é a pedra angular da Ciéncia da Computação. Neste capítulo, apresentaremos os conceitos fundamentais deste estudo, incluindo tópicos acerca do descobrimento e da representação de algoritmos, bem como os principais conceitos ligados ao controle de iterações e à recursão. Assim procedendo, também estaremos apresentando alguns dos algoritmos de busca e ordenação mais conhecidos.

4.1 O conceito de algoritmo

Uma revisão informal
A definição formal de algoritmo
A natureza abstrata dos algoritmos

4.2 A representação de algoritmos

Primitivas
Pseudocódigo

4.3 Descoberta de algoritmos

A arte da resolução de problemas
Suba o primeiro degrau

4.4 Estruturas iterativas

Algoritmo de busca seqüencial
O controle do laço
O algoritmo de ordenação por inserção

4.5 Estruturas recursivas

Algoritmo de busca binária
Controle recursivo

4.6 Eficiéncia e correção

Eficiéncia de algoritmos
Verificação de software

4.1 O conceito de algoritmo

Na introdução, definimos informalmente algoritmo como um conjunto de passos que define a maneira como uma tarefa deve ser executada. Nesta seção, analisaremos mais de perto este conceito básico.

Uma revisão informal

Já encontramos muitos algoritmos em nosso estudo. Vimos algoritmos para converter representações de números de uma forma para outra, detectar e corrigir erros nos dados, compactar e descompactar arquivos de dados, controlar o compartilhamento de tempo em ambientes multitarefa e muitos mais. Além disso, vimos como eles podem ser expressos em uma linguagem de máquina e executados por uma máquina, cuja UCP desempenha as suas tarefas seguindo o algoritmo:

Enquanto a instrução de parada não for executada, continue a executar os seguintes passos:

- a. Busque a próxima instrução.
- b. Decodifique a instrução.
- c. Execute a instrução.

Como demonstrado pelo algoritmo que descreve um truque de mágica na Figura 0.1, os algoritmos não se restringem às atividades técnicas. Na verdade, eles estão subjacentes mesmo em atividades rotineiras, como descascar ervilhas:

Obtenha uma cesta de ervilhas com casca e uma tigela vazia.

Enquanto existirem ervilhas na cesta, continue a executar os seguintes passos:

- a. Pegue uma ervilha da cesta.
- b. Abra a casca da ervilha.
- c. Tire a ervilha da casca e coloque na tigela.
- d. Jogue fora a casca.

De fato, muitos pesquisadores acreditam que qualquer atividade da mente humana, inclusive imaginação, criatividade e tomada de decisão, é resultado da execução de algoritmo — conjectura que abordaremos novamente em nosso estudo de inteligência artificial (Capítulo 10).

Entretanto, antes de prosseguir, consideremos a definição formal de algoritmo.

A definição formal de algoritmo

Conceitos informais, fracamente definidos, são aceitáveis e comuns na vida diária, mas uma ciência deve se basear em terminologia bem-definida. Considere então a definição formal de algoritmo apresentada na Figura 4.1.

Note que a definição exige que o conjunto de passos em um algoritmo seja ordenado. Isso significa que os passos em um algoritmo devem possuir uma estrutura bem estabelecida em termos da ordem na qual são executados. Isso não implica que os passos sejam executados em seqüência consistam no primeiro passo, seguido do segundo e assim por diante. Alguns algoritmos, conhecidos como algoritmos paralelos, por exemplo, apresentam mais de uma sucessão de passos possível, cada qual projetada para ser executada por processadores diferentes em uma máquina com multiprocessamento. Em tais casos, o algoritmo como um todo não apresenta uma linha única de passos, em que possa ser identificada uma seqüência autêntica. Em vez disso, apresenta uma estrutura de várias ramificações e reconexões simultâneas de processamento, cada uma responsável pela execução de uma das diferentes partes do algoritmo. (Voltaremos a esse conceito na Seção 6 do Capítulo 5.) Outros exemplos incluem algoritmos executados por circuitos, como no caso do *flip-flop* da Seção 1.1, no qual cada porta lógica executa um único passo do algoritmo global. A ordem em que os passos são executados decorre de relações de causa e efeito, à medida que a saída de cada porta lógica se propaga pelo circuito.

Em seguida, considere a exigência de que um algoritmo deve consistir em passos executáveis. Para tanto, considere a instrução:

Construir uma lista de todos os inteiros positivos.

Seria impossível realizar essa instrução, já que existem infinitos números inteiros positivos. Assim, nenhum conjunto de instruções que inclua esta instrução seria um algoritmo. Os cientistas da computação usam o termo *efetivo* para indicar que algo é executável. Em outras palavras, dizer que um passo de um algoritmo é efetivo significa que ele pode ser, de alguma forma, realizado.

Outra exigência imposta pela definição da Figura 4.1 é que os passos de um algoritmo não sejam ambíguos. Em outras palavras, durante a execução de um algoritmo, a informação sobre o estado do processo deve ser suficiente para determinar unívoca e completamente as ações a serem tomadas em cada passo, ou seja, a execução de cada passo de um algoritmo não deve requerer criatividade, bastando apenas a capacidade de seguir instruções.

A definição na Figura 4.1 também exige que um algoritmo defina um processo que termine, isto é: a execução de um algoritmo deve sempre levar o processamento a um estado de término. A origem desta exigência está na Teoria da Computação, cuja meta é responder a questões como “Quais são os limites reais dos algoritmos e das máquinas?” Aqui, a Ciência da Computação procura distinções entre problemas cujas soluções podem ser obtidas algorítmicamente e problemas cujas soluções se situam além da capacidade de sistemas algorítmicos. No nosso contexto, é traçada uma fronteira entre os processos que culminam com uma resposta e os que simplesmente prosseguem indefinidamente sem produzir qualquer resultado.

Há, contudo, aplicações significativas aos processos que não terminam, tais como monitorar sinais vitais de pacientes em um hospital, ou manter constante a altitude de uma aeronave em vôo. Alguns argumentariam que estas aplicações envolvem apenas a repetição de algoritmos, em que cada um chegaria a seu próprio término e em seguida repetir-se-ia automaticamente. Outros diriam que tais argumentos são simplesmente tentativas forçadas de adesão a uma definição formal demasiadamente restritiva. Independentemente de quem esteja com a razão, o fato é que o termo algoritmo com freqüência é usado, em conotações informais, para referir-se a conjuntos de passos que não definem necessariamente processos com término. Um exemplo é o “algoritmo” de divisão que envolve números com vários dígitos, o qual não define um processo com término quando se divide 1 por 3. Tecnicamente, esses exemplos representam uso incorreto do termo.

A natureza abstrata dos algoritmos

É importante enfatizar a diferença entre um algoritmo e sua representação, que é análoga à diferença entre um conto e o livro a que pertence. O conto é, por natureza, abstrato, ou conceitual; o livro é uma representação física do conto. Se o livro for traduzido para outro idioma, ou se for reeditado em um formato diferente, apenas a representação do conto irá mudar, embora o conto propriamente dito permaneça o mesmo.

Da mesma forma, o algoritmo é abstrato e distinto de suas representações. Um único algoritmo pode ser representado de diversas formas. Por exemplo, um algoritmo que converte leituras de temperatura de graus Celsius para Fahrenheit é representado tradicionalmente pela fórmula algébrica

$$F = (9/5)C + 32$$

Entretanto, o mesmo algoritmo poderia ser representado pela instrução

Multiplicar a temperatura, lida em graus Celsius, por 9/5, e então somar 32 ao produto assim obtido.

Um algoritmo é uma série ordenada de passos não-ambíguos, executáveis,

Figura 4.1 A definição de algoritmo.

ou até mesmo na forma de um circuito eletrônico. Em todos esses casos, o algoritmo em questão é sempre o mesmo; só as suas representações é que diferem.

A distinção entre um algoritmo e sua representação apresenta um problema quando tentamos comunicá-lo. Um exemplo comum refere-se ao nível de detalhe no qual um algoritmo deve ser descrito. Para os meteorologistas, a instrução *converta a leitura da temperatura de graus Celsius para Fahrenheit* pode ser suficientemente clara, mas um leigo pode requerer uma explicação melhor, com a justificativa de que tal instrução é para ele ambígua. Note-se que o problema não é que o algoritmo subjacente seja ambíguo, mas que a sua representação não está suficientemente bem formulada para um leigo. Assim, neste caso, a ambigüidade está na representação, e não no próprio algoritmo. Na próxima seção, veremos como o conceito de operação primitiva pode ser utilizado para eliminar da representação dos algoritmos tais problemas de ambigüidade.

Finalmente, ainda tema de algoritmos e suas representações, também devemos esclarecer a diferença entre dois outros conceitos inter-relacionados — programas e processos. Um programa é uma representação de um algoritmo. De fato, cientistas da computação utilizam o termo *programa* para se referir a uma representação formal de um algoritmo, projetada para aplicação de computador. Definimos processo no Capítulo 3 como a atividade de executar um programa. Note-se, porém, que, se executar um programa corresponde a executar o algoritmo representado por ele, então o processo pode ser igualmente definido como a atividade de executar um algoritmo. Concluímos então que processos, algoritmos e programas são entidades distintas, embora relacionadas. Um programa é a representação de um algoritmo, enquanto um processo é a atividade de executar um algoritmo.



QUESTÕES/EXERCÍCIOS

1. Resuma a diferença entre os conceitos de processo, algoritmo e programa.
2. Dê alguns exemplos de algoritmos com os quais você está familiarizado. Eles são realmente algoritmos no sentido exato?
3. Identifique alguns pontos deixados vagos na nossa definição informal de algoritmo, apresentada na Seção 0.1.
4. Em que sentido os passos descritos na seguinte lista de instruções não constituem um algoritmo?
 - Passo 1. Tirar uma moeda do seu bolso e colocá-la sobre a mesa.
 - Passo 2. Retornar ao passo 1.

4.2 A representação de algoritmos

Nesta seção, trataremos de assuntos relativos à representação de um algoritmo. Nossa meta é introduzir os conceitos básicos de pseudocódigo e de operações primitivas (ou, simplesmente, primitivas), bem como estabelecer um sistema de representação para nosso uso.

Primitivas

A representação de um algoritmo requer alguma forma de linguagem. No caso dos seres humanos, esta pode ser a tradicional linguagem natural (inglês, russo, japonês) ou talvez a das figuras, conforme mostra a Figura 4.2, que descreve um algoritmo para construir um pássaro por meio de dobradura de papel. Em geral, estes meios naturais de comunicação conduzem a enganos, às vezes porque a terminologia utilizada apresenta mais de um significado. Em inglês, a frase *Visiting grandchildren** can be nerve-

*N. de T. Há duas traduções válidas para *Visiting grandchildren*: a primeira é *Visitar netos*, e a segunda, *Netos visitantes*.

racking tanto poderia significar que os netos causam problemas quando vêm fazer uma visita, como também poderia indicar que é problemático ir visitar os netos. Mal-entendidos também podem resultar da inadequação entre os níveis de detalhe exigido pelo leitor e o utilizado na formulação das frases. Poucos leitores sabem construir um pássaro com dobraduras de papel seguindo as instruções descritas na Figura 4.2, embora um estudante de origami o faça sem a menor dificuldade. Em suma, problemas de comunicação surgem quando a linguagem utilizada para a representação de um algoritmo não estiver definida com precisão, ou quando a informação não for adequadamente detalhada.

A Ciência da Computação trata estes problemas estabelecendo um conjunto bem-definido de elementos funcionais básicos com os quais podem ser construídas representações de algoritmos. Diz-se que cada bloco construtivo é uma operação primitiva, ou, simplesmente, **primitiva**. Ao definir rigorosamente estas primitivas, eliminam-se muitos problemas de ambigüidade, e ao exigir que algoritmos sejam descritos em termos destas primitivas estabelece-se um padrão uniforme para o nível de detalhe. Um conjunto de primitivas, juntamente com um de regras, estabelecendo de que maneira as primitivas podem ser combinadas para representar idéias mais complexas, constitui uma linguagem de programação.

Cada primitiva consiste em duas partes: sua sintaxe e sua semântica. *Sintaxe* refere-se à representação simbólica da primitiva, e *semântica*, ao conceito representado, ou seja, ao significado da primitiva. A sintaxe da palavra *ar* consiste em dois símbolos, enquanto a sua semântica é uma substância gasosa que envolve o mundo. Como exemplo, a Figura 4.3 apresenta algumas das primitivas usadas em origami.

Para se obter uma coleção de primitivas para uso na representação de algoritmos para serem executados em computadores, poderíamos retornar às instruções básicas, projetadas para serem executadas independentemente pela máquina. Se um algoritmo for expresso neste nível de detalhe, certamente teremos um programa expresso de forma adequada para ser executado pelo computador. Entretanto, é tedioso expressar algoritmos neste nível, por isso, geralmente se utiliza um conjunto de primitivas de nível mais alto, cada qual formando com seus elementos uma ferramenta abstrata, construída a partir das primitivas de baixo nível, provenientes da linguagem de máquina. O resultado é uma linguagem de programação formal, na qual os algoritmos podem ser expressos em uma forma conceitualmente mais alta do que na linguagem básica de máquina. Discutiremos tais linguagens de programação no próximo capítulo.

Representação de algoritmos durante seu projeto

A tarefa de projetar um algoritmo complexo exige que o projetista se mantenha a par de numerosos conceitos inter-relacionados – exigência que pode exceder as capacidades da mente humana. (Em um artigo na *Psychological Review* em 1956, George A. Miller relatou pesquisas que indicavam que a mente humana é capaz de manipular em torno de sete detalhes por vez.) Assim, o projetista de algoritmos complexos necessita de uma maneira de registrar e retomar parte de um algoritmo em desenvolvimento de acordo com as exigências de sua concentração.

Durante as décadas de 1950 e 1960, os fluxogramas (em que os algoritmos são representados por formas geométricas ligadas por setas) representavam o estado da arte em termos de ferramentas de projeto. Entretanto, os fluxogramas freqüentemente se tornam teias de setas que se cruzam, fazendo com que a estrutura subjacente do algoritmo seja no mínimo difícil de se perceber. Assim, o uso de fluxograma como ferramenta de projeto vem dando lugar a outras técnicas de representação. Um exemplo é o pseudocódigo usado neste texto, pelo qual os algoritmos são representados por estruturas bem definidas de texto. Contudo, os fluxogramas ainda são benéficos quando o objetivo é a apresentação, em vez de projeto. Por exemplo, as Figuras 4.8 e 4.9 aplicam a notação de fluxograma para demonstrar a estrutura algorítmica representada por instruções de controle populares.

A busca por melhores notações de projeto é um processo contínuo. No Capítulo 6, veremos que a tendência é usar técnicas gráficas para ajudar no projeto global de grandes sistemas de software, enquanto o pseudocódigo permanece popular para o projeto de componentes funcionais dentro de um sistema.

Concluindo, a representação de algoritmos é uma tarefa que requer cuidado e atenção. É importante entender as diferenças entre a sintaxe e a semântica das primitivas, e como elas podem ser combinadas para representar idéias mais complexas. Além disso, é necessário ter em mente que a representação de algoritmos deve ser clara e precisa, de modo que possa ser entendida por outros profissionais da área. Isso pode exigir a utilização de ferramentas de projeto específicas, como fluxogramas ou pseudocódigo, dependendo do contexto e do nível de detalhe desejado.

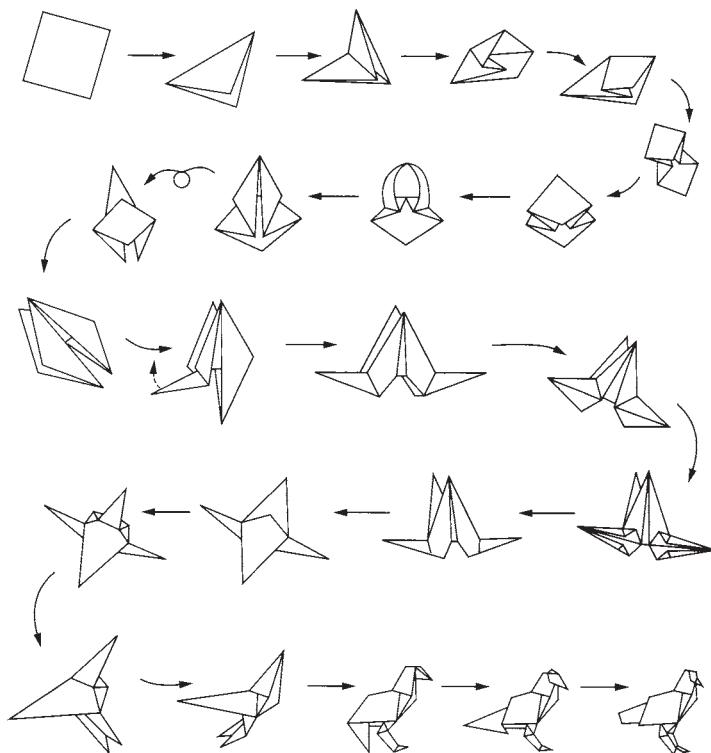


Figura 4.2 Construção de um pássaro pela dobradura de um pedaço quadrado de papel.

Pseudocódigo

Por ora, vamos adiar a apresentação de uma linguagem formal de programação, em favor de um sistema notacional menos formal e mais intuitivo, conhecido como pseudocódigo. Em geral, um **pseudocódigo** é um sistema notacional no qual as idéias podem ser informalmente expressas durante o processo de desenvolvimento do algoritmo.

Uma forma de obtenção de um pseudocódigo consiste em simplesmente relaxar as regras da linguagem formal na qual a versão final do algoritmo deverá ser expressa. Esta prática é adotada sempre que a linguagem de programação a ser usada para a codificação for conhecida de antemão. Neste caso, o pseudocódigo, utilizado durante os estágios iniciais do desenvolvimento do programa, compõe-se de estruturas sintático-semânticas semelhantes às empregadas pela linguagem de programação adotada, embora menos formais.

Nomenclatura de itens dos programas

Em uma linguagem natural, os itens freqüentemente possuem nome com múltiplas palavras, tais como “custo de produção de um recurso” ou “tempo estimado de chegada”. Contudo, quando expressamos algoritmos em um programa formal ou em uma versão do programa em pseudocódigo, os nomes com múltiplas palavras podem complicar a descrição de um algoritmo. A experiência tem mostrado que é melhor ter cada item identificado por um único bloco de texto.

Ao longo dos anos, muitas técnicas foram usadas para comprimir múltiplas palavras em uma única unidade léxica, obtendo nomes descritivos para itens nos programas. Uma delas é usar o símbolo de sublinhado para conectar palavras, produzindo nomes como `tempo_estimado_de_chegada`. Outra é usar letras maiúsculas para auxiliar um leitor a compreender um nome composto de múltiplas palavras. Por exemplo, pode-se começar cada palavra com uma letra maiúscula, obtendo nomes como `TempoEstimadoChegada`. Essa técnica freqüentemente é chamada notação Pascal, porque foi popularizada pelos usuários da linguagem de programação Pascal. Uma variação da notação Pascal é a notação Camelô, idêntica à Pascal, exceto em que a primeira letra permanece minúscula, como em `tempoEstimadoChegada`. Este texto segue a notação Pascal, mas a escolha é questão de gosto pessoal.

Nossa meta é tratar o desenvolvimento e a representação de algoritmos, mas sem limitar a discussão a alguma linguagem de programação em especial. Assim, nossa tática será a de desenvolver uma notação consistente e concisa para representar as estruturas semânticas mais frequentes, as quais, por sua vez, se tornarão as primitivas com que procuraremos expressar nossas idéias futuras. Uma tal estrutura semântica é a atribuição de valores a nomes descritivos. Por exemplo, queremos nos referir a valores usando nomes como Temperatura, PrecipitaçãoTotal, BalançoBancário e Altitude. Para estabelecer a associação entre nomes e valores, usamos a forma

Nome ← expressão

onde *nome* é o nome descritivo e *expressão* descreve o valor a ser associado ao nome. Leremos esta instrução como “atribuir ao *nome* o valor da *expressão*”. Por exemplo, a instrução

Fundos ← SaldoCC + Poupança

atribui o resultado da adição de SaldoCC com Poupança ao nome Fundos.

Outra estrutura semântica comum é a necessidade de selecionar uma de duas atividades possíveis dependendo da veracidade ou falsidade de alguma condição. Pode-se citar, por exemplo:

Se o produto interno bruto tiver aumentado, compre ações ordinárias; caso contrário, venda ações ordinárias.

Compre ações ordinárias se o produto interno bruto tiver aumentado; caso contrário, venda-as.

Compre ou venda ações ordinárias, dependendo do crescimento ou queda, respectivamente, do produto interno bruto.

Estas instruções podem ser reescritas de modo a inserir-se na estrutura:

se (condição) então (atividade)
senão (atividade)

Sintaxe	Semântica
	Girar o papel por exemplo
Um lado do papel sombreado	Distingue diferentes lados do papel por exemplo
	Representa uma dobradura em vale de modo que representa
	Representa uma dobradura em montanha de modo que representa
	Dobrar para cima de modo que produz
	Empurrar para dentro de modo que produz

Figura 4.3 Primitivas de origami.

na qual utilizamos as palavras-chave **se**, **então** e **senão** para anunciar o início das diversas partes da estrutura principal e parênteses para delimitar essas subestruturas. Adotando esta estrutura sintática para nosso pseudocódigo, obtém-se uma forma padronizada e uniforme de expressão para essa estrutura semântica freqüente.

Não obstante a instrução

Dividir o total por 366 ou 365, conforme o ano seja bissexto ou não, respectivamente.

apresente um estilo literário mais criativo, utilizaremos a forma mais objetiva

```
se(ano bissexto)
  então (dividir ← total por 366)
  senão (dividir ← total por 365)
```

Também adotamos a sintaxe abreviada:

```
se (condição) então (ação)
```

para casos que não envolvam a escolha entre duas atividades. Usando esta notação, a instrução

Se acontecer de as vendas diminuírem, reduza em 5% o preço.

será reduzido para

```
se (vendas diminuíram) então (reduzir em 5% o preço)
```

Outra estrutura semântica muito encontrada envolve a necessidade de continuar executando repetidamente uma instrução ou uma seqüência de instruções enquanto uma certa condição permanecer verdadeira. Exemplos informais incluem:

Havendo ingressos para vender, continuar vendendo ingressos.

e

Enquanto houver ingressos para vender, permanecer vendendo os ingressos.

Para tais casos, adotamos o seguinte padrão uniforme para o nosso pseudocódigo:

```
enquanto (condição) faça (ação)
```

em suma, tal instrução indica que se deseja verificar o valor da *condição*, e, se ele for verdadeiro, executar a *ação*, voltando novamente a conferir a *condição*. Quando constatar-se que seu valor é falso, passar-se-á a executar a próxima instrução. Assim, as duas instruções anteriormente ilustradas são reduzidas a:

```
enquanto (ainda há ingressos para vender) faça (vender um ingresso)
```

A prática da endentação* freqüentemente facilita a leitura de um programa. Por exemplo, a declaração

```
se (produto está sujeito a imposto)
  então (se (preço > limite)
    então (pagar x)
    senão (pagar y)
  )
  senão (pagar z)
```

*N. de T. Em inglês, *indentation*. Designa uma disciplina de tabulação do texto de um programa, de tal forma que a sua estrutura fique evidenciada, pelo posicionamento, na mesma vertical, do início de todas as estruturas de mesmo nível hierárquico nele contidas.

é mais fácil de compreender do que nesta forma equivalente:

```
se (produto está sujeito a imposto) então (se (preço > limite) então (pagar x)
senão (pagar y)) senão (pagar z)
```

Assim, adotaremos a endentação em nosso pseudocódigo. (Note-se que podemos usá-la para alinhar um parêntese que se feche diretamente abaixo de seu correspondente, para simplificar o processo de identificação do escopo de instruções ou frases.)

Queremos usar nosso pseudocódigo para descrever atividades que possam ser usadas em outras aplicações como ferramentas abstratas. A Ciência da Computação tem diversas designações para tais unidades de programação, entre as quais contam-se os termos *subprograma*, *sub-rotina*, *procedimento*, *módulo* e *função*, cada qual com suas próprias nuances de significado. Adotaremos *procedimento* no nosso pseudocódigo e usaremos este termo para introduzir o nome pelo qual a unidade de pseudocódigo será chamada. Mais precisamente, iniciaremos sempre um módulo de pseudocódigo com uma declaração da forma

procedimento nome

em que *nome* é o do módulo que se está especificando. A esta declaração introdutória, se seguem instruções, as quais definem a ação do módulo. Por exemplo, a Figura 4.4 mostra uma representação em pseudocódigo de um procedimento chamado *Saudações*, que imprime três vezes a mensagem “Olá”.

Sempre que for necessário utilizar, em algum outro ponto do nosso pseudocódigo, a tarefa que é realizada por um procedimento, sua ativação será solicitada pelo nome. Por exemplo, se dois procedimentos tiverem os nomes *ProcessarEmpréstimo* e *RejeitarAplicação*, então poderemos solicitar os seus serviços dentro de uma estrutura se-então-senão, escrevendo:

```
se (...) então (Executar o procedimento ProcessarEmpréstimo)
senão (Executar o procedimento RejeitarAplicação)
```

Isto resultará na execução do procedimento *ProcessarEmpréstimo* sempre que a condição testada for verdadeira, ou na de *RejeitarAplicação*, quando for falsa.

Deve-se sempre projetar procedimentos de forma que sejam tão gerais quanto possível. Convém que um procedimento projetado para ordenar listas de nomes seja capaz de ordenar qualquer lista — não apenas uma determinada. Assim, ele deve ser escrito de modo que a lista a ser ordenada não seja especificada pelo próprio procedimento. Ao contrário, convém que, dentro da representação do procedimento, a lista seja referenciada por meio de um nome genérico.

No nosso pseudocódigo, adotaremos a convenção de listar entre parênteses todos esses nomes genéricos, na mesma linha em que identificamos o nome do procedimento. Em particular, um procedimento chamado *Ordena*, que for projetado para ordenar qualquer lista de nomes, começaria com a declaração:

procedimento Ordena (Lista)

Mais adiante, na representação, no ponto em que for necessário efetuar uma referência à lista a ser ordenada, o nome genérico *Lista* será usado. Por outro lado, nas ocasiões em que os serviços do procedimento *Ordena* forem solicitados, deveremos identificar a lista que deverá substituir o nome fictício *Lista*, utilizado no procedimento *Ordena*. Assim, conforme as nossas necessidades, escreveremos algo como:

Aplicar o procedimento *Ordena* à lista dos membros da organização

e

Aplicar o procedimento *Ordena* à lista de convidados do casamento

dependendo de nossas necessidades.

procedimento Saudações
 Contador ← 3;
enquanto (Contador > 0) **faça**
 (imprimir a mensagem “Olá” e
 Contador ← Contador + 1)

Figura 4.4 O procedimento *Saudações* em pseudocódigo.

Convém recordar que o objetivo do nosso pseudocódigo é o de fornecer meios para representar algoritmos de uma maneira legível e informal. Queremos um sistema de notação que nos ajude a expressar idéias — sem nos tornarmos escravos de regras formais e rigorosas. Assim, estaremos livres para expandir ou modificar nosso pseudocódigo quando necessário. Em particular, se as instruções dentro de um conjunto de parênteses envolverem outras instruções dentro de parênteses, poderá ficar difícil emparelhar visualmente parênteses que abrem com os que fecham. Nesses casos, muita gente acha útil complementar um parêntese que está se fechando com um comentário explicativo de que a instrução ou frase está sendo terminada. Especificamente, pode-se complementar o parêntese final de uma instrução *enquanto* com as palavras “fim do enquanto” produzindo uma instrução como

enquanto (...) faça

(.

.

) **fim do enquanto**

ou talvez

enquanto (...) faça

(**se (...)**

então (

.

) **fim do se**

) **fim do enquanto**

O fato é que estamos tentando expressar um algoritmo em uma forma legível, e então podemos introduzir auxílios visuais (indentação, comentários etc.) para atingir esta meta. Além disso, se encontramos um tema recorrente que ainda não foi incorporado ao pseudocódigo, podemos decidir pela extensão do nosso pseudocódigo, adotando um sistema consistente para representar o novo conceito.



QUESTÕES/EXERCÍCIOS

1. Uma primitiva em um dado contexto pode tornar-se, em outro contexto, uma combinação de primitivas. Por exemplo, a nossa instrução *Enquanto* é uma primitiva em nosso pseudocódigo, embora seja implementada como uma combinação de instruções de máquina. Dê dois exemplos deste fenômeno, em situações nas quais não sejam empregados computadores.
2. Em que sentido a construção de procedimentos corresponde à de primitivas?
3. O algoritmo de Euclides calcula o máximo divisor comum de dois inteiros positivos X e Y pelo seguinte processo:
Enquanto nenhum dos valores de X e Y for zero, continuar dividindo o maior deles pelo menor, atribuindo para X e Y os valores do divisor e do resto, respectivamente. (O valor final de X será o máximo divisor comum que se deseja calcular.)
Expresse este algoritmo em nosso pseudocódigo.
4. Descreva um conjunto de primitivas que seja utilizado em uma área que não seja a programação de computadores.

4.3 Descoberta de algoritmos

O desenvolvimento de um programa consiste em duas atividades — descobrir o algoritmo subjacente e representá-lo na forma de um programa. Até este ponto temos nos preocupado com a representação de algoritmos, sem considerar como foram criados. Ainda hoje, o delineamento de algoritmos é o passo que apresenta mais desafios no processo de desenvolvimento de *software*. Afinal, descobrir um algoritmo é encontrar um método de solucionar um problema. Assim, entender como os algoritmos são descobertos é entender o próprio processo de resolução de problemas.

A arte da resolução de problemas

As técnicas de resolução de problemas e a necessidade de aprender mais sobre eles não são um tema exclusivo da Ciência da Computação, mas permeiam quase todas as áreas do conhecimento. A relação íntima entre o processo de descoberta de algoritmos e a resolução geral de problemas levou os cientistas da computação a se unirem aos demais disciplinas em busca de técnicas melhores para a resolução de problemas. Em última instância, deseja-se reduzir o processo de resolução de problemas a um algoritmo autônomo, mas comprovou-se que essa meta é impossível. (Este importante resultado é material de discussão do Capítulo 11, no qual mostraremos a existência de problemas que não têm soluções algorítmicas.) Assim, a capacidade de resolver problemas permanece mais como uma habilidade artística a ser desenvolvida do que como uma ciência precisa, a ser aprendida.

Como evidência desta natureza ilusória e artística da resolução de problemas, as seguintes fases, vagamente definidas, do processo de resolução de problemas, apresentadas em 1945 pelo matemático G. Polya, permanecem como os princípios básicos em que atualmente se apoiam as tentativas de treinamento da habilidade humana de resolução de problemas.

- Fase 1.* Entender o problema.
- Fase 2.* Construir um plano para solucionar o problema.
- Fase 3.* Colocar o plano em funcionamento.
- Fase 4.* Avaliar a solução quanto à precisão e ao seu potencial como ferramenta para solucionar outros problemas.

Traduzidas para o contexto do desenvolvimento de programas, essas fases se tornam:

- Fase 1.* Compreender o problema.
- Fase 2.* Adquirir uma idéia da forma como um procedimento algorítmico poderia resolver o problema.
- Fase 3.* Formular o algoritmo e representá-lo na forma de um programa.
- Fase 4.* Avaliar o programa quanto à precisão e ao seu potencial como ferramenta para resolver outros problemas.

Tendo apresentado a lista de Polya, devemos enfatizar que essas fases não são passos a seguir quando se tenta resolver um problema, mas fases a serem seguidas algum dia, durante o processo de solução. A palavra-chave aqui é *seguidas*. Não se resolve problemas seguindo. Pelo contrário, para resolver um problema, é preciso tomar a iniciativa e liderar. Se enfrentarmos uma tarefa de resolução de um problema com um raciocínio do tipo *Agora terminou a fase 1, e é hora de passar para a fase 2*, dificilmente teremos sucesso. No entanto, se nos envolvermos com o problema até conseguirmos resolvê-lo, podemos observar o que foi realizado para isso e constatar que as quatro fases de Polya foram realmente cumpridas.

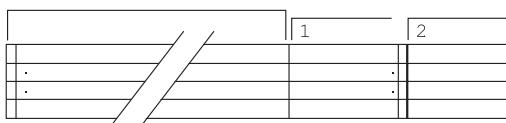
Outra observação importante é que as quatro fases de Polya não são necessariamente executadas em seqüência. Ao contrário do que defendem muitos autores, os solucionadores de problema mais bem-sucedidos costumam começar formulando estratégias para resolver um problema (fase 2) antes mesmo de compreendê-lo completamente (fase 1). Então, se estas estratégias falharem (durante as fases 3 ou 4), o solucionador do problema terá adquirido um conhecimento maior das complexidades do mesmo e, com

Estruturas iterativas na música

Os músicos usaram e programaram estruturas iterativas séculos antes dos cientistas da computação. Sem dúvida, a estrutura de uma canção (composta de múltiplos versos, cada um seguido do refrão) é exemplificada pela instrução “enquanto”.

enquanto (existir um verso) **faça**
 (cante o próximo verso;
 cante o refrão)

Além disso, a notação



é meramente um modo de o compositor expressar a estrutura

$N \leftarrow 1;$
enquanto ($N < 3$) **faça**
 (toque o trecho;
 toque a N -ésima finalização;
 $N \leftarrow N + 1$)

base nesta compreensão mais profunda, poderá reformular as demais estratégias, aumentando assim as suas possibilidades de sucesso.

É bom frisar que estamos discutindo sobre a forma como os problemas são resolvidos, e não sobre como gostaríamos que fossem resolvidos. O ideal seria se pudéssemos eliminar todo o desperdício inerente aos processos de tentativa e erro, já mencionados. Nos casos de desenvolvimento de grandes sistemas de *software*, descobrir um engano muito tarde — por exemplo, na fase 4 — pode representar um tremendo desperdício de recursos.

Evitar catástrofes como essa é uma importante meta dos engenheiros de *software* (Capítulo 6), os quais têm tradicionalmente insistido em obter a compreensão plena de um problema antes de tentar uma solução. Alguém poderia argumentar, contudo, que não se chega ao completo entendimento de um problema antes de sua plena solução, que o simples fato de existir um problema não resolvido implica uma falta de compreensão. Insistir na compreensão plena do problema antes de propor qualquer solução representa, portanto, uma atitude de um tanto idealista.

Como ilustração, considere-se o seguinte problema:

O indivíduo A é encarregado de determinar a idade dos três filhos do indivíduo B. B conta para A que o produto das idades das crianças é 36. Depois de levar em conta esta informação, A responde que precisa de mais informação, e então B conta para A a soma das idades das crianças. Novamente, A diz que necessita de mais informação, e assim B conta que a criança mais velha toca piano. Depois de ouvir isto, A responde a B a idade das três crianças.

Qual é a idade das crianças?

À primeira vista, a última informação parece não ter conexão com o problema, mas é ela que permite a A finalmente determinar a idade das crianças. Como isso acontece? Vamos formular um plano de ataque para depois segui-lo, embora ainda tenhamos muitas perguntas a fazer acerca do problema. Nossa plano consistirá em rastrear os passos descritos pelo enunciado do problema, mantendo, passo a passo, o registro da informação disponível ao indivíduo A, ao longo do tempo.

A primeira informação dada a A é que o produto das idades das crianças é 36. Isto significa que a tripla ordenada, que representa as três idades, é uma das listadas na Figura 4.5(a). A informação seguinte fornece a soma dos elementos da tripla procurada. Não nos foi dito qual seria esta soma, mas sabemos que esta informação não foi suficiente para A isolar a tripla correta; então esta deverá ser tal que sua soma apareça pelo menos duas vezes na tabela da Figura 4.5(b). Entretanto, as únicas tripas que aparecem na Figura 4.5(b) com somas idênticas são (1,6,6) e (2,2,9), ambas com a soma igual a 13. Esta é a informação disponível para A ao receber a última informação, e é quando finalmente entendemos o significado desta última informação. É irrelevante o fato de tocar piano, mas é vital a informação da

existência de (só) uma criança mais velha. Isto elimina a tripla (1,6,6) e, assim, nos permite concluir que as idades das crianças são 2, 2 e 9 anos.

Assim, neste caso, enquanto não tentamos implementar nosso plano para resolver o problema (fase 3), não foi possível ter uma plena compreensão do mesmo (fase 1). Se tivéssemos teimado em completar a fase 1 antes de começar a resolver o problema, provavelmente nunca teríamos achado as idades das crianças. Tais irregularidades no processo de resolução de problemas contribuem para dificultar o desenvolvimento de métodos sistemáticos para a resolução de problemas.

Outra irregularidade é a inspiração misteriosa que ocorre a um solucionador de problema em potencial que, tendo trabalhado aparentemente sem sucesso na resolução de um problema, pode mais tarde visualizar subitamente uma solução enquanto estiver realizando outra tarefa. Este fenômeno foi identificado por H. von Helmholtz, por volta de 1896, e discutido pelo matemático Henri Poincaré em uma conferência diante da Société de Psychologie, em Paris. Lá, Poincaré descreveu sua experiência de concepção da solução para um problema em que havia inicialmente trabalhado e depois posto de lado e iniciado outros projetos. O fenômeno ocorre como se uma parte do subconsciente da mente humana continuasse trabalhando e, caso tenha sucesso, imediatamente passa a solução obtida para a mente consciente. Hoje, o período transcorrido entre o trabalho consciente em um problema e a inspiração súbita é conhecido como período de incubação, e a sua compreensão continua sendo objeto de pesquisa.

Suba o primeiro degrau

Nossa discussão sobre a resolução de problemas tem adotado um ponto de vista um tanto filosófico, evitando entrar em confronto direto com a questão de como tentar resolver o problema. Há, é claro, numerosas abordagens de resolução de problemas, todas com possibilidades de sucesso em determinadas situações. Identificaremos em breve algumas delas. Por enquanto, podemos sempre identificar uma mesma conduta, qualquer que seja a técnica adotada, que pode ser resumida no seguinte: *suba o primeiro degrau*. Para ilustrar, consideremos o seguinte problema simples:

Antes de A, B, C e D participarem de uma corrida, eles fizeram as seguintes previsões:

- A previu que B ganharia.
- B previu que D seria o último.
- C previu que A seria o terceiro.
- D previu que a previsão de A estaria correta.

Apenas uma destas previsões deu certo, e esta foi feita pelo vencedor da corrida. Em que ordem A, B, C e D terminam a corrida?

Depois de ler o problema e analisar os dados, não se leva muito tempo para perceber que, dado que as previsões de A e D são equivalentes, e só uma previsão era verdadeira, tanto a previsão de A como a de D devem ser falsas. Assim, nem A nem D foram os vencedores. Neste ponto, subimos o primeiro degrau e obter a solução completa para o nosso problema será simplesmente uma questão de estender a partir daí o nosso conhecimento. Como a previsão de A é falsa, B também não pode ter sido o vencedor. A única escolha que resta para o vencedor é C. Portanto, C ganhou a corrida, logo, sua previsão estava correta, e concluímos que A chegou em terceiro lugar. Isso significa que a ordem final pode ter sido apenas CBAD ou então CDAB. O primeiro palpite deve ser, no entanto, descartado, já que a previsão de B deve ser falsa. Então, a ordem final foi CDAB.

a. Triplas cujo produto é 36	b. Somas das tripas da parte (a)
(1, 1, 36)	1+1+36=38
(1, 2, 18)	1+2+18=21
(1, 3, 12)	1+3+12=16
(1, 4, 9)	1+4+9=14
(1, 6, 6)	1+6+6=13
(2, 2, 9)	2+2+9=13
(2, 3, 6)	2+3+6=11
(3, 3, 4)	3+3+4=10

Figura 4.5

É claro que ser aconselhado a subir o primeiro degrau não é o mesmo que ser instruído acerca de como isso deve ser feito. Alcançar este ponto de apoio, bem como perceber como ampliar este ponto inicial de apoio até a obtenção de uma solução completa do problema, requer iniciativas criativas da parte do possível solucionador do problema. Há contudo diversos enfoques gerais, propostos por Polya e outros, sobre como encontrar tal ponto de partida. Deve-se tentar trabalhar o problema *de marcha a ré*. Por exemplo, se ele consiste em encontrar um modo de produzir uma certa saída a partir de uma dada entrada, pode-se partir desta saída e tentar percorrer o caminho de volta à entrada fornecida. Esta abordagem é tipicamente adotada por alguém que tente descobrir o algoritmo de construção do pássaro em dobradura de papel, mostrado na seção anterior. Ele tende a desfazer as dobras do pássaro acabado, em uma tentativa de observar de que maneira foi ele construído.

Outro enfoque para a resolução geral de problemas é procurar um problema relacionado com o que nos interessa no momento, que seja mais fácil de resolver ou já tenha sido resolvido antes, e então tentar solucionar o problema usando a mesma forma de resolução. Esta técnica é especialmente valiosa no contexto do desenvolvimento de programas. Freqüentemente, a maior dificuldade encontrada no desenvolvimento de programas não é resolver uma instância particular de um problema, mas encontrar um algoritmo geral, que possa ser empregado para resolver todas as instâncias desse problema. Mais precisamente, se estivermos diante da tarefa de desenvolver um programa encarregado de ordenar alfabeticamente listas de nomes, então a nossa tarefa não será a de ordenar uma determinada lista, mas a de encontrar um algoritmo geral capaz de ordenar qualquer lista de nomes. Assim, embora as instruções seguintes ordenem corretamente a lista David, Alice, Carol, Bob, elas não constituem o algoritmo de propósito geral que desejamos:

Trocá de posição os nomes *David* e *Alice*.

Mover o nome *Carol* para a posição entre os nomes *Alice* e *David*.

Mover o nome *Bob* para a posição entre os nomes *Alice* e *Carol*.

O que precisamos é de um algoritmo que ordene esta lista, bem como quaisquer outras que encontrarmos. Isso não significa que a nossa solução para ordenar uma lista particular seja totalmente desprezível na nossa procura por um algoritmo de propósito geral. Por exemplo, estaríamos colocando nosso pé na porta ao considerar tais casos particulares, em uma tentativa de determinar princípios gerais que possam, por sua vez, ser usados para desenvolver o algoritmo de propósito geral desejado. Nesse caso, então, a nossa solução será obtida pela técnica da resolução de um conjunto de problemas relacionados.

Pode-se também aplicar a técnica do **refinamento sucessivo**, que consiste essencialmente em não tentar realizar imediatamente uma tarefa inteira, em todos os seus detalhes. A técnica propõe que primeiro se decomponha o problema em vários subproblemas. A idéia é que, dividindo o problema original pode-se visualizar a solução global desejada composta de uma série de passos. A técnica do refinamento sucessivo propõe que os passos assim construídos sejam, por sua vez, decompostos em passos menores, e estes, em outros ainda menores, e assim por diante, até que o problema inteiro seja reduzido a um conjunto de subproblemas cujas soluções possam ser facilmente obtidas.

Seguindo esta linha, o refinamento sucessivo constitui uma metodologia cima-baixo (*top-down*), na qual o processo de desenvolvimento leva do geral para o particular. A metodologia baixo-cima (*bottom-up*) leva do específico para o geral. Embora teoricamente contrastantes, na prática, os dois enfoques se complementam. Por exemplo, a decomposição de um problema, proposta pelo refinamento da metodologia cima-baixo, freqüentemente é guiada pela intuição do solucionador de problema, a qual opera de forma baixo-cima.

As soluções produzidas pela técnica do refinamento sucessivo apresentam uma estrutura modular natural, e é esta a razão principal para a sua popularidade em projetos de algoritmos. Se um algoritmo tem uma estrutura modular natural, pode ser facilmente adaptado a uma representação modular, o que conduz ao desenvolvimento de um programa gerenciável. Além disso, os módulos produzidos pelo refinamento sucessivo são compatíveis com o conceito de programação em equipe, no qual diversas pessoas

são nomeadas para o desenvolvimento de um produto de *software*. Afinal, se a tarefa do *software* está dividida em subproblemas (ou módulos em potencial), os elementos do grupo podem trabalhar independentemente nestas subtarefas, sem que uns interfiram no trabalho dos outros.

Essas vantagens do refinamento sucessivo, no contexto do desenvolvimento de *software*, conquistaram muitos adeptos. Apesar de todos os seus pontos positivos, o refinamento sucessivo não é a palavra final no processo de descoberta de algoritmos. Pelo contrário, ele é essencialmente uma ferramenta organizacional, cujos atributos, em matéria de resolução de problemas, decorrem dessa organização. O refinamento sucessivo é uma metodologia óbvia para ser usada quando se organiza uma campanha política de âmbito nacional, se escreve um artigo ou se planeja uma convenção de vendas. De modo similar, a maioria dos projetos de desenvolvimento de *software* encontrados na comunidade de processamento de dados apresenta um grande componente organizacional. A tarefa não é tanto a de descobrir algum algoritmo novo e surpreendente, mas a de organizar, em um pacote coerente, as tarefas a serem executadas. Por essas razões, o refinamento sucessivo se tornou a principal metodologia de projeto em processamento de dados.

No entanto, é apenas um dos muitos métodos de interesse para os cientistas da computação e, portanto, não se deve acreditar que a descoberta de qualquer algoritmo possa ser feita por meio de refinamento sucessivo. De fato, adotar na tarefa de resolução de problemas noções preconcebidas e ferramentas pré-selecionadas às vezes pode mascarar a simplicidade de um problema. O problema das idades das crianças, discutido anteriormente, é um excelente exemplo desse fenômeno. Os estudantes de álgebra invariavelmente abordam o problema como um sistema de equações simultâneas, abordagem esta que leva à conclusão errônea de que não existe solução e freqüentemente induz o solucionador a acreditar que a informação recebida não é suficiente para resolver o problema. Outro exemplo é o seguinte:

Ao subir em um barco a partir de um cais, o seu chapéu cai na água, sem você perceber. A água do rio flui a 2,5 milhas por hora, de modo que o seu chapéu começa a se afastar, seguindo o fluxo, na superfície da água. Enquanto isso, você continua a viajar rio acima a uma velocidade de 4,75 milhas por hora em relação à água. Depois de 10 minutos, você percebe que perdeu o chapéu, manobra o barco e começa a perseguir o chapéu rio abaixo. Quanto tempo levará para alcançar o seu chapéu?

A maioria dos estudantes de álgebra e também os entusiastas em calculadora de bolso tratam este problema determinando a distância percorrida, rio acima, pelo barco em 10 minutos e a distância percorrida pelo chapéu, rio abaixo, durante este mesmo tempo. Então, eles determinam quanto tempo levará para o barco percorrer, rio abaixo, até alcançar esta posição. Contudo, quando o barco chegar na posição, o chapéu já terá flutuado para mais longe, rio abaixo! Assim, o pretenso solucionador de problema começa a aplicar técnicas de cálculo diferencial ou é apanhado em um ciclo para determinar sempre onde o chapéu estará, a cada vez que o barco for para onde o chapéu estava.

E o problema é muito mais simples que isto. O truque é resistir ao desejo de começar a escrever fórmulas e fazer cálculos. Em vez disto, precisamos ajustar a nossa perspectiva. Todo o problema acontece no rio. O fato de a água estar se movendo em relação à orla é irrelevante. Pense no mesmo problema adaptado a uma esteira grande, no lugar de um rio. Primeiro, resolva o problema com a esteira parada. Se, de pé sobre a esteira, você depositar a seus pés o chapéu e caminhar sobre a esteira, afastando-se dele durante 10 minutos, então você levaria 10 minutos para retornar ao seu chapéu. Agora ponha a esteira em movimento. Isto significa que a paisagem começará a se mover ao contrário, mas, como você está sobre a esteira, isto não muda a sua posição em relação a ela nem ao chapéu: você levará os mesmos 10 minutos para retornar ao seu chapéu.

Concluímos que a descoberta de algoritmos continua sendo uma arte desafiadora, que deve ser desenvolvida ao longo de um certo período, em vez de ser ensinada como um assunto que consiste em métodos bem definidos. Treinar um potencial solucionador de problemas para seguir certas metodologias é tolher sua criatividade, que na verdade deveria ser incentivada.



QUESTÕES/EXERCÍCIOS

1. a. Encontre um algoritmo para resolver o seguinte programa: Dado um inteiro positivo n , achar, dentre todas as listas de inteiros positivos cuja soma é n , a lista de inteiros positivos cujo produto seja máximo. Por exemplo, se $n = 4$, a lista desejada é 2, 2 pois 2×2 é maior que $1 \times 1 \times 1 \times 1$, que $2 \times 1 \times 1$ e que 3×1 . Se $n = 5$, a lista desejada é 2, 3.
b. Qual seria a lista desejada se $n = 2001$?
c. Explique como você subiu o primeiro degrau.
2. a. Suponha que nos tenha sido entregue um tabuleiro de damas, que consista em 2^n linhas e 2^n colunas, para algum inteiro positivo n , e em uma caixa de peças, moldadas no formato de L, cobrir exatamente três quadrados do tabuleiro. Se qualquer peça estiver, ainda que parcialmente, fora do tabuleiro, será possível rearranjá-las de modo que cubram todo o tabuleiro, sem que nunca se sobreponham nem fiquem fora dos limites do tabuleiro?
b. Explique de que maneira a sua solução para o item (a) pode ser usada para mostrar que $2^{2n} - 1$ é divisível por 3 para todos os inteiros positivos n .
c. Como os itens (a) e (b) se relacionam com as fases de resolução de problemas propostas por Polya?
3. Decodifique a seguinte mensagem, que está escrita em inglês. Então, explique como você subiu o primeiro degrau.
Pdeo eo pda yknnayp wjosan.

4.4 Estruturas iterativas

Nossa meta agora será estudar algumas das estruturas repetitivas usadas na descrição de processos algorítmicos. Nesta seção, discutiremos as **estruturas iterativas**, nas quais é efetuada a repetição da execução de um conjunto de instruções, e na próxima, introduziremos a técnica de recursão. Além disso, como exemplos, introduziremos alguns algoritmos conhecidos de busca* e ordenação** — as buscas seqüencial e binária e a ordenação por inserção —, uma vez que envolvem a aplicação de estruturas repetitivas. Iniciamos com o algoritmo de busca seqüencial.

Algoritmo de busca seqüencial

Consideremos o problema de procurar em uma lista a ocorrência de um valor. Desejamos desenvolver um algoritmo que determine se este valor está ou não presente na lista. Se o valor estiver na lista, consideraremos encerrada com sucesso a nossa operação de busca; caso contrário, a busca terá fracassado. Vamos admitir que a lista esteja ordenada de acordo com algum critério conhecido. Por exemplo, se for uma lista de nomes, assumimos que os nomes estão dispostos em ordem alfabética, ou, se for uma lista de números, que estão dispostos em ordem crescente.

Para colocar um pé na porta, imaginemos uma maneira de procurar um nome em uma lista de aproximadamente 20 convidados. Nessas condições, percorremos a lista a partir do seu início e compararemos cada elemento com o nome desejado. Se encontrarmos este nome, a busca terá terminado com sucesso. Porém, se chegarmos ao final da lista sem encontrar o nome, a nossa busca terá fracassado. De fato, se atingirmos um nome maior (alfabeticamente) que o desejado sem ter encontrado este último,

*N. de T. Em inglês, *searching*.

**N. de T. Em inglês, *sorting*.

nossa busca terá fracassado. (Convém lembrar que a lista está organizada em ordem alfabética, e que alcançar um nome alfabeticamente posterior ao nome procurado é suficiente para indicar que o objeto desejado não se encontra na lista.) Em suma, a nossa vaga idéia é a de continuarmos nos aprofundando na lista enquanto ainda houver nomes a serem testados, desde que o nome que se procura seja alfabeticamente posterior ao último nome da lista que já tenha sido considerado.

Em nosso pseudocódigo, este processo pode ser representado assim:

```

Selecionar como elemento em teste o primeiro elemento da lista.
enquanto (nome procurado > elemento em teste e
          ainda há elementos a considerar)
    faça (Selecionar como elemento em teste o próximo da lista)

```

Ao terminar a execução da estrutura enquanto, uma de duas condições será verdadeira: ou o valor desejado foi encontrado ou ele não consta na lista. Em qualquer caso, detectamos uma busca com sucesso ao comparar o elemento em teste com o valor desejado. Se eles forem iguais, a busca terá tido êxito. Assim, ao final do nosso pseudocódigo, acrescentamos a seguinte instrução:

```

se (valor desejado = elemento em teste)
    então (Informar que a busca foi um sucesso)
    senão (Informar que a busca fracassou)

```

Finalmente, observamos que a primeira instrução da nossa rotina de busca está baseada na suposição de que a lista em questão contém pelo menos um elemento. Podemos argumentar que esta é uma suposição segura, mas, para ter certeza, podemos reescrever a nossa rotina como a opção senão da seguinte instrução:

```

se (A lista está vazia)
    então (Informar que a busca fracassou.)
    senão ...

```

Isto produz o procedimento mostrado na Figura 4.6. Note-se que este procedimento pode ser chamado por outros, usando instruções como:

Aplicar à lista de passageiros o procedimento de busca, à procura do nome Darrel Baker.
para verificar se Darrel Baker é um passageiro. Entretanto, a instrução

Aplicar a uma lista de ingredientes o procedimento de busca, considerando noz-moscada como o valor a ser localizado.

permite descobrir se a palavra noz-moscada aparece na lista de ingredientes.

Em resumo, o algoritmo apresentado na Figura 4.6 consulta os elementos seqüencialmente, na ordem em que aparecem na lista. Por isso, é chamado algoritmo de **busca seqüencial**. Por causa de sua simplicidade, freqüentemente é usado para listas pequenas, ou quando outros fatores recomendam o seu uso. No caso de listas longas, as buscas seqüenciais não se mostram tão eficientes quanto outras técnicas (o que constataremos em breve).

O controle do laço

O uso iterativo de uma instrução ou de seqüências de instruções é um importante conceito de algoritmos. Uma forma de implementação de repetições é através do uso da estrutura conhecida como **laço** (*loop*), na qual uma coleção de instruções, que forma o corpo do laço, é executada de modo repetitivo, sob a direção de um processo controlador. Um exemplo típico de tais estruturas pode ser encontrado no algoritmo de busca seqüencial apresentado na Figura 4.6. Nele, usamos uma instrução enquanto para controlar a repetição da execução da instrução Selecionar como elemento em teste o próximo da Lista. De fato, a instrução

```

procedimento Busca (Lista,ValorDesejado);
se (Lista vazia)
  então
    (informar que a busca fracassou)
  senão
    (Selecionar como elemento em teste o primeiro elemento da lista;
     enquanto (ValorDesejado > elemento em teste e
       ainda há elementos a considerar)
       faça (Selecionar como elemento em teste o próximo da lista.);)
    se (ValorDesejado = elemento em teste)
      então (informar que a busca foi bem-sucedida.)
      senão (informar que a busca fracassou.)
  ) fim do se

```

Figura 4.6 O algoritmo de busca seqüencial em pseudocódigo.

Como regra geral, o uso de uma estrutura iterativa proporciona maior flexibilidade que simplesmente escrever várias vezes o corpo de tal estrutura. Por exemplo, embora a estrutura iterativa

Executar três vezes a instrução “Adicionar uma gota de ácido sulfúrico”.

seja equivalente à seqüência:

Adicionar uma gota de ácido sulfúrico.
 Adicionar uma gota de ácido sulfúrico.
 Adicionar uma gota de ácido sulfúrico.

não podemos produzir uma seqüência semelhante que seja equivalente à instrução iterativa denotada por:

enquanto (o nível de pH for maior que 4) **faça**
 (adicionar uma gota de ácido sulfúrico)

porque não sabemos com antecedência quantas gotas de ácido serão necessárias.

Analisemos, agora mais de perto, a composição do controle do laço. Pode-se ficar tentado a dar pouca importância a essa parte da estrutura iterativa, uma vez que é o corpo da iteração que de fato executa a tarefa desejada (por exemplo, adicionar gotas de ácido) — as atividades de controle da iteração aparecem como uma sobrecarga, simplesmente porque decidimos executar de modo iterativo o corpo da iteração. No entanto, a experiência mostra que o controle das iterações é a parte desta estrutura que se mostra mais sujeita a erros, merecendo, portanto, uma atenção especial.

O controle de um laço consiste em três atividades: iniciação, teste e modificação (Figura 4.7), sendo necessária a presença dos três componentes para que haja sucesso nesse controle. A atividade de teste deve propiciar o encerramento da iteração ao detectar uma condição que sinalize o término da operação. Esta condição é conhecida como condição terminal. É com este propósito que existe uma atividade de teste em cada instrução enquanto do nosso pseudocódigo. No caso da instrução enquanto, contudo, a condição declarada determina a execução do corpo do laço — a condição terminal é a negação da condição que aparece na estrutura enquanto. Assim, na instrução enquanto da Figura 4.6, a condição terminal é:

(ValorDesejado ≤ elemento em teste) ou (não existem mais elementos a considerar)

As outras duas atividades de controle da iteração asseguram que a condição terminal venha de fato a ocorrer. O passo de iniciação estabelece um ponto de partida para a iteração, e o passo de modificação altera esta condição até que seja atingido o seu término. Por exemplo, na Figura 4.6, a iniciação acontece na instrução que precede a instrução enquanto, onde se estabelece que o elemento em teste é o

enquanto (*condição*) **faça** (*corpo*)

exemplifica o conceito de uma estrutura iterativa, cuja execução mostra um padrão cíclico

testar a *condição*

executar o *corpo*

testar a *condição*

executar o *corpo*

.

.

testar a *condição*

até que se torne *falso* o valor testado da *condição*.

primeiro elemento da lista. O passo de modificação, neste caso, é efetuado no corpo da iteração, e nele a nossa posição de interesse (identificada pelo elemento em teste) vai sendo deslocada em direção ao final da lista. Assim, tendo executado o passo de iniciação, chega-se à condição terminal, após a aplicação repetida do passo de modificação. (Se nunca for encontrado o valor procurado, chegaremos, em última instância, ao final da lista.)

Devemos enfatizar a exigência de que os passos de iniciação e de modificação conduzam a uma condição terminal apropriada. Esta característica é fundamental para que haja um controle adequado da iteração, e por isso é preciso ter certeza absoluta de sua correta aplicação sempre que se projetar uma estrutura iterativa. Um insucesso nesta empreitada pode acarretar erros, mesmo nos casos mais simples. Um exemplo comum é encontrado nas instruções:

```
Número ← 1;
enquanto (Número ≠ 6) faça
    (Número ← Número + 2)
```

Aqui a condição terminal é o valor de Número ser 6. Entretanto, seu valor inicial é 1, que depois é incrementado em 2 no passo de modificação. Assim, à medida que o laço vai sendo executado, os valores atribuídos a Número serão 1, 3, 5, 7, 9 e assim por diante, mas nunca o valor 6. Logo, o laço jamais terminará.

Há duas estruturas iterativas comuns, que diferem somente quanto à ordem em que são executados os componentes de controle do laço. A primeira pode ser exemplificada pela seguinte instrução, em nosso pseudocódigo:

enquanto (condição) faça (ação)

cuja semântica está representada na Figura 4.8, na forma de um **fluxograma**. Tais diagramas empregam algumas figuras geométricas para representar passos individuais, bem como setas para indicar a seqüência de execução desses passos. As diferentes formas geométricas indicam o tipo da ação executada no passo correspondente. Um losango indica uma decisão. Um retângulo, uma instrução arbitrária ou uma seqüência de instruções. Note-se que o teste de terminação, em uma estrutura enquanto, é realizado antes da execução das atividades especificadas pelo corpo da iteração.

Em contraste, a estrutura mostrada na Figura 4.9 especifica que o corpo da iteração deve ser executado antes do teste de terminação. Nesta estrutura alternativa, o corpo da iteração é executado pelo menos uma vez, ao passo que na estrutura enquanto, nunca será executado se a condição de terminação já estiver satisfeita na primeira vez que for testada.

Usamos a forma sintática

repete (ação) até (condição)

em nosso pseudocódigo para representar a estrutura mostrada na Figura 4.9. Assim, a instrução

```
repete (retirar uma moeda do bolso)
até (não haver mais moedas no bolso)
```

Iniciação:	Estabelecer um estado inicial que será modificado para atingir uma condição de terminação.
Teste:	Comparar o estado corrente com a condição de terminação e finalizar a iteração se forem iguais.
Modificação:	Alterar o estado de modo a convergir para a condição de terminação.

Figura 4.7 Componentes de controle da iteração.

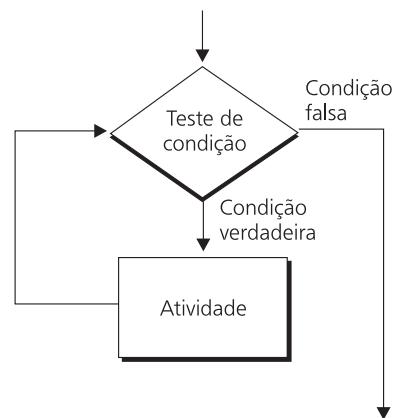


Figura 4.8 A estrutura iterativa *enquanto*.

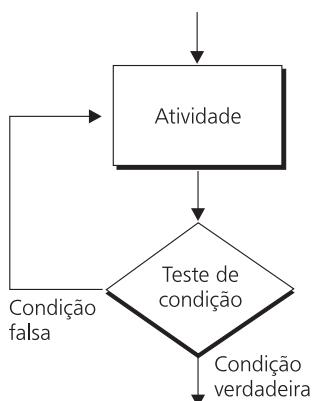


Figura 4.9 A estrutura iterativa *repete*.

admite que, no início, haja alguma moeda no seu bolso, enquanto

enquanto (ainda há moedas no bolso)
faça (retirar uma moeda do bolso)

não pressupõe tal condição inicial.

De acordo com a terminologia de nosso pseudocódigo, iremos nos referir a essas estruturas como estrutura iterativa *enquanto* e estrutura iterativa *repete*. Em contextos mais genéricos, você pode encontrar a estrutura iterativa *enquanto* com o nome de **laço pré-teste** (já que o teste de terminação é feito antes da execução do corpo do laço), e a estrutura *repete* com o nome de **laço pós-teste** (já que o teste de terminação é feito após a execução do corpo do laço).

O algoritmo de ordenação por inserção*

Como um exemplo adicional de estruturas iterativas, consideremos novamente o problema de ordenar alfabeticamente uma lista de nomes. Antes, porém, identifiquemos as restrições sob as quais trabalharemos. Nossa meta é efetuar uma ordenação de elementos internamente em uma lista. Em outras palavras, desejamos ordenar a lista e apenas reposicionar seus elementos dentro da própria lista, em vez de movê-los da lista para outra região de memória. A situação agora é análoga à do problema de ordenar uma lista cujos elementos, escritos em cartões, estão todos espalhados em uma escrivaninha lotada, sobre a qual foi aberto um pequeno espaço suficiente para conter os cartões, sem a possibilidade de empurrar objetos para abrir mais espaço. Esta restrição é comum em aplicações computacionais, não porque o espaço de trabalho dentro da máquina esteja necessariamente lotado, como é o caso da escrivaninha, mas simplesmente por querermos utilizar eficientemente a área de armazenamento disponível.

Vamos subir o primeiro degrau, começando por estudar de que forma poderíamos ordenar manualmente os cartões sobre a escrivaninha. Considere a seguinte lista de nomes:

Fred
Alice
David
Bill
Carol

Um modo de ordenar esta lista consiste em observar que a sublista formada apenas pelo primeiro nome, Fred, já está ordenada, porém a sub-lista formada pelos dois primeiros nomes, Fred e Alice, não. Assim, selecionamos o cartão com o nome Alice, deslocamos Fred para baixo na posição em que estava Alice, e então colocamos Alice no espaço livre no topo da lista, conforme representado pela primeira linha da Figura 4.10. Neste ponto, a nossa lista teria o seguinte aspecto:

Alice
Fred
David
Bill
Carol

*N. de T. Em inglês, *insertion sort*.

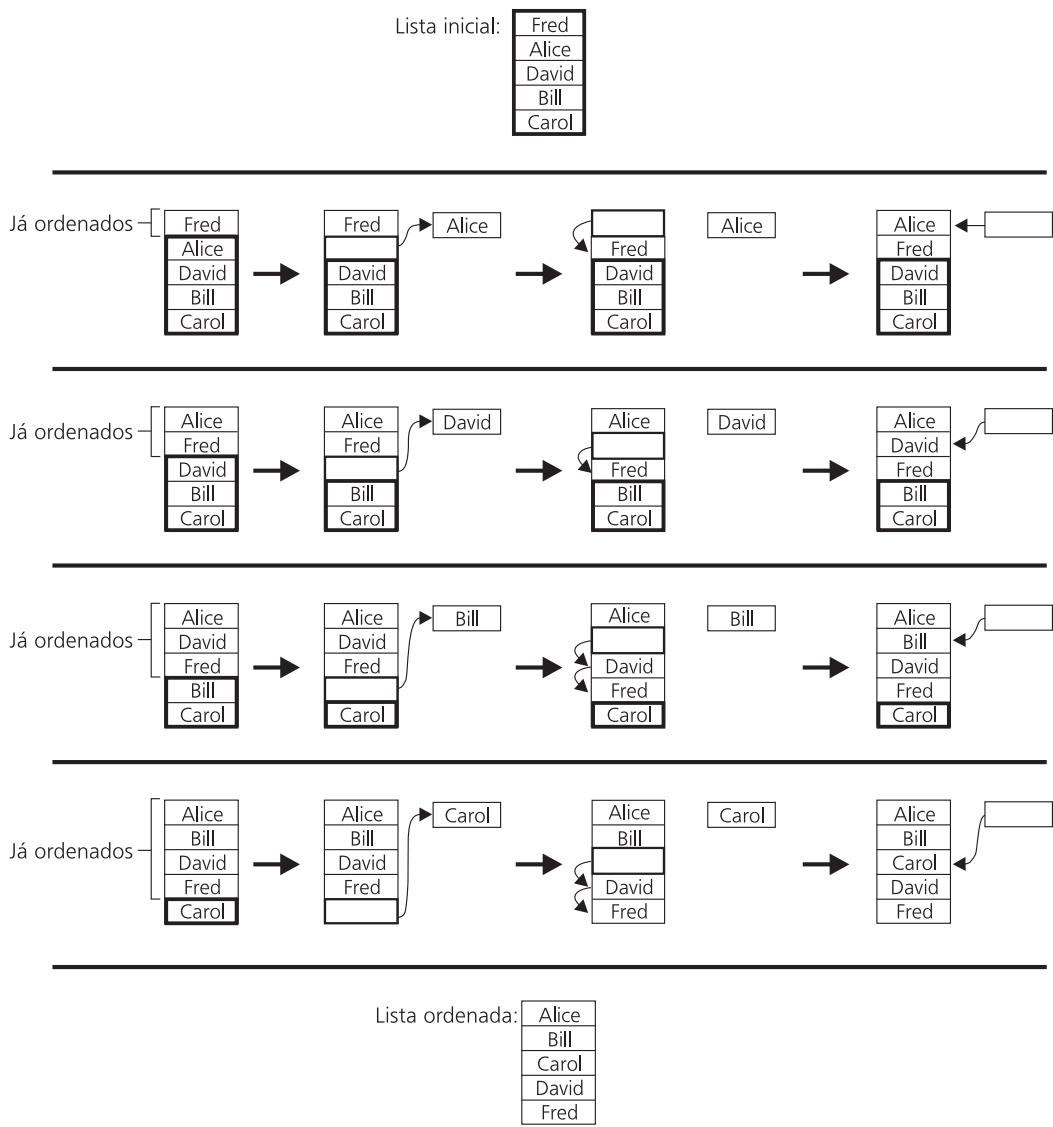


Figura 4.10 Ordenação da lista Fred, Alice, David, Bill e Carol.

Agora, os dois primeiros nomes formam uma sublista ordenada, porém os três primeiros ainda não. Assim, selecionamos o terceiro nome, David, deslocamos Fred para baixo, para ocupar o lugar de David, e em seguida inserimos o nome David na posição deixada por Fred, conforme mostrado na segunda linha da Figura 4.10. Agora, os três primeiros nomes da lista estão ordenados. Prosseguindo desta maneira, obteremos uma lista na qual os quatro primeiros nomes estarão ordenados. Selecionamos o quarto nome, Bill, deslocamos os nomes Fred e David para baixo e então inserimos o nome Bill na posição livre (veja a terceira linha da Figura 4.10). Finalmente, completamos o processo de ordenação, selecionando Carol, deslocando Fred e David para baixo e então inserindo Carol na posição livre resultante (veja a quarta linha da Figura 4.10).

Uma vez analisado o processo de ordenação de uma lista particular, a tarefa agora é a de generalizar este processo para obter um algoritmo geral de ordenação de listas. Com essa finalidade, podemos observar que cada linha da Figura 4.10 representa um mesmo processo geral: selecionar o primeiro nome na lista não-ordenada, deslocar para baixo os nomes já ordenados que forem posteriores ao nome extraído e reinserir este nome no local da lista ordenada em que foi gerada uma posição livre. Se identificarmos como elemento-pivô o nome extraído da lista não-ordenada, este processo pode ser expresso, no nosso pseudocódigo, como segue:

Mover o elemento-pivô para uma posição temporária, deixando na lista uma posição livre;
enquanto (houver um nome maior que o pivô acima da posição livre) **faça**
 (mover para a posição livre o nome encontrado logo acima dela, deixando, em seu lugar, uma nova posição livre)
 Mover o elemento-pivô para a posição livre resultante

A seguir, observamos que este processo deve ser executado de modo iterativo. No início, o pivô seria o segundo elemento da lista e então, antes de cada execução adicional, a seleção do pivô seria deslocada para baixo na lista, até que o último elemento fosse corretamente posicionado. Isto é, durante as repetições da rotina precedente, a posição inicial do pivô deve avançar do segundo para o terceiro elemento, depois deste para o quarto e assim por diante até que a rotina tenha posicionado o último elemento da lista. Segundo este raciocínio, podemos controlar a repetição desejada com as instruções:

N ← 2;
enquanto (o valor de N não exceder o comprimento da lista) **faça**
 (Selecionar o N-ésimo elemento da lista como pivô;
 .
 .
 .
 N ← N + 1)

onde o “comprimento da lista” se refere ao número de elementos da lista e os pontos indicam o lugar onde a rotina precedente deve ser colocada.

Nosso programa completo em pseudocódigo é mostrado na Figura 4.11. Em síntese, o programa ordena uma lista ao remover repetidamente um elemento da lista e inseri-lo em seu devido lugar. Por causa desse processo repetitivo de inserção é que o algoritmo subjacente é chamado **ordenação por inserção**.

Note-se que a estrutura da Figura 4.11 comprehende duas iterações aninhadas, sendo que a mais externa é expressa pela primeira instrução enquanto e a mais interna, pela segunda. Cada execução do corpo da iteração mais externa aciona repetidamente a iniciação e a execução da iteração mais interna,

procedimento Ordena (Lista)
 N ← 2;
enquanto (o valor de N não exceder o comprimento da lista) **faça**
 (Selecionar o N-ésimo elemento como pivô;
 Mover o pivô para uma posição temporária, criando uma posição livre na lista;
enquanto (existir um nome acima da posição livre e ele for maior que o pivô) **faça**
 (mover o nome acima da posição livre para ela, criando nova posição livre acima do nome)
 Mover o pivô para a posição livre da lista;
 N ← N + 1
)

Figura 4.11 O algoritmo de ordenação por inserção expresso em pseudocódigo.

até que sua condição terminal seja satisfeita. Assim, a cada execução do corpo da iteração mais externa, são efetuadas diversas execuções do corpo da iteração mais interna.

O componente de iniciação do controle do laço mais externo consiste em estabelecer o valor inicial de N com a instrução

$N \leftarrow 2$

O componente de modificação é manipulado incrementando N ao fim do corpo do laço com a instrução

$N \leftarrow N + 1$

A condição terminal ocorre quando o valor de N excede o comprimento da lista.

O controle do laço interno é iniciado com a remoção do pivô da lista, criando assim uma posição livre. O passo de modificação do laço é feito movendo-se os elementos para baixo, o que causa a ascensão da posição livre. A condição terminal consiste em a posição livre estar imediatamente abaixo de um nome que não seja maior que o pivô, ou em ela atingir o topo da lista.



QUESTÕES/EXERCÍCIOS

1. Modifique o programa de busca seqüencial da Figura 4.6 para que possa ser aplicado a listas não-ordenadas.
2. Converta a rotina a seguir, denotada em pseudocódigo:
 $Z \leftarrow 0;$
 $X \leftarrow 1;$
enquanto ($X < 6$) **faz**
 $(Z \leftarrow Z + X;$
 $X \leftarrow X + 1)$
 em uma rotina equivalente, usando a instrução *repete*.
3. Algumas das linguagens de programação populares atuais usam a sintaxe
`while (...) do (...)`
 para representar um laço pré-teste, e a sintaxe
`do (...) while (...)`
 para representar um laço pós-teste. Embora elegantes no projeto, que problemas podem resultar dessa similaridade?
4. Suponha que a ordenação por inserção, apresentada na Figura 4.11, seja aplicada à lista George, Cheryl, Alice e Bob. Descreva a configuração da lista ao final de cada execução do corpo da estrutura enquanto mais externa.
5. Por que não podemos, na instrução *enquanto* mais interna da Figura 4.11, substituir a expressão maior que por igual ou maior que?
6. Uma variação do algoritmo de ordenação por inserção é a **ordenação por seleção**. Ela inicia selecionando o menor elemento da lista e movendo-o para a frente. Então seleciona o menor dos nomes restantes e move para a segunda posição da lista. Ao selecionar repetidamente o menor elemento da posição restante da lista e movê-lo para a frente, a versão ordenada da lista cresce a partir do início da lista, enquanto a parte final, que consiste nos elementos ainda desordenados, decresce. Use o nosso pseudocódigo para expressar um procedimento similar ao da Figura 4.11, que ordene uma lista usando o algoritmo da ordenação por seleção.
7. Outro algoritmo de ordenação bem conhecido é o **método da bolha**. Ele é baseado no processo de comparar repetidamente dois nomes adjacentes e trocá-los de posição se não

estiverem na ordem desejada. Suponhamos que a lista a ser ordenada contenha n elementos. O método da bolha inicia comparando (e possivelmente trocando de posição) os elementos nas posições n e $n - 1$. Então, considera os elementos nas posições $n - 1$ e $n - 2$ e continua movendo para a frente na lista até que o primeiro e o segundo elementos da lista sejam comparados (e possivelmente trocados de posição). Observe que esse passo irá empurrar o menor elemento da lista para a frente. Do mesmo modo, um novo passo garantirá que o próximo menor elemento da lista será empurrado para a segunda posição. Assim, perfazendo-se um total de $n - 1$ passos na lista, ela ficará ordenada. (Se alguém pudesse ver o algoritmo trabalhando, observaria os elementos menores borbulhando em direção ao topo da lista — observação esta que dá nome ao algoritmo). Use o nosso pseudocódigo para expressar um procedimento similar ao da Figura 4.11, que ordene uma lista usando o método da bolha.

4.5 Estruturas recursivas

As estruturas recursivas fornecem às estruturas repetitivas uma alternativa ao paradigma de laço. Enquanto um laço envolve a repetição de um conjunto de instruções de modo que o conjunto seja completado e então repetido, a recursão envolve a repetição do conjunto de instruções como uma subtarefa de si própria. Um exemplo é encontrado no processamento de chamadas telefônicas que utiliza uma chamada em espera. Ali, uma conversação telefônica incompleta é deixada de lado temporariamente enquanto outra chamada é atendida. O resultado é que duas conversações são realizadas. Contudo, elas não são feitas uma atrás da outra, como em uma estrutura de laço, mas em vez disso, uma é feita dentro da outra.

Como introdução à recursão, consideremos o algoritmo de **busca binária** que aplica a metodologia dividir para conquistar ao processo de busca.

Algoritmo de busca binária

Suponhamos novamente o problema da busca de um dado elemento em uma lista ordenada, mas desta vez usando como ponto de partida o mesmo método que aplicamos normalmente para encontrar uma palavra em um dicionário. Não fazemos a procura seqüencialmente, palavra por palavra, nem mesmo procuramos página por página. Ao contrário, abrimos o dicionário em uma página qualquer, dentro de uma região em que provavelmente estará a palavra desejada. Se tivermos sorte, localizaremos de imediato a posição correta; caso contrário, será necessário continuar procurando. Contudo, neste ponto, estreitamos a busca consideravelmente.

Obviamente, no caso de pesquisar um dicionário, temos conhecimento prévio da parte em que as palavras provavelmente serão encontradas. Se estamos procurando a palavra *sonambulismo*, começamos abrindo na parte final do dicionário. No caso de listas genéricas, contudo, não temos esta vantagem. Assim, concordamos em sempre iniciar a nossa busca no “meio”. Colocamos o termo *meio* entre aspas para indicar a possibilidade de uma lista com um número par de elementos não ter meio. Neste caso, o elemento central se refere ao primeiro elemento na segunda metade da lista.

Se o elemento do meio da lista for o procurado, poderemos declarar que a busca foi bem-sucedida. Caso

Busca e ordenação

Os algoritmos de busca seqüencial e binária são apenas dois de muitos algoritmos para realizar o processo de busca. De modo semelhante, a ordenação por inserção é apenas um de muitos algoritmos de ordenação. Outros algoritmos clássicos de ordenação incluem a ordenação por intercalação (discutida no Capítulo 11), a ordenação por seleção (Questão / Exercício 5 na Seção 4.4), o método da bolha (Questão / Exercício 6 na Seção 4.4), a ordenação rápida (*QuickSort*), que aplica a abordagem de dividir para conquistar ao processo de ordenação, e a ordenação no monte (*HeapSort*) que usa uma técnica inteligente para encontrar os elementos que devem ser movidos para o início da lista. Você encontrará discussões a respeito desses algoritmos nos livros listados em *Leituras Adicionais* no fim deste capítulo.

contrário, poderemos ao menos restringir o processo de busca à primeira ou à última metade da lista, dependendo de o valor procurado ser menor ou maior que o elemento do meio. (Lembre-se de que a lista está ordenada.)

Para pesquisar o restante da lista, poderíamos aplicar a busca seqüencial, mas em vez disso, apliquemos à parte da lista o mesmo método que foi usado para a lista inteira, ou seja, selecionamos o elemento do meio na parte restante da lista como o próximo a ser considerado. Como antes, se ele for o procurado, terminamos a busca. Caso contrário, restringiremos nossa pesquisa a uma parte ainda menor da lista.

Essa abordagem ao processo de busca é ilustrada na Figura 4.12, na qual consideramos a tarefa de pesquisar a lista à esquerda da figura à procura do elemento John. Inicialmente, consideremos o elemento do meio, Harry. Uma vez que o elemento procurado é maior, a pesquisa continua considerando a metade inferior da lista original. O elemento do meio da lista é Larry. Como o procurado precede Larry, voltamos nossa atenção à primeira metade da sublistas corrente. Quando interroga-mos o elemento do meio dessa sublistas secundária, encontramos o procurado, John, e declaramos que a pesquisa foi bem-sucedida. Em síntese, nossa estratégia é dividir a lista em questão sucessivamente em segmentos menores até que o elemento procurado seja encontrado ou que a pesquisa seja reduzida a um segmento vazio.

É preciso enfatizar este último ponto. Se o elemento procurado não constar na lista original, nosso método de pesquisa irá prosseguir dividindo a lista em segmentos menores até que o segmento a ser considerado esteja vazio. Nesse ponto, o nosso algoritmo deverá reconhecer que a pesquisa fracassou.

A Figura 4.13 é um primeiro esboço de nossos pensamentos usando o nosso pseudocódigo. Ele indica que devemos começar uma pesquisa testando se a lista está vazia. Nesse caso, devemos declarar que a pesquisa fracassou. Caso contrário, consideraremos o elemento do meio da lista. Se ele não for o procurado, pesquisaremos na primeira ou na segunda metade da lista. As duas possibilidades requerem uma pesquisa

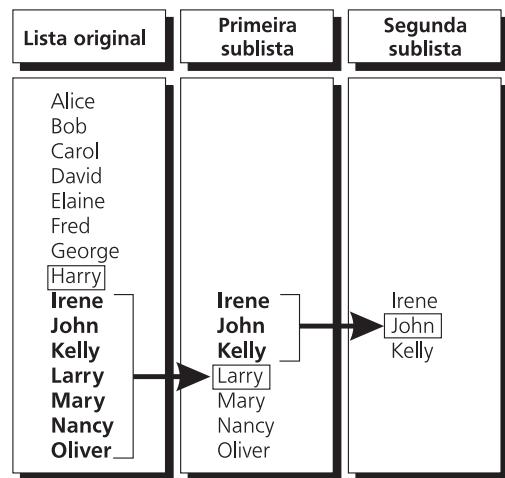


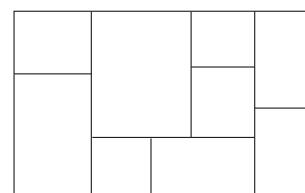
Figura 4.12 Aplicação da nossa estratégia para pesquisar a lista à procura do elemento John.

Estruturas recursivas na arte

O seguinte procedimento recursivo pode ser aplicado a uma tela retangular para produzir desenhos no estilo do pintor holandês Piet Mondrian (1872-1944), que fazia quadros nos quais o painel retangular era dividido sucessivamente em retângulos menores. Tente seguir o procedimento para produzir desenhos similares ao mostrado à direita. Comece aplicando o procedimento a um retângulo que represente a tela na qual você está trabalhando.

```

procedimento Mondrian (Retângulo)
se (o tamanho do Retângulo for muito grande para o seu gosto artístico)
  então (dividir o Retângulo em dois retângulos menores;
    aplicar o procedimento Mondrian a um dos retângulos menores;
    aplicar o procedimento Mondrian ao outro retângulo menor)
  
```



```

se (Lista vazia)
então
    (Declarar que a pesquisa fracassou.)
senão
    [Selecionar o elemento do “meio” da lista como elemento em teste;
    Executar o bloco de instruções abaixo que for apropriado ao caso:
        caso 1: ValorProcurado = Elemento em teste
            (Declarar que a pesquisa foi bem-sucedida)
        caso 2: ValorProcurado < Elemento em teste
            (Pesquisar a parte da lista que precede o elemento em teste
            em busca do ValorProcurado e declarar o resultado dessa pesquisa.)
        caso 3: ValorProcurado > Elemento em teste
            (Pesquisar a parte da lista que sucede o elemento em teste
            em busca do ValorProcurado e declarar o resultado dessa pesquisa.)
] fim do se

```

Figura 4.13 Um primeiro esboço da técnica de busca binária.

considerando o elemento do meio dessa lista. Assim, podemos fornecer o procedimento Busca e inserindo referências àquele procedimento, em que as pesquisas secundárias são necessárias. O resultado é mostrado na Figura 4.14.

Note que esse procedimento contém uma referência a si próprio. De fato, se o seguirmos e chegarmos à instrução Aplicar o procedimento Busca..., aplicaremos à lista menor o mesmo procedimento que estávamos aplicando à lista original. Se esta busca for bem-sucedida, retornaremos declarando o sucesso de nossa busca original; se a pesquisa secundária fracassar, retornaremos declarando que a busca original fracassou.

Para ver como o procedimento da Figura 4.14 realiza a sua tarefa, vamos segui-lo na busca da lista Alice, Bill, Carol, David, Evelyn, Fred e George, sendo Bill o elemento desejado. Nossa busca começa pela seleção de David (o elemento central da lista) como o elemento a testar. Como o valor desejado (Bill) é menor que o elemento em teste, será necessário aplicar o procedimento Busca à lista de elementos que

precedem David, ou seja: Alice, Bill e Carol. Assim, criamos uma segunda cópia do procedimento de busca, à qual será atribuída esta tarefa secundária.

Durante algum tempo, teremos em execução duas cópias deste nosso procedimento de busca, conforme mostra a Figura 4.15. A evolução da execução da instância original fica suspensa temporariamente no ponto em que se encontra a instrução

```

procedimento Busca (Lista, ValorProcurado)
se (Lista vazia)
então
    (Declarar que a pesquisa fracassou.)
senão
    [Selecionar o elemento do “meio” da lista como elemento em teste;
    Executar o bloco de instruções abaixo que for apropriado ao caso:
        caso 1: ValorProcurado = Elemento em teste
            (Declarar que a pesquisa foi bem-sucedida)
        caso 2: ValorProcurado < Elemento em teste
            (Pesquisar a parte da lista que precede o elemento em teste em
            busca do ValorProcurado e declarar o resultado dessa pesquisa.)
        caso 3: ValorProcurado > Elemento em teste
            (Pesquisar a parte da lista que sucede o elemento em teste em
            busca do ValorProcurado e declarar o resultado dessa pesquisa.)
] fim do se

```

Figura 4.14 O algoritmo de busca binária em pseudocódigo.

secundária. Seria interessante realizar essas pesquisas solicitando os serviços de uma ferramenta abstrata. Mais especificamente, nossa abordagem é aplicar um procedimento chamado Busca para efetuar essa pesquisa secundária. Portanto, para completar o nosso programa, precisamos dispor desse procedimento.

Entretanto, esse procedimento deve executar a mesma tarefa expressa pelo pseudocódigo que acabamos de escrever. Ele deve primeiramente conferir se a lista que lhe foi dada está vazia. E se não estiver, ele deverá prosseguir

para ver como o procedimento da Figura 4.14 realiza a sua tarefa, vamos segui-lo na busca da lista Alice, Bill, Carol, David, Evelyn, Fred e George, sendo Bill o elemento desejado. Nossa busca começa pela seleção de David (o elemento central da lista) como o elemento a testar. Como o valor desejado (Bill) é menor que o elemento em teste, será necessário aplicar o procedimento Busca à lista de elementos que

precedem David, ou seja: Alice, Bill e Carol. Assim, criamos uma segunda cópia do procedimento de busca, à qual será atribuída esta tarefa secundária.

Aplicar o procedimento Busca para verificar se ValorProcurado está na parte da Lista que precede o elemento em teste

enquanto aplicamos à lista Alice, Bill e Carol a segunda cópia do

procedimento de busca. Completada a busca secundária, descartamos a segunda cópia, informamos seus resultados à cópia original e continuamos sua execução. Deste modo, a segunda cópia do procedimento funciona como uma rotina subordinada à original, que desaparece após a execução da tarefa solicitada pelo módulo original.

A busca secundária seleciona Bill como seu elemento de teste, pois é o elemento central da lista Alice, Bill e Carol. Por ser o próprio elemento desejado, a busca é considerada bem-sucedida, e o procedimento termina sua execução.

Neste momento, após a conclusão da busca secundária solicitada pela instância original do procedimento de busca, esta retoma a sua execução. Sabemos que, se a busca secundária tiver êxito, devemos informar que a busca original foi um sucesso. Nossa processos determinou corretamente que Bill é um elemento integrante da lista Alice, Bill, Carol, David, Evelyn, Fred e George.

Verifiquemos o que ocorrerá se solicitarmos à rotina da Figura 4.14 que efetue na lista Alice, Carol, Evelyn, Fred e George a busca do elemento David. Desta vez, a cópia original do procedimento seleciona Evelyn como o elemento a testar e conclui que o elemento desejado deve se encontrar na parte superior da lista. Aplica, então, a uma segunda cópia do procedimento de busca, a lista composta dos elementos que figuram antes de Evelyn — isto é, a lista Alice e Carol. Nesse ponto, a situação está representada na Figura 4.16.

A segunda cópia do procedimento seleciona Carol como o elemento em teste e conclui que o nome desejado deve se encontrar na parte inferior da lista. Aplica, então, a uma terceira cópia do procedimento à lista de nomes que seguem a Carol na lista Alice e Carol. Tal sublista é vazia, e a terceira cópia teria de encontrar nela o elemento David. A situação, neste estágio, está representada na Figura 4.17. A cópia original do procedimento terá de localizar Evelyn, seu elemento a testar, dentro da lista Alice, Carol, Evelyn, Fred e George; a segunda cópia, por sua vez, deverá localizar o seu elemento a testar, Carol, na lista Alice e Carol; a terceira cópia efetuará a sua busca na lista vazia.

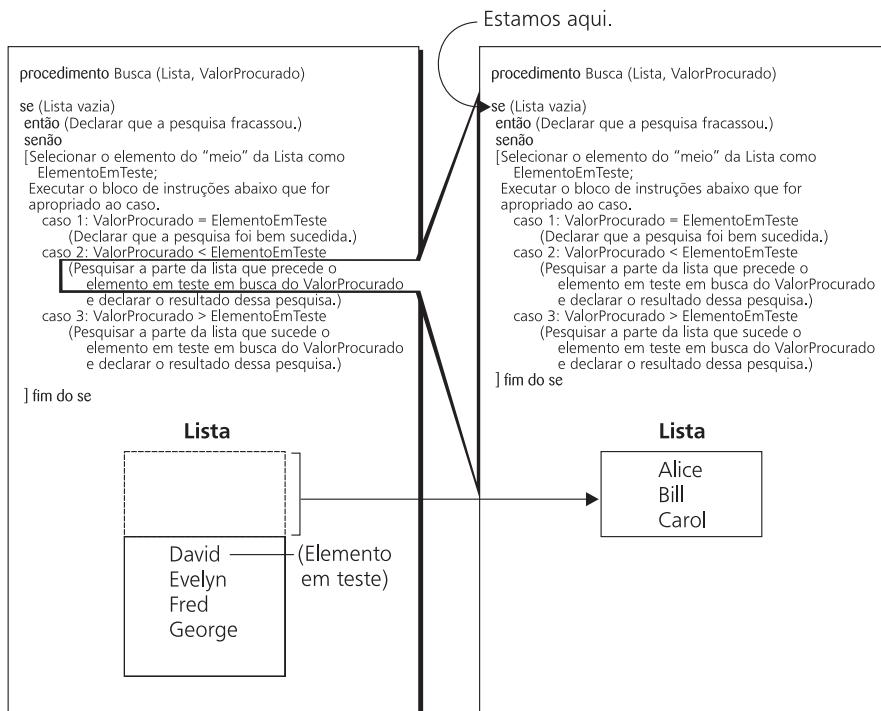


Figura 4.15

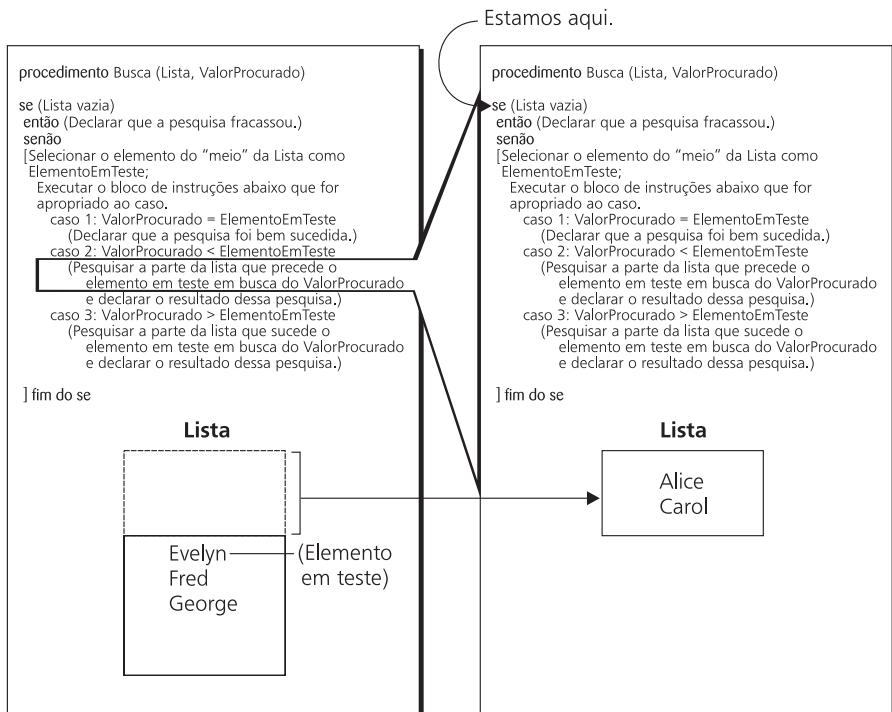


Figura 4.16

É evidente que a terceira cópia do procedimento declarará imediatamente que a sua busca fracassou, de modo que terminará rapidamente. Seu término permitirá que a segunda cópia prossiga em sua tarefa, logo percebendo que a busca solicitada resultou em fracasso. Isso a fará terminar em seguida, declarando também fracassada a sua própria tarefa. Este resultado, que havia sido solicitado pela cópia original do procedimento, também levará esta cópia a terminar, declarando, por sua vez, o fracasso da busca original. Assim, a nossa rotina concluirá corretamente que o elemento David não está contido na lista Alice, Carol, Evelyn, Fred e George.

Em resumo, se observarmos os exemplos anteriores, veremos que o processo empregado pelo algoritmo representado na Figura 4.14 consiste simplesmente em dividir repetidamente a lista em duas partes menores, de forma que o restante da busca se restrinja a apenas uma destas partes. Este método de bipartição é a razão pela qual o algoritmo é conhecido como busca binária.

Controle recursivo

O algoritmo da busca binária é semelhante ao da busca seqüencial porque ambos executam um processamento repetitivo. Todavia, a implementação da repetição é bem diferente. Enquanto a busca seqüencial trabalha com uma forma circular de repetição, a busca binária executa cada fase da repetição na forma de uma subtarefa da fase anterior. Esta técnica é conhecida como **recursão**.

Como vimos, a ilusão criada pela execução de um algoritmo recursivo é a existência de múltiplas réplicas dele mesmo, denominadas *ativações*, que aparecem e desaparecem no desenrolar do algoritmo. Das ativações existentes em um determinado momento, apenas uma permanece em estado de execução contínua. As demais permanecem dormentes, cada uma esperando pelo término de alguma outra para poder continuar.

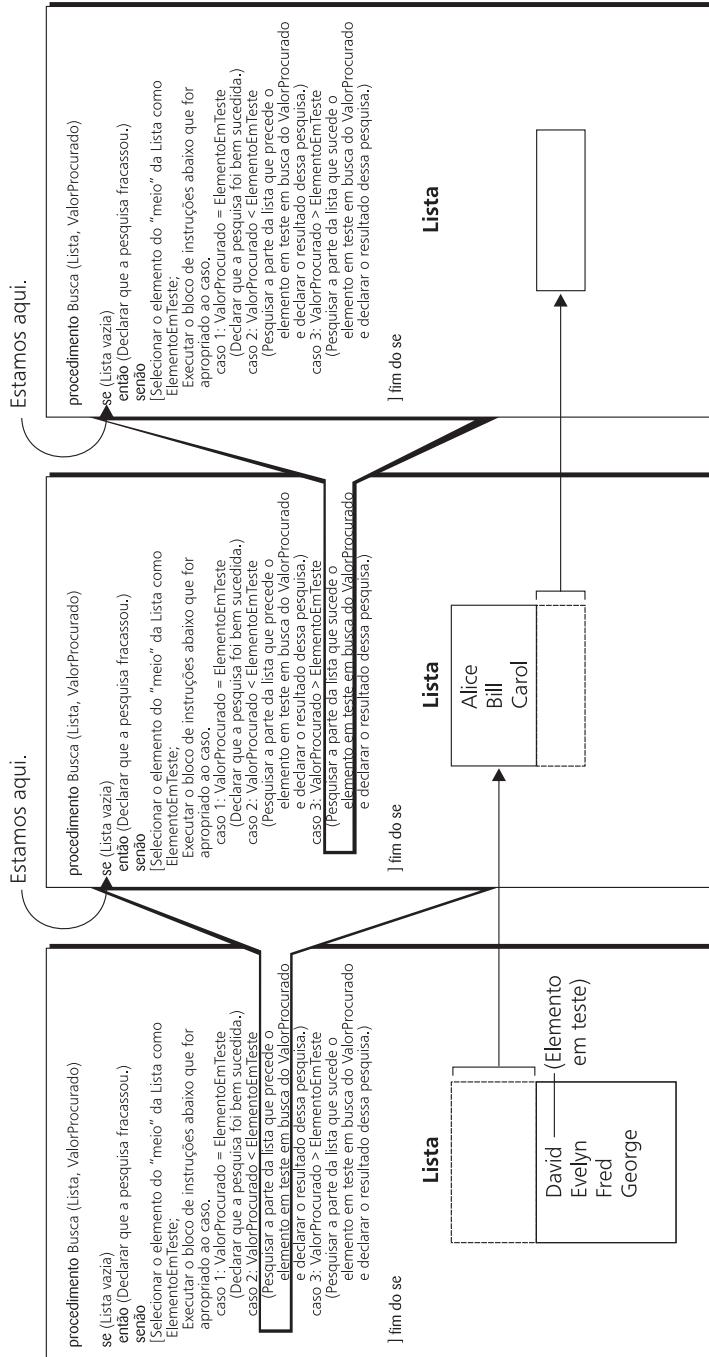


Figura 4.17

Por serem processos repetitivos, os sistemas recursivos, a exemplo das estruturas iterativas, também dependem muito de um controle adequado. Da mesma forma que ocorre no controle iterativo, os sistemas recursivos dependem de um teste de condição de terminação e de uma conduta de projeto que assegure que esta condição será de fato alcançada. Um controle recursivo adequado envolve, portanto, os mesmos três elementos — iniciação, modificação e teste de terminação — estudados no controle de laços.

Em geral, uma rotina recursiva é projetada para testar sua condição de terminação (freqüentemente denominada caso básico ou degenerativo) antes de solicitar ativações adicionais. Se esta condição não for atendida, o procedimento promoverá a execução de uma nova ativação, para solucionar uma versão revisada do problema que a ativação corrente enfrenta, que esteja mais próxima da condição de terminação. No entanto, caso a condição de terminação tenha sido atingida, adota-se uma solução que não dependa de ações recursivas adicionais, forçando a finalização da ativação atual sem promover ações recursivas adicionais.

Vejamos de que forma as fases de iniciação e de modificação do controle da repetição foram implementadas na nossa rotina recursiva de busca binária da Figura 4.14. Neste caso, a criação de ativações adicionais cessa ou ao ser encontrado o elemento desejado ou quando a tarefa se reduz a uma busca em lista vazia. O processo começa, implicitamente, com uma lista inicial e um elemento a ser localizado. A partir daí, a rotina modifica a tarefa inicial, reduzindo-a a uma outra busca, em uma lista menor. Dado que a lista original é de comprimento finito e diminui a cada etapa de modificação, podemos ficar seguros de que o elemento desejado será encontrado ou, então, que a tarefa será reduzida à busca em uma lista vazia. Concluímos, assim, que esse processo repetitivo é certamente finito.

Tendo observado o processo de controle tanto das estruturas iterativas como das recursivas, você desejará saber se ambas são equivalentes, em matéria de poder computacional. Isto é, se um algoritmo foi projetado com uma estrutura de laço, um outro algoritmo que use apenas técnicas recursivas pode ser projetado para resolver o mesmo problema e vice-versa? Tais perguntas são importantes em Ciência da Computação, pois suas respostas fornecem informações sobre as características necessárias em uma linguagem de programação, para que se possa obter o sistema de programação mais poderoso possível. Retornaremos a este tema no Capítulo 11, onde consideraremos alguns aspectos mais teóricos da Ciência da Computação e seus fundamentos matemáticos. Com tais idéias em mente, podemos provar, no Apêndice E, a equivalência entre estruturas iterativas e recursivas.



QUESTÕES/EXERCÍCIOS

1. Quais nomes serão consultados pelo algoritmo de busca binária (Figura 4.14) quando este estiver buscando o nome Joe na lista: Alice, Bob, Carol, David, Evelyn, Fred, George, Henry, Irene, Joe, Karl, Larry, Mary, Nancy e Oliver?
2. Qual o número máximo de elementos a serem consultados quando se utiliza o algoritmo de busca binária, para uma lista de 200 elementos? E para 100.000 elementos?

4.6 Eficiência e correção

Dos tópicos restantes que poderiam ser discutidos como parte da nossa introdução aos algoritmos, dois conceitos são tratados nesta seção, cobrindo problemas que vêm à mente quando alguém desenvolve seus próprios programas. O primeiro destes é a eficiência, e o outro, a correção* dos algoritmos.

*N. de T. Em inglês, *correctness* — neste contexto, a palavra *correção* está sendo empregada no sentido de *qualidade de correto*.

Eficiência de algoritmos

Embora as máquinas modernas sejam capazes de executar milhões de instruções por segundo, a eficiência continua sendo uma preocupação central no projeto de algoritmos. Geralmente a escolha entre um algoritmo eficiente e um ineficiente pode significar, para um dado problema, a diferença entre uma solução prática ou não.

Consideremos o problema de matrícula universitária, que necessita localizar e atualizar registros de estudantes. Embora a universidade tenha, de fato, cerca de 10.000 estudantes em cada semestre, seu “arquivo corrente de estudantes” contém o registro de mais de 30.000, que serão considerados estudantes correntes se tiverem feito matrícula em pelo menos um curso nos últimos anos, mas que não concluíram uma etapa inteira do curso. Podemos imaginar estes registros armazenados no computador da seção de alunos, em uma lista ordenada segundo o número de identificação dos estudantes. Para encontrar qualquer registro de estudante, o encarregado irá essencialmente procurar nesta lista, ordenada de acordo com o número de identificação do estudante.

Foram apresentados dois algoritmos de busca para tal lista: a busca seqüencial e a binária. A questão agora é decidir se uma escolha entre estes dois algoritmos faria alguma diferença no caso do arquivo de estudantes. Consideremos inicialmente a busca seqüencial.

Dado um número de identificação de estudante, o algoritmo de busca seqüencial inicia a busca no começo da lista e compara um a um com o número desejado os elementos da lista. Se não houver informação acerca do valor procurado, nada concluirímos sobre a extensão da sua busca nesta lista. Entretanto, podemos dizer que, depois de muitas buscas, a extensão média da busca deverá ser aproximadamente a metade da extensão total da lista; algumas serão menores; outras, mais longas. Concluímos que, dentro de um intervalo de tempo, a busca seqüencial testará em média uns 15.000 registros por busca. Se para obter e conferir um número de identificação de um registro for necessário 10 milissegundos (dez milésimos de segundo) de processamento, esta busca se completará, em média, em 150 segundos ou 2,5 minutos — tempo intolerável para o encarregado esperar o registro do estudante aparecer na tela. Mesmo se o tempo para recuperar e conferir cada registro fosse reduzido para apenas 1 milissegundo, a busca ainda exigiria em média 15 segundos, o que é ainda um tempo de espera longo.

A busca binária, porém, opera pela comparação do valor desejado com o elemento que se encontra no meio da lista. Se este não for o elemento desejado, então, pelo menos, o restante da busca se restringirá à metade da lista original. Assim, depois de testar o elemento central de uma lista de 30.000 registros de estudantes, a busca binária ainda deverá verificar no máximo 15.000 registros. Após o segundo teste, permanecem no máximo 7.500, e depois da terceira tentativa, a lista em questão estará reduzida a não mais que 3.750 elementos. Continuando desta forma, verificamos que, se o registro desejado estiver na lista, será encontrado depois do teste de, no máximo, 15 dos 30.000 registros da lista. Assim, se cada tentativa for executada em 10 milissegundos, a busca de um dado registro, usando esse algoritmo, exigirá somente 0,15 segundos — o que significa que o acesso a um registro de estudante específico parecerá instantâneo ao encarregado. Concluímos que a escolha entre o algoritmo de busca seqüencial e o de busca binária terá um impacto significativo nessa aplicação.¹

Esse exemplo mostra a importância da área da Ciéncia da Computação conhecida como análise de algoritmos, que inclui o estudo de recursos como o tempo e espaço de armazenamento que os algoritmos necessitam. Uma aplicação primordial desses estudos é a avaliação dos méritos relativos de algoritmos alternativos. Em nosso caso, analisamos o tempo exigido pelos algoritmos de busca seqüencial e binária para determinar qual era a melhor solução em uma aplicação específica. Em geral, essa análise é realizada em um contexto mais genérico, isto é, quando consideramos algoritmos para pesquisar listas, não focalizamos uma lista particular de comprimento determinado, mas tentamos identificar uma fórmula

¹Para obter os benefícios do algoritmo de busca binária, os registros de estudante devem ser armazenados de maneira a permitir que os elementos do meio das sublistas possam ser recuperados sem muito trabalho. Estudaremos como fazer isso nos Capítulos 7 e 8.

que indique o desempenho do algoritmo em listas de comprimento arbitrário. Esse tipo de análise geralmente envolve a identificação do melhor caso, do pior e do médio.

Nossa análise prévia focalizou o desempenho médio do algoritmo de busca seqüencial e o pior caso de algoritmo de busca binária. Embora tenhamos nos concentrado em uma lista de comprimento determinado, não é difícil generalizar nosso raciocínio para listas de comprimento arbitrário. Em particular, quando aplicado a uma lista com n elementos, o algoritmo de busca seqüencial irá interrogar em média $n/2$ elementos, enquanto o de busca binária consultará no máximo $\lg n$ elementos, na pior das hipóteses. ($\lg n$ representa o logaritmo de n na base dois.)

Vamos analisar agora o algoritmo de ordenação por inserção (resumido na Figura 4.11) de uma maneira similar. Lembre-se de que esse algoritmo envolve a seleção de um elemento chamado pivô, a sua comparação com aqueles que o precedem até o seu lugar adequado ser encontrado, e então a sua inserção. Uma vez que a atividade de comparar dois nomes predomina no algoritmo, nossa abordagem será a de encontrar o número de comparações feitas quando se ordena uma lista de comprimento n .

O algoritmo inicia selecionando o segundo elemento da lista como pivô e então progride, tomando os elementos sucessivos como pivôs, até que seja alcançado o fim da lista. No melhor caso, cada pivô já está em seu lugar e, assim, necessita ser comparado com apenas um nome para que isso possa ser constatado. Portanto, no melhor caso, a aplicação da ordenação por inserção a uma lista de n elementos requer $n - 1$ comparações. (O segundo elemento é comparado com um nome, o terceiro, com um nome, e assim por diante.)

Entretanto, o pior caso é aquele no qual cada pivô deve ser comparado com todos os elementos precedentes para que seu lugar adequado possa ser encontrado. Isso ocorrerá se a lista original estiver na ordem inversa à desejada. Nesse caso, o primeiro pivô (segundo elemento da lista) é comparado com um nome, o segundo pivô (terceiro elemento da lista), com dois nomes e assim por diante (Figura 4.18). Portanto, o número total de comparações ao ordenar uma lista com n elementos é $1 + 2 + 3 + \dots + (n - 1)$, que equivale a $n(n - 1)/2$ ou $(1/2)(n^2 - n)$. Por exemplo, se a lista contiver 10 elementos, o pior caso para o algoritmo de ordenação por inserção exigirá 45 comparações.

No caso médio da ordenação por inserção, devemos esperar que cada pivô seja comparado com a metade dos elementos precedentes. Isso resulta na metade das comparações realizadas no pior caso, um total de $(1/4)(n^2 - n)$ comparações para ordenar uma lista com n nomes. Se, por exemplo, usarmos a ordenação por inserção para ordenar várias listas de comprimento 10, deveremos esperar que o número médio de comparações seja 22,5.

O significado desse resultado é que o número de comparações feitas durante a execução do algoritmo de ordenação por inserção dá uma aproximação do tempo necessário para executar o algoritmo. Usando essa aproximação, a Figura 4.19 mostra um gráfico que indica como o tempo exigido para executar o algoritmo de ordenação por inserção cresce em função do comprimento da lista. Esse gráfico se baseia em nossa análise do pior caso do algoritmo, na qual concluímos que ordenar uma lista de comprimento n exige no máximo $(1/2)(n^2 - n)$ comparações entre seus elementos. No gráfico, marcamos vários comprimentos de lista e indicamos o tempo necessário em cada caso. Note que à medida que os comprimentos crescem com incrementos uniformes, o tempo exigido para ordenar a lista cresce com incremen-



Figura 4.18
Aplicação da ordenação por inserção no pior caso.

tos cada vez maiores. Assim, o algoritmo vai se tornando menos eficiente à medida que o tamanho da lista aumenta.

Vamos aplicar uma análise similar ao algoritmo de busca binária. Lembre-se de que concluímos que pesquisar uma lista com n elementos usando esse algoritmo envolve a consulta de, no máximo, $\lg n$ elementos, o que, mais uma vez, dá uma aproximação do tempo necessário para executar o algoritmo em listas de vários comprimentos. A Figura 4.20 mostra um gráfico baseado nessa análise, no qual novamente marcamos vários comprimentos de lista igualmente espaçados e identificamos o tempo gasto pelo algoritmo em cada caso. Note que o tempo aumenta com incrementos cada vez menores, isto é, o algoritmo de busca binária torna-se mais eficiente à medida que o tamanho da lista aumenta.

A característica de distinção entre as Figuras 4.19 e 4.20 é a forma geral das curvas envolvidas. É a forma geral de um gráfico, em vez das específicas, que revela o desempenho de um algoritmo à medida que seus dados de entrada crescem. Observe que a forma geral de um gráfico é determinada pelo tipo de expressão que está sendo desenhada em vez da expressão específica — todas as expressões lineares produzem uma linha reta; todas as quadráticas produzem uma parábola; todas as logarítmicas produzem a forma logarítmica mostrada na Figura 4.20. É costume identificar uma curva com a expressão mais simples que produz a sua forma. Em particular, identificamos a forma parabólica com a expressão n^2 e a logarítmica com a expressão $\lg n$.

Vimos que a forma do gráfico obtido na comparação do tempo exigido por um algoritmo na realização de sua

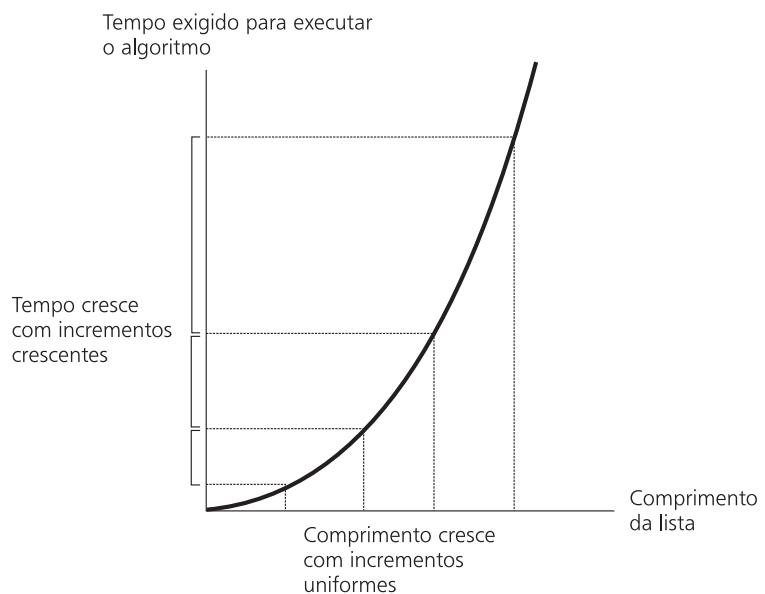


Figura 4.19 Gráfico de análise do pior caso do algoritmo de ordenação por inserção.

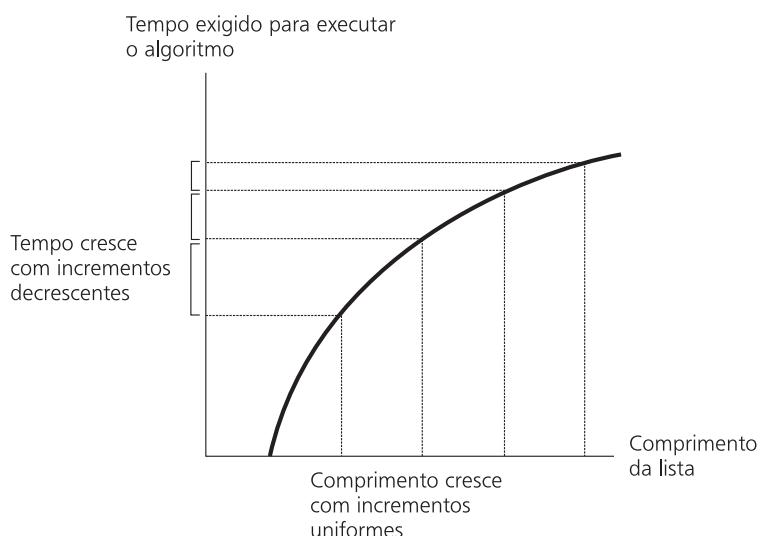


Figura 4.20 Gráfico de análise do pior caso do algoritmo de busca binária.

tarefa com o tamanho dos dados de entrada reflete as características de eficiência do algoritmo. Assim, é comum classificar os algoritmos de acordo com a forma desses gráficos — normalmente baseados em análise de pior caso. A notação usada para identificar essas classes é chamada “notação teta”. Todos os algoritmos cujo gráfico tem a forma de uma parábola, como o de ordenação por inserção, são incluídos na classe representada por $\Theta(n^2)$ (leia-se “teta de n ao quadrado”); todos os algoritmos cujo gráfico tem a forma de expressão logarítmica, como o da busca binária, são incluídos na classe identificada por $\Theta(\lg n)$ (leia-se “teta de $\log n$ ”). Sabendo-se a classe à qual um particular algoritmo pertence, podemos prever o seu desempenho e compará-lo com outros algoritmos para resolver o mesmo problema. Dois algoritmos em $\Theta(n^2)$ terão tempos similares quando o tamanho de suas entradas crescer, enquanto a exigência de tempo de um algoritmo em $\Theta(\lg n)$ não se expandirá tão rapidamente quanto a de um outro em $\Theta(n^2)$.

Verificação de software

Relembreamos que a quarta fase da análise de Polya sobre a resolução de problemas (Seção 4.3) trata da avaliação da solução quanto à sua precisão e ao seu potencial como ferramenta para a resolução de outros problemas. O significado da primeira parte desta fase pode ser captado no seguinte exemplo:

Um viajante, com uma corrente de ouro formada por uma cadeia de sete argolas, deve ficar por sete noites em um hotel isolado. O aluguel por uma noite equivale a uma argola da corrente. Qual o número mínimo de argolas que devem ser cortadas de modo que o viajante possa pagar ao hoteleiro uma argola a cada manhã, sem antecipar o pagamento pela hospedagem?

Primeiro, percebemos que nem toda argola da corrente deve ser cortada. Se cortarmos somente a segunda argola, poderíamos liberar a primeira e a segunda das outras cinco. Seguindo este raciocínio, a solução seria cortar a segunda, a quarta e a sexta argolas da corrente e teríamos um processo que solta todas as argolas cortando somente três (Figura 4.21). Além disso, qualquer número menor de cortes deixaria duas argolas conectadas, e assim concluímos que a resposta correta para o nosso problema é três.

Além da verificação de software

Os problemas de verificação, como discutido no texto, não são exclusividade do *software*. Igualmente importante é o problema de se confirmar que o *hardware* que executa um programa está livre de defeitos. Isso envolve a verificação do projeto dos circuitos, bem como da construção da máquina. Mais uma vez, o estado da arte conta muito com os testes, o que, como no caso do *software*, significa que erros sutis ainda podem ser encontrados nos produtos acabados. Os registros indicam que o Mark I, construído na Harvard University, na década de 1940, continha erros de fiação que não foram detectados em muitos anos de operação. Um exemplo mais recente foi um defeito na parte que trata os números reais nos primeiros processadores Pentium. Nos dois casos, o erro foi detectado antes que consequências mais graves tivessem ocorrido.

Entretanto, após reconsiderarmos o problema, observamos que quando cortamos somente a terceira argola da corrente, obtemos três pedaços de correntes de comprimentos iguais a um, dois e quatro (Figura 4.22). Com estes pedaços, podemos proceder da seguinte forma:

Na primeira manhã, entregar ao hoteleiro a única argola solta.

Na segunda, receber de volta esta argola e entregar ao hoteleiro a corrente de duas argolas.

Na terceira, entregar novamente a argola solta ao hoteleiro.

Na quarta, receber de volta as correntes que ficaram com o hoteleiro e lhe entregar a corrente de quatro argolas.

Na quinta, entregar novamente ao hoteleiro a argola solta.

Na sexta, receber esta argola de volta e entregar a corrente de duas argolas.

Na sétima, entregar novamente a argola solta.

Como consequência, a nossa primeira resposta, que acreditávamos estar correta, está errada. Então, como podemos estar seguros de que esta nova solução está correta? Podemos argumentar: Como uma argola isolada deve ser entregue ao hoteleiro na primeira manhã, isto significa que ao menos uma argola da cadeia deve ser cortada, e como a nossa nova solução exige apenas um corte, ela deverá ser ótima.

Do ponto de vista da programação, este exemplo enfatiza a diferença entre um programa que se supõe correto e um que está correto. Os dois programas não são necessariamente o mesmo. A comunidade de processamento de dados possui inúmeras histórias incríveis que envolvem *software* que, embora tido como correto, falhou em um momento crítico devido a alguma situação imprevista. A verificação de *software* torna-se, assim, uma tarefa importante, e a busca de métodos eficientes de verificação é uma área de pesquisa em plena atividade na Ciência da Computação.

Uma linha atual de pesquisa nesta área procura aplicar técnicas de lógica formal para provar a correção de um programa. Isto é, a meta é aplicar a lógica formal para provar que um algoritmo representado por um programa faz o que se propõe a fazer. A tese subjacente afirma que, reduzindo o processo de verificação a um procedimento formal, é possível proteger-se das conclusões imprecisas normalmente associadas aos argumentos intuitivos, como foi o caso do problema da corrente de ouro. Vamos considerar com maior cuidado a aplicação desta conduta à tarefa de verificação de programas.

Da mesma maneira que uma prova matemática formal se fundamenta em axiomas (as provas geométricas geralmente são baseadas nos axiomas da geometria euclidiana, enquanto outras se fundamentam nos axiomas da teoria de conjuntos), uma prova formal da correção de um programa baseia-se nas especificações segundo as quais o programa foi projetado. Para provar que um programa ordena listas de nomes corretamente partimos da suposição de que os dados de entrada constituem uma lista de nomes. Se o programa foi projetado para calcular a média de um ou mais números positivos, partimos da suposição de que os dados de entrada consistem em um conjunto de um ou mais números positivos. Em suma, uma prova de correção parte da suposição de que certas condições, denominadas **precondições**, estejam satisfeitas ao início da execução do programa.

O próximo passo da prova de correção é verificar como as consequências destas precondições se propagam pelo programa. Com esta finalidade, pesquisadores analisaram várias estruturas de programa para determinar de que modo uma afirmação, que se sabia ser verdadeira antes da execução da estrutura, é afetada ao executar a estrutura. Como ilustração simples, tendo-se a certeza de que uma dada afirmação que envolve o valor de Y é verdadeira, imediatamente antes da execução da instrução

$$X \leftarrow Y$$

então esta mesma afirmação poderá ser feita acerca de X , depois que a instrução for executada. Mais precisamente, se o valor de Y não for 0 antes da execução da instrução, então poderemos concluir que o de X não será 0 após a execução da instrução.

Um exemplo mais abrangente ocorre no caso de uma estrutura se-então-senão, como

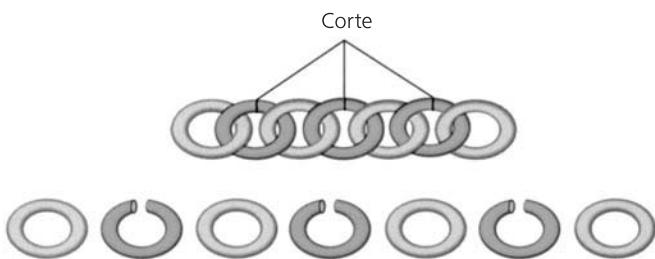


Figura 4.21 Separação de uma cadeia de argolas usando somente três cortes.

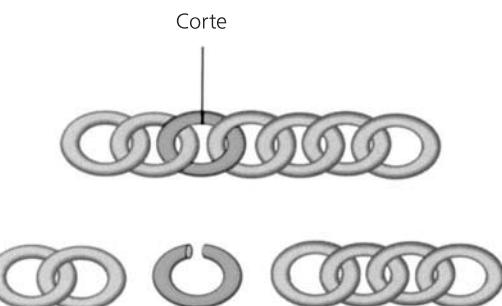


Figura 4.22 Resolução do problema com apenas um corte.

se (condição) então (instrução 1)
senão (instrução 2)

Aqui, se alguma afirmação for verdadeira antes da execução da estrutura, então, imediatamente antes de executar a *instrução 1*, saberemos que tanto esta afirmação como a condição de teste são verdadeiras, enquanto que, se a *instrução 2* for executada, saberemos que tanto a afirmação em questão como a negação da condição serão simultaneamente verdadeiras.

Seguindo regras similares a estas, uma prova da correção se dá com a identificação das afirmações, conhecidas como **asserções**, que podem ser estabelecidas em vários pontos no programa. O resultado será uma coleção de asserções, em que cada uma é consequência das precondições do programa e da seqüência de instruções que o conduzem aos pontos aos quais as asserções se referem. Concluiremos

que o programa está correto se, assim procedendo, determinarmos que a asserção correspondente ao seu final está de acordo com as especificações desejadas.

Por exemplo, considere a estrutura iterativa enquanto representada na Figura 4.23. Suponha que, como consequência das precondições fornecidas no ponto A, possamos estabelecer que uma dada asserção é verdadeira toda vez que o teste de terminação for executado (ponto B) durante o processo repetitivo. (Tal asserção, estabelecida para uma iteração, é conhecida como uma **invariante do laço**.) Então, terminada a iteração, a execução estará no ponto C, no qual concluiremos que tanto a invariante do laço quanto a condição de terminação são verdadeiras. (A invariante do laço continua verdadeira porque o teste de terminação não altera qualquer valor do programa, e a condição terminal é verdadeira porque, caso não fosse, a iteração não terminaria.) Se estas afirmações combinadas implicarem o resultado desejado, nossa prova de correção do programa poderá ser completada simplesmente mostrando que os componentes de iniciação e de modificação da iteração conduzem finalmente à condição terminal.

Podemos comparar esta análise com o nos-

so exemplo de ordenação por inserção, mostrado na Figura 4.11. O laço mais externo nesse programa se baseia na seguinte invariante do laço:

Cada vez que o teste de terminação é executado, os nomes que precedem o N-ésimo elemento constituem uma lista ordenada.

e também na condição terminal seguinte:

O valor de N é maior que o comprimento da lista.

Assim, se a iteração termina, sabemos que as duas condições são satisfeitas, o que implica o fato de a lista inteira estar ordenada.

Infelizmente, a verificação formal de programa não foi suficientemente refinada para ser utilizada com facilidade em aplicações gerais de processamento de dados. O resultado é que, na maioria dos casos práticos, o software é “verificado” por meio de testes sob várias condições — um processo extremamente

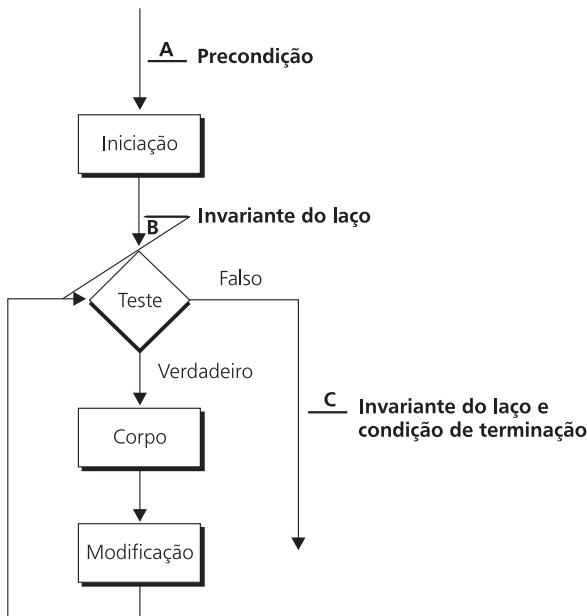


Figura 4.23 As asserções associadas com uma estrutura *enquanto* típica.

frágil. Afinal de contas, a verificação por testes nada prova, senão que o programa funciona corretamente para os dados de teste. Qualquer conclusão adicional é pura projeção. Os erros contidos em um programa muitas vezes são consequência de equívocos sutis facilmente escondidos também nos testes. Assim, erros em um programa, como o nosso no problema da corrente de ouro, podem ficar e freqüentemente ficam sem ser detectados, embora esforços significativos tenham sido feitos para evitá-los. Um exemplo drástico ocorreu na AT&T. Um erro no *software* que controlava 114 estações de chaveamento permaneceu sem ser detectado desde a instalação em dezembro de 1989 até 15 de janeiro de 1990, quando um conjunto único de circunstâncias causou o bloqueio desnecessário de aproximadamente cinco milhões de chamadas durante nove horas.



QUESTÕES/EXERCÍCIOS

- Suponha que uma máquina programada com o nosso algoritmo de ordenação por inserção necessite em média um segundo para ordenar uma lista de 100 nomes. Quanto tempo você estima que ela necessitará para ordenar uma lista de 1.000 nomes? E de 10.000?
- Dê um exemplo de um algoritmo pertencente a cada uma das seguintes classes: $\Theta(\lg n)$, $\Theta(n)$ e $\Theta(n^2)$.
- Liste as classes $\Theta(n^2)$, $\Theta(\lg n)$, $\Theta(n)$ e $\Theta(n^3)$ em ordem decrescente de eficiência.
- Considere o seguinte problema e uma sugestão de resposta. A resposta sugerida está correta? Justifique.

Problema: Suponha que uma caixa contenha três cartas. Uma delas é pintada de preto nas duas faces, outra de vermelho também nas duas faces, e a terceira carta é pintada de vermelho em uma das faces e de preto na outra. Uma delas é tirada da caixa e você é autorizado a ver uma de suas faces. Qual a probabilidade de a outra face da carta ser da mesma cor da face que você viu?

Resposta sugerida: A probabilidade é 50%. Suponha que a face da carta que você viu seja vermelha. (Este argumento seria idêntico para a carta cuja face é preta.) Somente duas cartas entre as três possuem uma face vermelha. Assim, a carta que você viu deve ser uma delas. Uma dessas duas cartas é vermelha na outra face, enquanto a outra é preta. Assim, a carta que você viu tanto pode ser vermelha na outra face quanto preta.

- O seguinte trecho de programa é uma tentativa de computar o quociente (despreze o resto da divisão) de dois inteiros positivos por meio da contagem do número de vezes que o divisor pode ser subtraído do dividendo antes que a diferença se torne menor que o divisor. Por exemplo, 7 dividido por 3 resulta em 2 porque 3 pode ser subtraído de 7 duas vezes. O programa abaixo está correto? Justifique sua resposta.

Contador $\leftarrow 0$;

Resto \leftarrow Dividendo:

repete (Resto \leftarrow Resto - Divisor;

 Contador \leftarrow Contador + 1)

até (Resto < Divisor)

Quociente \leftarrow Contador.

- O seguinte trecho de programa foi projetado para calcular o produto de dois inteiros não-negativos X e Y, acumulando a soma de X cópias de Y — por exemplo, 3 multiplicado por 4 é calculado acumulando a soma de três quatros. O programa está correto? Justifique sua resposta.

Produto $\leftarrow Y$;

Contador $\leftarrow 1$;

enquanto (Contador < X) **faça**

(Produto ← Produto + Y;
Contador ← Contador + 1)

7. Admitindo a precondição de que o valor associado a N seja um inteiro positivo, estabeleça uma invariante de laço que leve à conclusão de que, se a rotina seguinte terminar, então Soma assumirá o valor $0 + 1 + \dots + N$.
 $\text{Soma} \leftarrow 0;$
 $I \leftarrow 0;$
enquanto ($I < N$) **faça**
 $(I \leftarrow I + 1;$
 $\text{Soma} \leftarrow \text{Soma} + I)$
Forneça um argumento que prove que essa rotina realmente termina.
8. Suponha que tanto o programa quanto o *hardware* que o executa tenham sido formalmente verificados e considerados corretos. Isso garante a correção?

Problemas de revisão de capítulo

(Os problemas marcados com asteriscos se referem às seções opcionais.)

1. Dê um exemplo de um conjunto de passos que esteja de acordo com a definição informal de algoritmo apresentada no parágrafo inicial da Seção 4.1, mas que contrarie a definição mostrada na Figura 4.1.
2. Explique a diferença entre os conceitos de ambigüidade em um algoritmo proposto e de ambigüidade na representação de um algoritmo.
3. Descreva de que maneira o uso de primitivas ajuda a eliminar ambigüidades da representação de um algoritmo.
4. Escolha um assunto que lhe seja familiar e escreva um pseudocódigo para fornecer diretrizes em relação ao assunto. Descreva as primitivas que você usaria e a sintaxe para representá-las. (Se você está tendo dificuldade em pensar em um assunto, tente na área dos esportes, das artes ou dos artesanatos.)
5. O seguinte programa representa um algoritmo, no sentido estrito? Justifique a sua resposta.
 $\text{Contador} \leftarrow 0;$
enquanto ($\text{Contador} \neq 5$) **faça**
 $(\text{Contador} \leftarrow \text{Contador} + 2)$
6. Em que sentido os seguintes passos não constituem um algoritmo?

Passo 1: Desenhe um segmento de reta entre os pontos de coordenadas cartesianas (2,5) e (6,11).

- Passo 2: Desenhe um segmento de reta entre os pontos com coordenadas cartesianas (1, 3) e (3,6).
- Passo 3: Desenhe um círculo de raio dois, cujo centro esteja na intersecção dos dois segmentos traçados anteriormente.
7. Reescreva o seguinte trecho de programa usando a estrutura **repete** no lugar de **enquanto**. Certifique-se de que a nova versão imprima os mesmos valores impressos pela versão original.
 $\text{Contador} \leftarrow 2;$
enquanto ($\text{Contador} < 7$) **faça**
(imprimir o valor corrente de Contador e
 $\text{Contador} \leftarrow \text{Contador} + 1)$
8. Reescreva o seguinte trecho de programa utilizando uma estrutura **enquanto** em lugar de uma **repete**. Certifique-se de que a nova versão imprima os mesmos valores impressos pela versão original.
 $\text{Contador} \leftarrow 1;$
repete
(imprimir o valor corrente de Contador e
 $\text{Contador} \leftarrow \text{Contador} + 1)$
até ($\text{Contador} = 5$)
9. O que deve ser feito para traduzir um laço pós-teste expresso na forma
repete (...) **até** (...)
em um laço pós-teste equivalente expresso na forma
faça (...) **enquanto** (...)

- 10.** Projete um algoritmo que, recebendo uma sequência formada pelos dígitos 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9, rearranje os dígitos de tal modo que a nova sequência represente, com esses mesmos dígitos, um valor que seja imediatamente mais elevado que o inicialmente recebido (ou declare que tal sequência não existe, se for o caso). Assim, a partir de 5647382901, o algoritmo deve gerar 5647382910.

- 11.** Escreva um algoritmo para determinar o número de vezes que uma cadeia de caracteres aparece como subcadeia em outra cadeia. Por exemplo, *abc* aparece duas vezes na cadeia *abcdabc* e a cadeia *aba* aparece duas vezes na cadeia *ababa*. Note que *aa* aparece três vezes em *aaaa*.
- 12.** Faça um algoritmo para determinar o dia da semana para qualquer data a partir de 1º de janeiro de 1700. Por exemplo, 17 de agosto de 2001 caiu em uma sexta-feira.

- 13.** Qual é a diferença entre uma linguagem de programação formal e um pseudocódigo?
- 14.** Qual é a diferença entre sintaxe e semântica?
- 15.** A conta de somar abaixo está na notação tradicional de base dez. Cada letra representa um dígito diferente. Que dígito está sendo representado por cada letra? Como você subiu o primeiro degrau?

$$\begin{array}{r} \text{XYZ} \\ + \text{YWy} \\ \hline \text{ZYzW} \end{array}$$

- 16.** A conta de multiplicar abaixo está na notação tradicional de base dez. Cada letra representa um dígito diferente. Que dígito está sendo representado por cada letra? Como você subiu o primeiro degrau?

$$\begin{array}{r} \text{XY} \\ \times \text{YX} \\ \hline \text{XY} \\ \text{YZ} \\ \hline \text{WVY} \end{array}$$

- 17.** Quatro exploradores com apenas uma lanterna devem caminhar em um túnel de uma mina. No máximo dois exploradores podem caminhar juntos e qualquer um deve levar a lanterna. Os exploradores, chamados Andrews, Blake,

Johnson e Kelly podem percorrer o túnel em um minuto, dois minutos, quatro minutos e oito minutos, respectivamente. Quando caminham juntos, eles o fazem na velocidade do explorador mais lento. De que maneira os quatro exploradores podem atravessar o túnel em apenas 15 minutos? Após resolver esse problema, explique como você subiu o primeiro degrau.

- 18.** Você dispõe de duas taças de vinho: uma pequena e a outra grande. Encha a pequena com vinho e então despeje na taça grande. Em seguida, encha a taça pequena com água e derrame uma parte da água na taça grande. Misture o conteúdo da taça grande e despeje na pequena até enchê-la. Haverá mais água na taça grande do que vinho na pequena? Após resolver esse problema, explique como você subiu o primeiro degrau.
- 19.** Duas abelhas, chamadas Romeu e Julieta, vivem em diferentes colméias, mas se encontraram e se apaixonaram. Em uma manhã primaveril sem vento, elas simultaneamente deixaram suas colméias para se visitar. Suas rotas se cruzaram em um ponto a 50 metros da colmeia mais próxima, mas elas não se viram e continuaram suas viagens. No destino, gastaram o mesmo tempo para constatar que a outra não estava e retornaram. Desta vez, se encontraram em um ponto a 20 metros da colmeia mais próxima. Fizeram um lanche e voltaram às suas colméias. Que distância separa as duas colméias? Após resolver esse problema, explique como você subiu o primeiro degrau.
- 20.** Projete um algoritmo que, dadas duas cadeias de caracteres, verifique se a primeira pode ser localizada, como subcadeia, em algum ponto da segunda.
- 21.** O algoritmo seguinte foi projetado para imprimir os termos iniciais da conhecida sequência de Fibonacci. Identifique o corpo da iteração. Localize os passos de inicialização, modificação e teste do controle da repetição. Qual a lista de números gerada por este algoritmo?

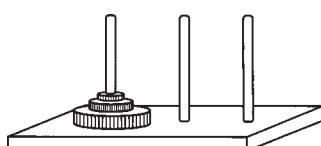
Último $\leftarrow 0$;
Corrente $\leftarrow 1$;
enquanto (Corrente < 100) **faz**
(imprimir o valor de Corrente;

- Temp ← Último;
 Último ← Corrente; e
 Corrente ← Último + Temp)
- 22.** Que sucessão de números será impressa pelo algoritmo seguinte, se ele iniciar com os valores de entrada 0 e 1?
procedimento EscritaMisteriosa (Último, Corrente)
se (Corrente < 100) **então**
 (imprimir o valor de Corrente;
 Temp ← Corrente + Último;
 aplicar EscritaMisteriosa aos valores Corrente e Temp)
- 23.** Modifique o procedimento EscritaMisteriosa do problema anterior de forma que os valores sejam impressos em ordem invertida.
- 24.** Que letras são consultadas pelo algoritmo de procura binária (Figura 4.14) quando aplicado à lista A, B, C, D, E, F, G, H, I, J, K, L, M, N e O para a busca da letra J? E quando se busca a letra Z?
- 25.** Em média, quantas comparações de nomes são feitas ao ser pesquisada uma lista de 6000 entradas, usando-se a busca seqüencial? E se for usada a busca binária?
- 26.** Identifique o corpo da estrutura iterativa seguinte e conte quantas vezes ele será executado. Se o teste da iteração fosse mudado para enquanto (Contador não 6), qual seria a nova resposta?
 Contador ← 1;
enquanto (Contador não 7) **faz**
 (imprimir o valor de Contador e
 Contador ← Contador + 3)
- 27.** Que problemas serão esperados se o programa seguinte for implementado em um computador? (Sugestão: recordar os problemas causados pelos erros de arredondamento associados à aritmética de vírgula flutuante.)
 Contador ← um décimo;
repete
 (imprimir o valor de Contador e
 Contador ← Contador + um décimo)
até (Contador = 1)
- 28.** Projete uma versão recursiva do algoritmo de Euclides (Questão 3 da Seção 4.2).

- 29.** Aplicando os procedimentos Teste1 e Teste2 abaixo, para o mesmo valor de entrada igual a 1, em que as saídas das duas rotinas diferirão?
procedimento Teste1 (Contador)
se (Contador não 5)
então (imprimir o valor de Contador e
 aplicar Teste1 ao valor Contador + 1)
procedimento Teste2 (Contador)
se (Contador não 5)
então (aplicar Teste2 ao valor Contador + 1
 e imprimir o valor de Contador)
- 30.** Identifique os elementos importantes do mecanismo de controle nas rotinas do problema anterior. Qual a condição específica que causa o término do processo? Em que ponto o estado do processo é modificado em direção à condição terminal? Em que ponto o estado do processo de controle é iniciado?
- 31.** Escreva um algoritmo para gerar uma seqüência de inteiros positivos (em ordem crescente) cujos únicos divisores primos sejam 2 e 3. Em outras palavras, o seu programa deverá produzir a seqüência 2, 3, 4, 6, 9, 12, 16, 18, 24, 27, ... Esse programa representa um algoritmo no sentido estrito?
- 32.** Para a lista constituída pelos nomes Alice, Byron, Carol, Duane, Elaine, Floyd, Gene, Henry e Iris, responda às seguintes perguntas:
- Qual algoritmo de busca (seqüencial ou binária) encontrará o nome Gene mais depressa?
 - Qual algoritmo de busca (seqüencial ou binária) encontrará o nome Alice mais rapidamente?
 - Qual algoritmo de busca (seqüencial ou binária) é mais rápido para descobrir a ausência do nome Bruce na lista?
 - Qual algoritmo de busca (seqüencial ou binária) descobrirá em menos tempo a ausência do nome Sue na lista?
 - Quantos elementos deverão ser comparados para localizar o nome Elaine, no caso da busca seqüencial? E na busca binária?
- 33.** O fatorial de 0 é definido como 1. O fatorial de um inteiro positivo é definido como o produto desse inteiro multiplicado pelo fatorial do

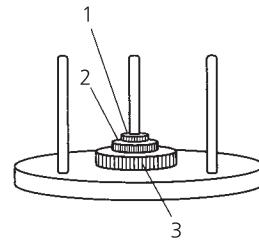
próximo inteiro menor que este e não-negativo. Usamos a notação $n!$ para expressar o fatorial do inteiro n . Assim, o fatorial de 3 (escrito $3!$) pode ser calculado da seguinte maneira: $3 \times (2!) = 3 \times (2 \times (1!)) = 3 \times (2 \times (1 \times (0!))) = 3 \times (2 \times (1 \times (1))) = 6$. Escreva um algoritmo recursivo que calcule o fatorial de um valor fornecido.

- 34.** a. Suponha que você deva ordenar uma lista com cinco nomes e que tenha escrito um algoritmo que ordena listas com quatro nomes. Escreva um algoritmo para ordenar a lista de cinco nomes e que utilize o algoritmo disponível.
b. Escreva um algoritmo recursivo para ordenar listas arbitrárias de nomes, baseado na técnica aplicada na resolução do item (a).
- 35.** O quebra-cabeças chamado Torres de Hanoi consiste em três pinos, e cada um contém vários anéis, empilhados em ordem descendente de diâmetro, de baixo para cima. O problema consiste em mover a pilha de anéis para algum outro pino. É permitido mover somente um anel de cada vez, e em nenhum momento um anel pode ser colocado sobre um outro que lhe seja menor. Observe que, se o quebra-cabeças tivesse somente um anel, a solução seria trivial. Quando se trata de mover vários anéis, se pudermos mover todos, exceto o maior, para algum outro pino, o maior poderá ser colocado no terceiro pino e os restantes ser posteriormente transferidos para cima deste. Usando esta observação, desenvolva um algoritmo recursivo que resolva o quebra-cabeças das Torres de Hanoi para um número arbitrário de anéis.
- 36.** Uma alternativa de solução para o quebra-cabeças das Torres de Hanoi (Problema 35) consiste em imaginar os pinos dispostos circularmente e localizados conforme as posições 4h, 8h e 12 horas de um relógio. Os anéis, que inicialmente estão todos em um dos pinos, são numerados como 1, 2, 3 e assim por diante,



atribuindo-se o valor 1 ao menor anel. Os anéis ímpares, quando estão no topo de uma pilha, podem mover-se para o próximo pino em sentido horário; de modo similar, os anéis pares podem mover-se em sentido anti-horário (contanto que estes movimentos não coloquem um anel sobre um outro menor). Sob estas condições, mover sempre o anel de maior valor e que puder ser deslocado. Com base nesta observação, desenvolva um algoritmo não-recursivo para resolver o problema das Torres de Hanoi.

- 37.** Desenvolva dois algoritmos, um baseado em estruturas iterativas e o outro, em estruturas recursivas, para imprimir o salário diário de um trabalhador que, a cada dia, recebe o dobro do salário do dia anterior (iniciando com um centavo no primeiro dia de trabalho), durante 30 dias. Quais problemas, relativos ao armazenamento de dados,



você provavelmente encontrará se suas soluções forem implementadas em um computador real?

- 38.** Escreva um algoritmo para calcular a raiz quadrada de um número positivo. Inicie com o próprio número como primeira tentativa. Produza repetidamente uma nova tentativa a partir da primeira, calculando a média entre a tentativa anterior e o resultado da divisão do número original por este valor. Analise o controle deste processo iterativo. Qual condição específica deve ser usada para terminar a iteração?
- 39.** Desenvolva um algoritmo que liste todos os possíveis arranjos dos símbolos em uma cadeia de cinco caracteres distintos.
- 40.** Desenvolva um algoritmo que, dada uma lista de nomes, encontre o nome mais longo da lista. Determine a ação que seu algoritmo executará se houver vários nomes com este mesmo comprimento. O que seu algoritmo fará se todos

os nomes da lista tiverem o mesmo comprimento?

41. Desenvolva um algoritmo que, dada uma lista de cinco ou mais números, encontre os cinco menores e os cinco maiores, sem ordenar a lista inteira.
42. Arranje os nomes Brenda, Doris, Raymond, Steve, Timothy e William em uma ordem que exija o menor número de comparações quando a lista for ordenada pelo algoritmo de ordenação por inserção (Figura 4.11).
43. Qual é o número máximo de elementos comparados quando o algoritmo de pesquisa binária (Figura 4.14) é aplicado a uma lista de 4.000 nomes? Compare com a pesquisa seqüencial (Figura 4.6).
44. Use a notação teta para classificar os algoritmos tradicionais de soma e multiplicação, isto é, quando se somam dois números, cada um com n dígitos, quantas adições individuais devem ser feitas? Quando se multiplicam dois números, cada um com n dígitos, quantas multiplicações individuais devem ser feitas?
45. Algumas vezes, uma ligeira modificação em um problema pode alterar significativamente a forma de sua solução. Por exemplo, encontre um algoritmo simples para resolver o seguinte problema e classifique-o usando a notação teta:

Divida um grupo de pessoas em dois subgrupos disjuntos (de tamanhos arbitrários) de tal forma que a diferença entre o total das idades dos membros dos dois grupos seja a maior possível.

Agora, mude o problema de modo que a diferença desejada seja a menor possível e classifique a sua abordagem ao problema.
46. Extraia uma coleção de números da seguinte lista, cuja soma seja 3165. Até que ponto a sua abordagem ao problema é eficiente?
26, 39, 104, 195, 403, 504, 793, 995, 1156, 1673
47. A iteração implementada pela rotina abaixo termina? Justifique sua resposta. Explique o que ocorrerá se esta rotina for realmente executada por um computador (consulte a Seção 1.7).

 $X \leftarrow 1;$
 $Y \leftarrow 1/2;$

enquanto (X não igual a 0) **faz**

$(X \leftarrow X - Y;$
 $Y \leftarrow Y \div 2)$

48. O seguinte trecho de programa foi desenvolvido para calcular o produto de dois inteiros não-negativos X e Y acumulando a soma de X cópias de Y ; por exemplo, 3 multiplicado por 4 é calculado acumulando a soma de três 4s. O seguinte trecho de programa está correto? Justifique sua resposta.

Produto $\leftarrow 0$;
Contador $\leftarrow 0$;
repete (Produto \leftarrow Produto + Y ,
Contador \leftarrow Contador + 1)
até (Contador = X)
49. O seguinte trecho de programa foi desenvolvido para informar qual o maior entre dois inteiros positivos X e Y . O trecho está correto? Justifique sua resposta.

Diferença $\leftarrow X - Y$;
se (Diferença for positiva)
 então (imprimir “ X é maior do que Y ”)
 senão (imprimir “ Y é maior do que X ”)
50. O seguinte trecho de programa foi desenvolvido para localizar o maior elemento de uma lista, não vazia, de inteiros. Ele está correto? Justifique sua resposta.

ValorEmTeste \leftarrow primeiro elemento da lista;
ElementoCorrente \leftarrow primeiro elemento da lista;
enquanto (ElementoCorrente não é o último elemento da lista) **faz**
 se (ElementoCorrente > ValorEmTeste)
 então (ValorEmTeste \leftarrow ElementoCorrente)
 ElementoCorrente \leftarrow próximo elemento da lista)
51. a. Identifique as precondições para a rotina de busca seqüencial apresentada na Figura 4.6. Introduza uma invariante de laço para a estrutura enquanto desse programa, tal que, combinada com a condição terminal, implique que o algoritmo indicará corretamente o sucesso ou fracasso da busca efetuada ao término da iteração.
b. Apresente um argumento que prove que a iteração enquanto da Figura 4.6 realmente termina.
52. Baseado nas precondições de que X e Y contêm inteiros não-negativos, identifique uma

invariante de laço para a seguinte estrutura enquanto, a qual, combinada com a condição terminal, implique que seja $X = Y$ o valor associado a Z , ao término da iteração.

```
Z ← X;  
J ← 0;  
enquanto ( $J < Y$ ) faça  
  ( $Z ← Z - 1$ ;  
    $J ← J + 1$ )
```

Questões sociais

As seguintes questões procuram auxiliar o leitor no entendimento de alguns assuntos éticos, sociais e legais no campo da computação. O objetivo não é meramente o de fornecer respostas a tais questões. O leitor também deve justificá-las e verificar se as justificativas apresentadas preservam sua consistência de uma questão para outra.

1. Por ser atualmente impossível verificar completamente a precisão de programas complexos, sob quais circunstâncias, se existirem, o criador de tais programas pode ser responsabilizado por erros neles existentes?
2. Suponha que você tenha uma idéia e a transforme em um produto a ser consumido por muitas pessoas. Ele exigiu um ano de seu trabalho e um investimento de U\$ 50.000 para resultar em um produto final útil ao público em geral. Em sua forma final, porém, o produto, já de conhecimento público, pode ser usado pela maioria das pessoas sem que elas precisem adquiri-lo de você. Que direitos de remuneração você tem? É ético copiar, sem autorização, *software* de computador?
3. Suponha que um pacote de *software* seja tão caro que esteja totalmente fora de suas posses. É ético copiá-lo para o seu próprio uso? (Afinal de contas, você não estará tirando do produtor uma venda, visto que não teria mesmo como comprar o pacote.)
4. A propriedade de “coisas” sempre foi um tópico discutido. A propriedade de rios, florestas, oceanos etc. tem sido bem debatida. Em que sentido alguém, ou alguma instituição, pode ser proprietária de um algoritmo?
5. Suponha que um programador descubra um novo e surpreendente algoritmo, capaz de quebrar a segurança em um sistema operacional multiusuário. Ele terá direitos autorais sobre este algoritmo? Neste caso, quais direitos? Os seus direitos variam conforme o assunto do algoritmo? É ético anunciar e fazer circular técnicas para quebrar sistemas de segurança? Importa saber o que está sendo quebrado?
6. Algumas pessoas acham que os novos algoritmos são descobertos, enquanto outras acham que são criados. Em que filosofia você se insere? Pontos de vista diferentes levam a conclusões diferentes quanto à propriedade dos algoritmos e dos direitos associados?
7. É ético projetar um algoritmo para realizar um ato ilegal? Importa se o algoritmo nunca for usado?
8. Um autor é remunerado quando seu romance é transposto para o cinema, embora o enredo freqüentemente seja alterado no filme. Quanto do enredo deve ser alterado para que ele seja considerado uma nova história? Que alterações devem ser feitas em um algoritmo para que ele se torne um novo algoritmo?

9. Os softwares educacionais estão sendo comercializados para crianças de 18 meses de idade ou menos. Os proponentes argumentam que esses softwares fornecem imagens e sons que de outra forma não estariam disponíveis para muitas crianças. Os oponentes argumentam que tais programas são substitutos indevidos à interação pessoal dos pais com as crianças. Qual é a sua opinião? Você deve formar uma opinião sem conhecer melhor o software?

Leituras adicionais

- Brassard, G., and P. Bratley. *Fundamentals of Algorithmics*. Englewood Cliffs, NJ: Prentice-Hall, 1996.
- Cormen, T. H., C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms* 2nd ed. New York: McGraw-Hill, 2002.
- Gries, D. *The Science of Programming*. New York: Springer-Verlag, 1998.
- Harbin, R. *Origami — the Art of Paper Folding*. London: Hodder Paperbacks, 1973.
- Knuth, D. E. *The Art of Computer Programming*, vol. 3, 3rd ed. Reading, MA: Addison Wesley Longman, 1998.
- Kruse, R. L. and A. J. Ryba. *Data Structures and Program Design in C++*. Upper Saddle River, NJ: Prentice Hall, 1999.
- Polya, G. *How to Solve It*. Princeton, NJ: Princeton University Press, 1973.
- Roberts, E. S. *Thinking Recursively*. New York: Wiley, 1986.
- Sedgewick, R., and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Reading, MA: Addison-Wesley, 1996.

CAPÍTULO 5

Linguagens de programação

Seria quase impossível o desenvolvimento de sistemas complexos de *software*, como os sistemas operacionais, os *softwares* de redes e a grande quantidade de aplicações disponíveis atualmente, se os seres humanos fossem obrigados a expressar os seus algoritmos diretamente em linguagem de máquina. Trabalhar com um volume significativo de detalhes da linguagem de máquina e, ao mesmo tempo, tentar organizar sistemas complexos é, no mínimo, uma experiência desgastante. Como consequência, linguagens de programação semelhantes ao nosso pseudocódigo foram desenvolvidas, permitindo expressar algoritmos em um formato que fosse tanto agradável como fácil de traduzir para instruções em linguagem de máquina. Tais linguagens evitam a complexidade dos registradores, endereços de memória e ciclos de máquina durante o processo de desenvolvimento de programas, concentrando-se, em lugar disso, nas características do problema a ser solucionado. Neste capítulo, estudaremos o tópico de linguagens de programação.

- 5.1 Perspectiva histórica**
 - As primeiras gerações de linguagens
 - Independência de máquina
 - Paradigmas de programação
- 5.2 Conceitos tradicionais de programação**
 - Variáveis e tipos de dados
 - Estruturas de dados
 - Constantes e literais
 - Instruções de atribuição
 - Instruções de controle
 - Comentários
- 5.3 Módulos de programas**
 - Procedimentos
 - Parâmetros
 - Funções
 - Instruções de entrada e saída
- 5.4 Implementação de linguagens**
 - O processo de tradução
 - Ligaçāo e carregamento
 - Pacotes para o desenvolvimento de *software*
- * **5.5 Programação orientada a objeto**
 - Classes e objetos
 - Construtores
 - Recursos adicionais
- * **5.6 Programação de atividades concorrentes**
- * **5.7 Programação declarativa**
 - Dedução lógica
 - Prolog

*Os asteriscos indicam sugestões de seções consideradas opcionais.

5.1 Perspectiva histórica

Iniciemos o nosso estudo apresentando o desenvolvimento histórico das linguagens de programação.

As primeiras gerações de linguagens

Originalmente, o processo de programação era realizado de uma forma laboriosa, em que o programador escrevia todos os algoritmos em linguagem de máquina. Este método teve grande importância para a já então rigorosa atividade do projeto de algoritmos, a qual, porém, na maioria dos casos, conduzia ao aparecimento de erros que precisavam ser localizados e corrigidos (processo conhecido como *depuração*) antes da liberação final do programa.

O primeiro passo para simplificar o processo de programação foi eliminar o uso de dígitos numéricos para representar os códigos de operação e os operandos, presentes nas linguagens de máquina. Para esta finalidade, foi comum o emprego de mnemônicos no lugar de notações hexadecimais para representar os diversos códigos de operações durante a etapa de projeto. Em vez de empregar o código de operação correspondente à instrução para carregar um registrador, o programador escreveria, por exemplo, LD, ou, para simbolizar o armazenamento do conteúdo de um registrador na memória, usaria ST. No caso dos operandos, foram estabelecidas regras segundo as quais o programador atribuía nomes descritivos (em geral chamados *identificadores*) às posições de memória e os utilizava nas instruções em lugar dos endereços numéricos de posições de memória. Um caso específico deste uso foi a utilização de nomes como R0, R1, R2,... para os registradores do processador.

Escolhendo nomes descritivos para as posições de memória e usando mnemônicos para representar códigos de operação, os programadores aumentaram significativamente a legibilidade de sequências de instruções de máquina. Como exemplo, retomemos a rotina em linguagem de máquina apresentada na Figura 2.7 da Seção 2.2, a qual efetua a adição dos conteúdos das posições de memória 6C e 6D e guarda o resultado na posição 6E. As instruções, em notação hexadecimal, apresentam-se como:

```
156C
166D
5056
306E
C000
```

Se associarmos o nome *Valor* à posição de memória 6C, *Imposto* à posição 6D e *Total* à posição 6E, usando a técnica mnemônica, a mesma rotina poderá ser expressa da seguinte maneira:

```
LD R5, Valor
LD R6, Imposto
ADDI R0, R5 R6
ST R0, Total
HLT
```

A maioria concorda em que a segunda forma, embora ainda incompleta, é melhor que a primeira para representar a rotina. (Note-se que o mnemônico ADDI é usado para representar o código de operação de adição de inteiros, para distingui-lo do código de operação de adição de números reais, representado por ADDF.)

Quando estas técnicas foram introduzidas, os programadores escreviam em papel os programas nesta notação e depois os traduziam para uma forma que fosse utilizável em máquinas. Não demorou, porém, até que este processo de tradução fosse ele próprio identificado como um procedimento que poderia ser executado pela máquina. Como consequência, o uso de mnemônicos foi formalizado como uma linguagem de programação, chamada **linguagem de montagem**^{*}, e um programa chamado

^{*}N. de T. Em inglês, *assembly language*.

montador^{**} foi desenvolvido para traduzir os programas escritos em linguagem de montagem para um formato que fosse compatível com as máquinas. O programa recebeu o nome de *montador* porque sua tarefa era a de *montar* instruções de máquina a partir de códigos de operação e operandos, obtidos da conversão de mnemônicos e identificadores. O termo de **linguagem de montagem** tem origem neste fato.

O surgimento das linguagens de montagem constituiu um gigantesco passo na pesquisa de melhores ambientes de programação. De fato, muitos as consideraram como representantes de uma geração totalmente nova de linguagens de programação. A propósito, as linguagens de montagem passaram a ser conhecidas como linguagens de segunda geração, em contrapartida às de primeira geração, representadas pelas próprias linguagens de máquina.

Embora as linguagens de segunda geração apresentassem muitas vantagens sobre as linguagens de máquina a que correspondiam, ainda estavam longe de oferecer um ambiente de programação adequado. Afinal de contas, as primitivas utilizadas nas linguagens de montagem eram essencialmente as mesmas encontradas nas linguagens de máquina correspondentes. A diferença estava simplesmente na sintaxe usada para representá-las. Uma consequência desta forte associação entre linguagens de montagem e linguagens de máquina é que qualquer programa escrito em uma linguagem de montagem depende inherentemente de máquina. Isso quer dizer que as instruções de um programa são expressas em termos dos atributos de uma máquina em particular. Por sua vez, um programa escrito em linguagem de montagem não pode ser transportado com facilidade de uma máquina para outra, pois teria de ser reescrito de forma compatível com a configuração de registradores e com o conjunto de instruções da nova máquina.

Outra desvantagem de uma linguagem de montagem é que, embora um programador não seja forçado a programar as instruções na forma de padrões de bits, seu raciocínio continua voltado às minúcias do programa, aos pequenos passos incrementais da linguagem de máquina, em vez de concentrar-se na lógica global da solução para a sua tarefa. A situação é análoga à do projeto de uma casa em termos de tábuas, vidros, pregos, tijolos e assim por diante. É verdade que, em última análise, a construção da casa exige uma descrição baseada em tais componentes elementares, mas o processo de projeto ficará mais fácil se pensarmos em termos de cômodos, alicerces, janelas, telhados e assim por diante.

Em resumo, os componentes elementares nos quais um produto deve, em última instância, ser expresso não são necessariamente os elementos mais adequados para serem utilizados durante o seu projeto. O processo de projeto transcurre de forma mais apropriada com o uso de primitivas de alto nível, em que cada uma represente um conceito associado a alguma característica importante do produto. Uma vez terminado o projeto, tais primitivas podem ser traduzidas em termos de conceitos de nível mais baixo, de forma que os detalhes de implementação se evidenciem, do mesmo modo como uma construtora extrai, do projeto de um edifício, a lista de material a ser comprado.

Seguindo esta filosofia, os pesquisadores de computação começaram a desenvolver linguagens de programação que fossem mais direcionadas ao desenvolvimento de software do que as de montagem, de baixo nível. O resultado foi o aparecimento de uma terceira geração de linguagens de programação, que se distinguiram das anteriores pelo fato de suas primitivas serem de nível mais alto e independentes de

Software multiplataforma

Um programa de aplicação típico depende do sistema operacional para realizar muitas de suas tarefas. Ele pode necessitar dos serviços do gerente de janelas para sua comunicação com o usuário, ou usar o gerente de arquivos para obter dados do armazenamento em massa. Infelizmente, os sistemas operacionais exigem que as solicitações para esses serviços sejam feitas de maneiras diferentes. Assim, se os programas devem ser transferidos e executados em rede e *internets*, envolvendo diferentes máquinas e sistemas operacionais, eles devem ser independentes tanto da máquina como do sistema operacional. O termo *Multiplataforma* (*cross-platform*) é usado para refletir esse nível adicional de independência, isto é, um *software* multiplataforma é independente do projeto do sistema operacional, bem como do projeto do *hardware* da máquina, sendo portanto executável em redes com diversas máquinas.

**N. de T. Em inglês, *assembler*.

máquina. Os exemplos mais conhecidos são o FORTRAN (FORmula TRANslator), que foi desenvolvido para aplicações científicas e de engenharia, e o COBOL (COmmon Business-Oriented Language), desenvolvido pela marinha norte-americana para aplicações comerciais.

Em geral, o objetivo das linguagens de programação de terceira geração foi identificar um conjunto de primitivas de alto nível (essencialmente, com o mesmo espírito com que desenvolvemos nosso pseudocódigo do Capítulo 4) com as quais o *software* seria desenvolvido. Cada primitiva foi projetada para ser implementada por meio de uma seqüência de primitivas de baixo nível, disponíveis nas linguagens de máquina. Por exemplo, a instrução

atribua a PesoBruto o valor (PesoVeículo + PesoCarga)

expressa uma atividade de alto nível sem descrever de que forma uma máquina executará tal tarefa. Contudo, pode ser implementada por uma seqüência de instruções de máquina, como a que foi apresentada anteriormente. Assim, nossa estrutura em pseudocódigo

identificador ← expressão

pode ser vista como uma primitiva de alto nível em potencial.

Uma vez identificado este conjunto de primitivas de alto nível, um programa, conhecido como **tradutor**^{*}, foi criado para converter programas, escritos na forma de primitivas de alto nível, em programas de linguagem de máquina. O tradutor se assemelhava aos montadores de segunda geração, exceto pelo fato de geralmente precisar agregar (ou *compilar*) várias instruções de máquina, na forma de pequenas seqüências, para simular a atividade representada por uma única primitiva de alto nível. Por essa razão, estes programas tradutores se tornaram conhecidos pelo nome de **compiladores**.

O desenvolvimento do primeiro compilador é atribuído a Grace Hopper, que também promoveu o conceito de linguagens de terceira geração — tarefa que não foi tão fácil como se imagina. A idéia de escrever programas em uma forma similar à linguagem natural era tão revolucionária que muitos gerentes a combateram. De fato, Hopper demonstrou a versatilidade de um tradutor para uma linguagem de terceira geração, na qual foram usados termos alemães, em vez de ingleses. O ponto era que a linguagem de programação era construída a partir de um pequeno conjunto de primitivas que poderiam ser expressas em diversas linguagens naturais, com pequenas modificações no tradutor. Entretanto, ela se surpreendeu ao constatar que muitos na audiência se chocaram com o fato de ensinar um computador a entender alemão em tempos tão próximos da Segunda Guerra Mundial. Hoje sabemos que entender uma linguagem natural envolve muito mais do que responder a algumas primitivas rigorosamente definidas.

Uma alternativa popular aos tradutores, chamados **interpretadores**, surgiu como outro meio de implementar linguagens de terceira geração. Esses programas são similares aos tradutores, exceto em que executam as instruções à medida que elas vão sendo traduzidas, em vez de gravar a versão traduzida para uso futuro, isto é, em vez de produzir uma cópia em linguagem de máquina de um programa a ser executado mais tarde, um interpretador executa as instruções imediatamente após a sua tradução.

Independência de máquina

Com o desenvolvimento das linguagens de terceira geração, o objetivo da independência de máquina foi amplamente alcançado. Dado que as instruções das linguagens de terceira geração não referenciam os atributos de uma determinada máquina, elas podem ser facilmente compiladas tanto em uma máquina como em outra. Assim, um programa escrito em linguagem de terceira geração teoricamente poderia ser usado em qualquer máquina, bastando usar o compilador correspondente.

A realidade, no entanto, demonstrou que não é tão simples. Quando um compilador é projetado, certas restrições impostas pela máquina subjacente são, em última instância, refletidas como características da linguagem a ser traduzida. Por exemplo, os diferentes modos como as máquinas tratam as ope-

*N. de T. Em inglês, *translator*.

rações de E/S têm causado o aparecimento na “mesma” linguagem de diferentes características, ou dialetos. Conseqüentemente, em geral é necessário fazer ao menos pequenas modificações no programa antes de movê-lo de uma máquina para outra.

A causa deste problema de portabilidade é, em alguns casos, a falta de concordância em relação à composição correta da definição de uma linguagem em particular. Para auxiliar nesta questão, o American National Standards Institute (ANSI) e a International Organization for Standardization (ISO) adotaram e publicaram padrões para muitas das linguagens mais populares. Em outros casos, surgiram padrões informais, devido à popularidade de um dado dialeto de uma linguagem e ao desejo, por parte de alguns autores de compiladores, de oferecerem produtos compatíveis. Contudo, mesmo no caso de linguagens padronizadas, os projetistas de compiladores geralmente oferecem características, às vezes chamadas *extensões da linguagem*, que não são parte da linguagem padronizada. Se um programador tirar vantagem dessas características, o programa produzido não será compatível com ambientes que usam compiladores de outros vendedores.

Na história global das linguagens de programação, é realmente pouco significativo o fato de as linguagens de terceira geração não terem conseguido atingir uma verdadeira independência de máquina. Primeiramente, elas se mostraram tão próximas da independência de máquina que os programas nelas desenvolvidos podiam ser transportados de uma máquina para outra com relativa facilidade. Em segundo lugar, o objetivo de independência de máquina acabou convertendo-se apenas em um embrião para metas muito mais exigentes. Na época em que a independência de máquina finalmente se tornou acessível, a sua importância se diluiu diante das exigências da época, então muito mais fortes. De fato, a constatação de que máquinas podiam executar instruções de alto nível como

atribua a Total o valor (Preço + Imposto)

conduziu os pesquisadores a sonhar com ambientes de programação que permitissem aos seres humanos comunicar-se com as máquinas usando diretamente os conceitos abstratos com os quais costumam analisar os problemas, evitando assim a tradução dos conceitos para um formato compatível com a máquina. Além disso, os pesquisadores buscam máquinas voltadas mais ao desenvolvimento de algoritmos do que à sua execução. O resultado tem sido uma evolução constante nas linguagens de programação que impede uma classificação clara em termos de gerações.

Paradigmas de programação

A abordagem de gerações para classificar as linguagens de programação é baseada em uma escala linear (Figura 5.1), de acordo com o grau em que o usuário é libertado do mundo da ininteligibilidade computacional e autorizado a analisar elementos associados ao seu próprio problema. Na realidade, o desenvolvimento das linguagens de programação não evoluiu exatamente desta forma, mas se ramificou conforme foram aparecendo diferentes formas de processos de programação (diferentes paradigmas). Por conseguinte, o desenvolvimento histórico das linguagens de programação fica mais bem representado por um diagrama com

múltiplas trajetórias, conforme mostra a Figura 5.2, na qual estão apresentados os diversos caminhos, resultantes de diferentes paradigmas, que surgiram e progrediram de forma independente. A Figura 5.2 apresenta quatro caminhos que representam os paradigmas funcional, orientado a objeto, imperativo e

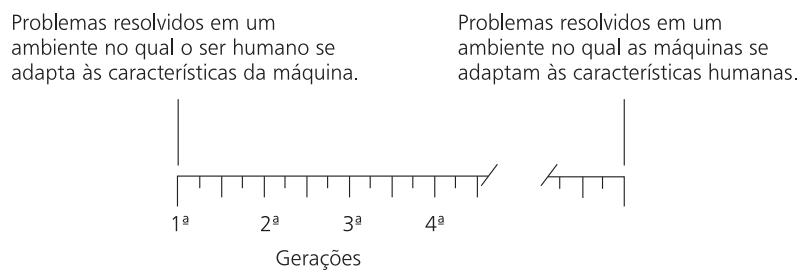


Figura 5.1 Gerações de linguagens de programação.

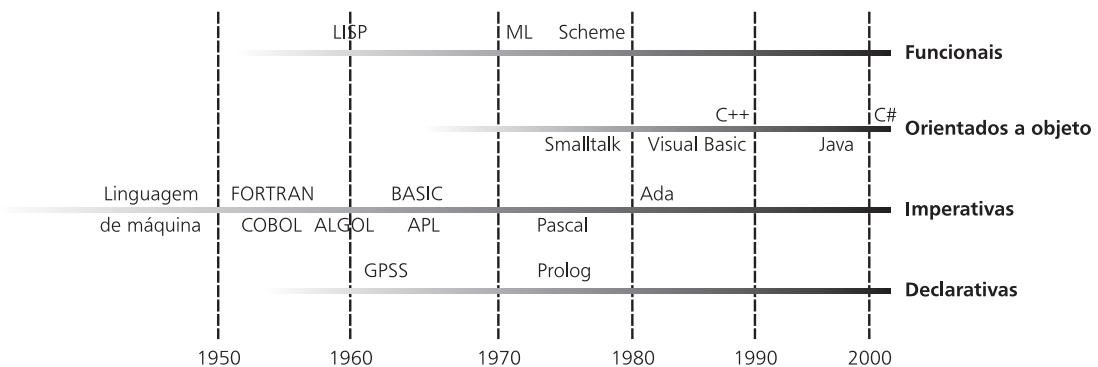


Figura 5.2 A evolução dos paradigmas de programação.

declarativo, com várias linguagens associadas a cada um, assinalando o aparecimento de cada uma em relação às demais. Isto não implica, necessariamente, que uma dada linguagem tenha sido criada a partir da precedente.

Devemos notar que embora os paradigmas identificados na Figura 5.2 sejam chamados *paradigmas de programação*, essas alternativas possuem ramificações além do processo de programação. Eles representam abordagens fundamentalmente diferentes para a construção de soluções para os problemas, portanto, afetam todo o processo de desenvolvimento de software. Nesse sentido, o nome *paradigmas de programação* é equivocado. Mais realista seria *paradigmas de desenvolvimento de software*.

O **paradigma imperativo**, também conhecido como **paradigma procedimental**^{*}, representa o enfoque tradicional para o processo de programação. De fato, é no paradigma imperativo que se baseia o nosso pseudocódigo do Capítulo 4, bem como a linguagem de máquina discutida no Capítulo 2. Como o nome sugere, o paradigma imperativo define o processo de programação como o desenvolvimento de uma seqüência de instruções que, quando executada, manipula dados para obter o resultado desejado. Assim, o paradigma imperativo diz-nos que a abordagem ao processo de programação é encontrar um algoritmo para resolver o problema e então expressá-lo como uma seqüência de instruções.

Em contraste com o paradigma imperativo, que requer do programador um algoritmo para resolver o problema, está o **paradigma declarativo**, que requer a descrição do problema. A estratégia aqui consiste em descobrir e implementar um algoritmo geral para solucionar problemas. Feito isso, será possível resolver problemas simplesmente apresentando-os de uma forma compatível com nosso algoritmo, para então aplicar tal algoritmo. Neste contexto, a tarefa do programador se resume a desenvolver uma descrição precisa do problema, em vez de descobrir um algoritmo para resolvê-lo.

O principal obstáculo ao desenvolvimento de um ambiente de programação baseado no paradigma declarativo é descobrir um algoritmo subjacente de resolução de problemas. Por isso, as primeiras linguagens declarativas eram, por natureza, de propósito específico, projetadas para serem utilizadas em aplicações particulares. Por exemplo, o método declarativo foi por muitos anos utilizado para simular sistemas (econômicos, físicos, políticos e assim por diante) para a execução de testes de hipóteses. Sob esta ótica, o algoritmo subjacente consistirá essencialmente no processo de simular o transcorrer do tempo por meio de cálculos repetitivos dos valores dos parâmetros (produto interno bruto, déficit da balança comercial e assim por diante), feitos com base nos cálculos anteriores. Implementar uma linguagem declarativa para tais simulações requer, então, a implementação de um algoritmo que execute tal procedimento repetitivo. Assim procedendo, a única tarefa exigida de um programador que faz uso desta

*N. de T. Em inglês, *procedural*.

linguagem será a de descrever as relações existentes entre os parâmetros a serem simulados. Então, o algoritmo de simulação apenas simula a passagem do tempo, usando as relações fornecidas para a execução dos seus cálculos.

Mais recentemente, o paradigma declarativo ganhou um grande impulso, ao descobrir-se que a lógica formal, disciplina estudada como parte da matemática, propicia a elaboração de um algoritmo simples para a resolução de problemas, adequado para uso em sistemas de programação declarativa de propósito geral. Isso acarretou um aumento de interesse no paradigma declarativo, bem como o surgimento da programação lógica, que será discutida na Seção 5.7.

O **paradigma funcional** visualiza o processo de desenvolvimento de programas como a conexão de “caixas pretas” predefinidas, em que cada uma recebe entradas e produz saídas de forma a se produzir o relacionamento desejado de entrada-saída. Os matemáticos se referem a tais “caixas” como funções, razão pela qual este paradigma recebeu o nome de funcional. As primitivas de uma linguagem de programação funcional consistem em funções elementares com as quais o programador deverá construir funções mais elaboradas, necessárias para resolver o problema. Assim, um programador que utiliza o paradigma funcional aborda o problema de desenvolvimento de *software* encontrando um modo de conectar as funções elementares, primitivas, para produzir um sistema que compute os resultados desejados. Em síntese, o processo de programação sob o paradigma funcional é construir funções com aninhamentos complexos de funções simples.

Para ilustrar, a Figura 5.3 mostra como pode ser construída uma função para calcular a média de uma seqüência de números, a partir de três funções mais simples: uma delas é *Soma*, que calcula a soma dos elementos da lista; outra, *Conta*, que conta o número de elementos da lista, e a terceira, *Divide*, que determina o quociente dos valores anteriores. Esta construção, na sintaxe da linguagem LISP, uma importante linguagem de programação funcional, pode ser representada pela expressão:

(Divide (Soma Numeros) (Conta Numeros))

cuja estrutura aninhada reflete o fato de que a função *Divide* opera com os resultados das funções *Soma* e *Conta*. Como outro exemplo, suponha que tenhamos uma função chamada *Ordena*, que ordene uma lista de números, e outra chamada *Primeiro*, que retorne o primeiro elemento de uma lista. Então, a expressão

(Primeiro (Ordena Lista))

fornecerá o menor elemento em *Lista*. Aqui, a estrutura aninhada indica que a saída da função *Ordena* é a entrada de *Primeiro*, isto é, a lista é ordenada e então o primeiro elemento é extraído.

Para entender a principal vantagem do paradigma funcional em relação ao imperativo, considere a analogia com um sistema de fábricas necessário para produzir bens de consumo. Esse sistema pode envolver uma fábrica que recebe carregamentos de minério de ferro, calcário e carvão e produz aço; uma fábrica que recebe carregamentos de aço e produz peças de automóvel; e outra fábrica que recebe carregamentos de componentes de automóvel e produz carros completos. No paradigma funcional, o sistema inteiro de fábricas seria visto como uma grande fábrica (embora distribuída geograficamente) que recebe carregamentos de matéria-prima e produz automóveis. As fábricas individuais seriam vistas meramente como componentes do sistema maior, que são co-

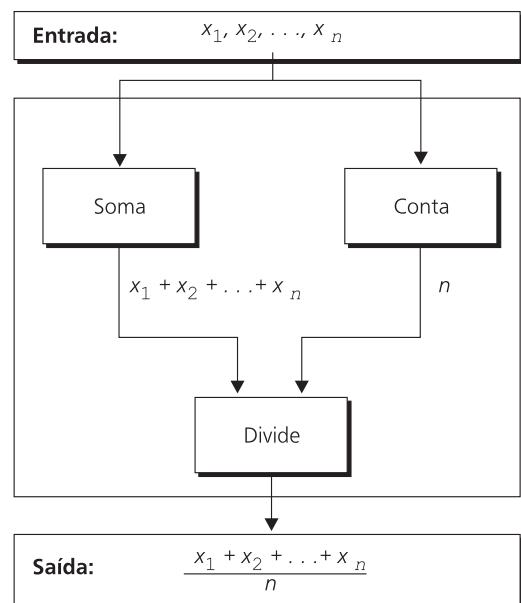


Figura 5.3 Uma função que calcula a média de uma seqüência de números, construída a partir de funções mais simples: *Soma*, *Conta* e *Divide*.

nectados de tal forma que as saídas de alguns são as entradas de outros, permitindo que os materiais fluam através do sistema de uma maneira predeterminada.

O paradigma imperativo, por sua vez, veria a produção de bens de consumo como um sistema de fábricas individuais, em que cada uma realiza suas próprias atividades e fabrica um produto que é guardado para consumo posterior. Sob este paradigma, o sistema pode ser expresso como:

```
EstoqueA ← resultado da aplicação do procedimento FabricaAço a MinérioFerro,  
          Calcário e Carvão;  
EstoqueB ← resultado da aplicação do procedimento FabricaAutomovel ao conteúdo de  
          EstoqueA
```

.

.

.

Vemos então que o paradigma funcional resulta em uma estrutura que poderemos esperar se todas as fábricas pertencerem à mesma corporação. Nesse caso, a produção de cada fábrica é coordenada com as necessidades das outras, de maneira que os materiais fluem continuamente através do sistema. Entretanto, o paradigma imperativo produz um sistema de fábricas no qual cada uma pertence a uma corporação independente e estoca os seus produtos para entrega posterior aos clientes. Em resumo, o paradigma funcional leva a um sistema no qual materiais são entregues a seu destino à medida que vão sendo produzidos, enquanto o paradigma imperativo leva a um sistema de estoques. Embora útil, esta distinção tem consequências significativas em termos de eficiência — tanto em sistemas de *software* com na produção de bens de consumo.

Os proponentes do paradigma funcional também assinalam que a construção de *software* complexo a partir de funções primitivas predefinidas leva a sistemas bem organizados cuja estrutura é naturalmente modularizada em termos de funções. Além disso, à medida que novas funções são desenvolvidas, elas podem se tornar primitivas para a construção de outras ainda mais complexas. (As versões originais do LISP forneciam apenas um punhado de primitivas, ao passo que os sistemas LISP atuais fornecem centenas.) Assim, o paradigma funcional provê um ambiente no qual hierarquias de abstração são facilmente implementadas, permitindo a construção de *software* novo a partir de grandes componentes predefinidos, em vez de se partir do zero. A criação de tais ambientes de desenvolvimento de *software* é uma meta primordial na engenharia de *software*, como veremos no Capítulo 6.

O **paradigma orientado a objeto**, que leva ao processo de programação chamado **programação orientada a objeto (POO)** é outra abordagem ao processo de desenvolvimento de *software*. Usando este esquema, unidades de dados são vistas como “objetos” ativos, em vez dos elementos passivos correspondentes, preconizados pelo paradigma imperativo tradicional. Para esclarecer este ponto, consideremos uma lista de nomes. No paradigma imperativo tradicional, esta lista é considerada apenas como um agrupamento de dados. Qualquer programa que tenha acesso a ela deverá incorporar os algoritmos necessários para executar as manipulações desejadas. Assim, a lista é passiva no sentido estrito, ou seja, em vez de assumir a responsabilidade de se autogerenciar, é mantida por um programa de controle. No paradigma orientado a objeto, porém, a lista é considerada um objeto, formado pela lista de dados propriamente dita e por um conjunto de rotinas responsáveis pela sua manipulação. Este arranjo pode incluir rotinas responsáveis pela inserção de novos elementos na lista, pela eliminação de um dado elemento, por descobrir se a lista está vazia e pela tarefa de ordenação da lista. Em contrapartida, um programa que deva fazer acessos a esta lista não precisará conter algoritmos que executem tais tarefas, mas deverá utilizar as rotinas já existentes no objeto. De certo modo, em vez de ordenar a lista, como ocorria no paradigma imperativo, o programa solicita à mesma que se auto-ordene.

Como outro exemplo da abordagem orientada a objeto, consideremos a tarefa de desenvolver uma interface gráfica para usuários. Aqui, os ícones da tela de um monitor são implementados como objetos. Cada objeto compreende um conjunto de rotinas que descrevem a forma como o objeto deve

responder à ocorrência de vários eventos, tais como a de ser selecionado por meio de um aperto do botão do *mouse* ou a de ser deslocado pela tela afora, em resposta aos movimentos do *mouse*. Assim, o sistema inteiro tem a forma de uma coleção de objetos, e cada um sabe como responder a certos eventos.

Muitas das vantagens de um projeto orientado a objeto são consequências da estrutura modular decorrente da filosofia de orientação a objeto. Cada objeto é implementado como uma unidade independente e bem definida. Uma vez determinadas dessa forma as propriedades de uma entidade, a definição assim obtida pode ser reutilizada todas as vezes que esta entidade se fizer necessária. Assim, os adeptos da programação orientada a objeto argumentam que tal paradigma fornece um ambiente natural para o uso de “blocos construtivos” no desenvolvimento de *software*. Eles preconizam bibliotecas de *software* baseado em objetos, com o qual novos sistemas de *software* podem ser elaborados, de forma similar ao que ocorre com muitos produtos tradicionais, construídos a partir de componentes pré-fabricados.

O paradigma da orientação a objeto está se tornando muito popular, e muitos acreditam que exercerá uma influência dominante no futuro. De fato, nos capítulos seguintes, mencionaremos várias vezes o papel desempenhado pelos métodos orientados a objeto em toda a Ciência da Computação. Outra vantagem da estrutura modular obtida a partir do paradigma orientado a objeto é que toda a comunicação entre os módulos é feita de uma maneira uniforme e bem definida (trocando mensagens entre os objetos) — uma técnica que é prontamente adaptável à comunicação entre computadores ligados em rede. Sem dúvida, o conceito de objetos que trocam mensagens entre si é uma maneira natural de abordar a tarefa de desenvolvimento de sistemas de *software* que serão distribuídos em uma rede. Assim, você não deve se surpreender ao constatar que o *software* baseado no modelo cliente-servidor (Seção 3.3) freqüentemente é desenvolvido no contexto orientado a objeto — o servidor é um objeto que responde às mensagens de outros objetos, que fazem o papel de clientes.

O paradigma orientado a objeto está tendo forte influência no campo da computação, portanto, será discutido em maior detalhe na Seção 5.5. Além disso, veremos repetidamente as implicações desse paradigma nos capítulos futuros. Em particular, testemunharemos a influência do paradigma orientado a objeto na área de Engenharia de *Software* (Capítulo 6), no projeto de bancos de dados (Capítulo 9) e, no Capítulo 7, veremos como a abordagem orientada a objeto para a construção de *software* evoluiu como extensão natural do estudo das estruturas de dados.

Finalmente, devemos notar que as rotinas internas aos objetos que descrevem como eles devem responder às várias mensagens são essencialmente pequenas unidades de programas imperativos. Assim, a maioria das linguagens orientadas a objeto contém muitas das características encontradas nas linguagens imperativas. De fato, a popular linguagem orientada a objeto C++ foi desenvolvida com a adição de recursos orientados a objeto à linguagem imperativa conhecida como C. Nas seções 5.2 e 5.3, exploraremos características comuns tanto às linguagens imperativas quanto às orientadas a objeto. Assim fazendo, estaremos discutindo conceitos que permeiam a maioria dos softwares atuais.



QUESTÕES/EXERCÍCIOS

1. Em que sentido um programa em linguagem de terceira geração é independente de máquina? Em que sentido ele ainda depende da máquina?
2. Qual a diferença entre um montador e um compilador?
3. Podemos resumir o paradigma de programação imperativo dizendo que ele coloca ênfase em descrever um processo que conduz à solução de um dado problema. Resuma, de forma semelhante, os paradigmas declarativo, funcional e orientado a objeto.
4. Em que sentido as linguagens de programação de terceira geração são de nível mais alto que as de gerações anteriores?

5.2 Conceitos tradicionais de programação

Nesta seção, consideraremos alguns dos conceitos genéricos encontrados em linguagens de programação imperativas e orientadas a objeto. Com este propósito, nas seções subsequentes, apresentaremos exemplos escritos nas linguagens Ada, C, C++, C#, FORTRAN, Java e Pascal. FORTRAN, Pascal e C são linguagens imperativas de terceira geração. C++ é uma linguagem orientada a objeto que foi desenvolvida como uma extensão da linguagem C. Java e C# são linguagens orientadas a objeto derivadas do C++. (O Java foi desenvolvido pela Sun Microsystems, enquanto o C# é um produto da Microsoft). A Ada foi originalmente projetada como uma linguagem imperativa de terceira geração, e contém muitas características das linguagens orientadas a objeto. Somente a sua versão mais nova, porém, realiza este paradigma de modo mais completo.

No Apêndice D, encontra-se uma rápida introdução a cada uma dessas linguagens, bem como um exemplo de como o algoritmo de ordenação por inserção poderia ser implementado em cada uma. Você

poderá desejar consultar esse apêndice à medida que for lendo esta seção. Lembre-se, contudo, de que nossa intenção é desenvolver um entendimento das características básicas encontradas nas linguagens de programação. Nossa uso de linguagens específicas é meramente para mostrar como as características discutidas são de fato implementadas. Assim, você não deve se perder nos detalhes dessas linguagens.

Culturas das linguagens de programação

Como acontece com as linguagens naturais, os usuários de diferentes linguagens de programação tendem a desenvolver diferentes culturas e freqüentemente debatem os méritos de suas perspectivas. Algumas vezes, essas diferenças são significativas, como no caso em que diferentes paradigmas de programação estão envolvidos. Em outros casos, as diferenças são sutis. Por exemplo, enquanto o texto faz distinção entre procedimentos e funções (Seção 5.3), os programadores C se referem a ambos como funções. Isto porque um procedimento em um programa C é apresentado

como uma função que não retorna valor. Um exemplo similar é que os programadores C++ se referem a um procedimento de um objeto como uma função-membro, enquanto o termo genérico é método. Essa discrepância pode ser explicada, uma vez que o C++ foi desenvolvido como uma extensão de C. Outra diferença cultural é que os programas em Pascal e em Ada normalmente são tipografados com as palavras reservadas em negrito – uma tradição que hoje em dia é largamente praticada pelos usuários do C, C++, FORTRAN e Java.

O texto procura se manter em posição neutra quanto às linguagens, usando terminologia genérica. Contudo, cada exemplo específico é apresentado em uma forma compatível com o estilo da linguagem envolvida. Ao encontrar esses exemplos, lembre-se de que eles são apresentados para mostrar como as idéias genéricas aparecem nas linguagens – não como um meio de ensinar os detalhes de uma linguagem particular. Procure observar a floresta em vez das árvores.

Embora tenhamos incluído linguagens orientadas a objeto, como C++, Java e C#, em nossos exemplos, abordaremos esta seção como se estivéssemos escrevendo um programa no paradigma imperativo, porque muitas unidades nos programas orientados a objeto (tais como o procedimento que descreve como o objeto deve reagir a estímulos externos) são essencialmente pequenos programas imperativos. Mais tarde, na Seção 5.5, focalizaremos características únicas do paradigma orientado a objeto.

As instruções, nas linguagens de programação tomadas como exemplos, pertencem a três categorias: instruções declarativas, instruções imperativas e comentários. As **instruções declarativas** definem uma terminologia personalizada a ser utilizada no programa, como, por exemplo, os nomes associados aos dados. As **instruções imperativas** descrevem os passos dos algoritmos subjacentes e os **comentários** facilitam a leitura de um programa, explicando suas características menos acessíveis em um formato mais inteligível para os interessados. Normalmente, os programas imperativos (ou uma unidade de programa imperativo, como um procedimento) inicia com uma coleção de instruções declarativas que descrevem os dados a serem manipulados pelo programa. Esse material preliminar é seguido pelas instruções imperativas que descrevem o algoritmo a ser executado (Figura 5.4). Os comentários são dispersos ao longo do programa, quando necessários para esclarecê-lo. Vamos então iniciar a nossa apresentação com os conceitos associados às instruções declarativas.

Variáveis e tipos de dados

Como indicado na Seção 5.1, as linguagens de programação de alto nível permitem que localizações na memória principal sejam referenciadas por nomes descritivos, em vez de endereços numéricos. Tal nome é conhecido como **variável**, em reconhecimento ao fato de que, ao mudar o valor armazenado na localização, o valor associado com o nome muda, à medida que o programa vai sendo executado. Nossas linguagens tomadas como exemplos exigem que as variáveis sejam estabelecidas por meio de uma instrução declarativa, antes de serem usadas no programa. Essas instruções declarativas também exigem que o programador descreva o tipo de dados armazenados na localização de memória associada com a variável.

Esse tipo é conhecido como **tipo de dados** e envolve a maneira como os itens de dados são codificados, bem como as operações que podem ser realizadas nesses dados. Por exemplo, o tipo **inteiro** se refere a dados numéricos que consistem em números inteiros, provavelmente armazenados segundo a notação de complemento de dois. As operações que podem ser realizadas nos números inteiros incluem as tradicionais operações aritméticas e a comparação de tamanhos relativos, como determinar se um valor é maior que outro. O tipo **real** (às vezes chamado **float**) refere-se a dados numéricos que podem conter valores outros que os números inteiros, provavelmente armazenados em notação de vírgula flutuante. As operações realizadas em dados do tipo real são similares às realizadas nos dados do tipo inteiro. Note, porém, que a atividade exigida para somar dois números do tipo real difere da que se utiliza na soma de dois inteiros.

Suponha, então, que desejemos usar a variável `LimitePeso` em um programa para nos referir a uma área de memória que contenha um valor numérico codificado em notação de complemento de dois. Nas linguagens C, C++, Java e C#, usamos a instrução

```
int LimitePeso;
```

que significa “O nome `LimitePeso` será usado mais tarde no programa com referência à área de memória que contém um valor armazenado segundo a notação de complemento de dois”. Múltiplas variáveis do mesmo tipo normalmente podem ser declaradas na mesma instrução. Por exemplo, a instrução

```
int Altura, Largura;
```

declara que `Altura` e `Largura` são variáveis do tipo inteiro. Além disso, a maioria das linguagens permite que seja atribuído um valor inicial às variáveis quando elas são declaradas. Assim,

```
int LimitePeso = 100;
```

não apenas declara `LimitePeso` como uma variável do tipo inteiro, mas também atribui-lhe o valor inicial 100.

Outros tipos de dados comuns incluem o caractere e o booleano. O tipo **caractere** se refere a dados que consistem em símbolos gráficos, provavelmente armazenados em ASCII ou Unicode. As operações que manipulam tais dados incluem comparações como determinar se um símbolo ocorre antes de outro, na ordem alfabética, verificar se uma cadeia de símbolos aparece como parte de uma outra e concatenar uma cadeia de símbolos ao final de uma outra para formar uma cadeia mais longa.

O tipo **booleano** se refere a dados que podem assumir somente os valores verdadeiro ou falso. Operações com dados de tipo booleano incluem consultas para verificar se o valor corrente é verdadeiro ou falso. Por exemplo, se a variável `FimDaLista` for declarada do tipo booleano, então a instrução da forma

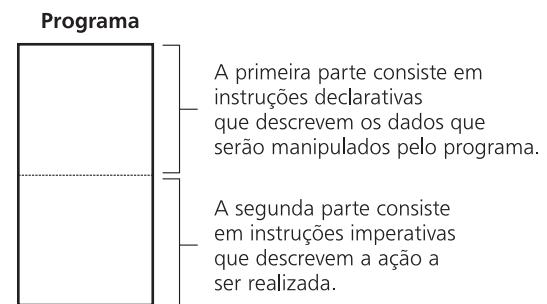


Figura 5.4 A composição de um típico programa imperativo ou unidade de programa.

se (FimDaLista) então (...) senão (...)
será razoável.

Outros tipos de dados que ainda não se tornaram primitivas comuns nas linguagens de programação incluem imagem, áudio, vídeo e hipertexto. Contudo, tipos como GIF, JPEG e HTML podem rapidamente se tornar tão comuns como inteiro e real. De fato, a linguagem Java, que contém ferramentas para manipular esses tipos, é um passo nessa direção.

A Figura 5.5 apresenta as mesmas instruções de declaração expressas em diversas linguagens tomadas como exemplos. (As variáveis Comprimento e Largura são declaradas como reais as variáveis Valor, Imposto e Total, como inteiros e a variável Simbolo é declarada como caractere. Note que as linguagens deferem mais no estilo que no conteúdo. Na Seção 5.4, veremos como um tradutor usa o conhecimento que ele obtém a partir de tais instruções declarativas para ajudar a traduzir um programa em linguagem de alto nível para linguagem de máquina. Por ora, devemos notar que essa informação pode ser usada na identificação de erros. Por exemplo, uma instrução que indique a adição de duas variáveis declaradas como caracteres provavelmente indica um erro.

Estruturas de dados

Além do tipo de dados, as variáveis em um programa freqüentemente são associadas com uma **estrutura de dados**, que se refere à forma conceitual dos dados. Por exemplo, um texto normalmente é visto como uma longa cadeia de caracteres, enquanto os registros de vendas podem ser vistos como uma tabela retangular de valores numéricos, onde cada linha representa as vendas feitas por um determinado funcionário e cada coluna, as vendas efetuadas em cada dia.

Uma estrutura de dados comum é o **arranjo homogêneo**, ou **matriz**, que é um bloco de valores do mesmo tipo, como uma lista unidimensional, uma tabela de duas dimensões, com linhas e colunas, ou tabelas com mais dimensões. Para declarar tais matrizes, a maioria das linguagens de programação utiliza instruções de declaração, nas quais o comprimento de cada dimensão da matriz é especificado. Por exemplo, a Figura 5.6 apresenta a estrutura conceitual estabelecida pela declaração

```
int Escores [2] [9];
```

a. Declarações de variáveis em Pascal

```
var
  Comprimento, Largura:    real;
  Valor, Imposto, Total: integer;
  Simbolo:                 char;
```

b. Declarações de variáveis em linguagem C, C++, C# e Java

```
float Comprimento, Largura;
int Valor, Imposto, Total;
char Simbolo;
```

c. Declarações de variáveis em linguagem FORTRAN

```
REAL Comprimento, Largura
INTEGER Valor, Imposto, Total
CHARACTER Simbolo
```

na linguagem C, que significa “A variável Escores será usada na unidade de programa a seguir para se referir a uma matriz bidimensional de inteiros com duas linhas e nove colunas”. A mesma declaração em FORTRAN seria escrita como

```
INTEGER Escores (2, 9)
```

Uma vez declarada uma matriz, é possível referenciá-la em sua totalidade, pelo seu nome, ou identificar um de seus componentes individuais por meio de valores inteiros chamados **índices** que especificam a linha, coluna e assim por diante. Contudo, a faixa desses índices varia de linguagem para linguagem. Por exemplo, no C (e em suas derivadas C++, Java e C#) os índices começam em 0, o que significa que o elemento na segunda linha e quarta coluna da matriz Escores (definida acima) seria referenciado por Escores [1] [3] e o elemento da primeira linha e primeira coluna, por Escores [0] [0]. Os índices, porém, começam em 1 em programas FORTRAN, de modo que o elemen-

Figura 5.5 Declarações de variáveis em diferentes linguagens.

to da segunda linha e quarta coluna seria referenciado por Escores (2, 4). (Veja novamente a Figura 5.6).

Algumas linguagens fornecem ao programador uma flexibilidade significativa no estabelecimento das faixas de índices usadas para referenciar matrizes. Por exemplo, a instrução

```
Escores: array [3..4,
               12..20] of integer;
```

em Pascal declara a variável Escores como uma matriz bidimensional de inteiros como antes, exceto em que as linhas são identificadas pelos valores 3 e 4 e as colunas, numeradas de 12 a 20. Assim, o elemento da segunda linha e quarta coluna seria referenciado por Escores [4, 15].

Em contraste com o arranjo homogêneo, no qual todos os itens de dados são do mesmo tipo, um **arranjo heterogêneo** é um bloco de dados no qual diferentes elementos podem ter diferentes tipos. Por exemplo, um bloco de dados que se refere a um funcionário pode consistir em um elemento chamado Nome, do tipo caractere, um elemento chamado Idade do tipo inteiro, e um elemento chamado NiveldeQualificacao do tipo real. A Figura 5.7 mostra como tal arranjo pode ser declarado em C e em Pascal.

Um componente de um arranjo heterogêneo normalmente é referenciado pelo nome do arranjo seguido por um ponto e o nome do componente. Por exemplo, o componente Idade no arranjo Funcionario da Figura 5.7 seria referenciado por Funcionario.Idade.

No Capítulo 7, veremos como as estruturas conceituais como os arranjos são de fato implementadas em um computador. Mais especificamente, aprenderemos que os dados contidos em um arranjo podem estar espalhados em uma larga área de memória principal ou do armazenamento em massa. Isso explica por que nos referimos às estruturas de dados como formas *conceituais* de dados. De fato, a disposição real dentro do sistema de armazenamento da máquina pode ser bem diferente da disposição conceitual.

Constantes e literais

Em alguns casos, um valor fixo, predeterminado, é usado em um programa. Por exemplo, um programa para o controle do tráfego aéreo nas proximidades de um aeroporto pode conter numerosas referências à sua altitude acima do nível do mar. Ao escrever tal programa, pode-se incluir explicitamente este valor, digamos 645 pés

Escores

			1					

Escores (2, 4) em FORTRAN, em que os índices começam em um.

Escores [1][3] em C e suas derivadas, em que os índices começam em zero.

Figura 5.6 Uma matriz bidimensional com duas linhas e nove colunas.

a. Declarações de arranjos em linguagem Pascal

var

```
Funcionario: record;
  Nome: packed array[1..8] of char;
  Idade: integer;
  NiveldeQualificacao: real
end
```

b. Declaração de arranjos em linguagem C

```
struct
  {char Nome [8];
  int Idade;
  float NiveldeQualificacao;
} Funcionarios;
```

c. Organização conceitual do arranjo

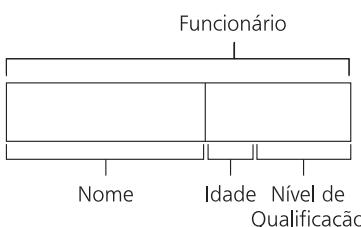


Figura 5.7 Declaração de arranjos heterogêneos, em Pascal e C.

de altura, todas as vezes que for necessário. O aparecimento explícito de um valor no programa é chamado de **literal**. A utilização de literais gera instruções com o seguinte aspecto:

```
AltitudeEfetiva ← Altimetro + 645
```

onde AltitudeEfetiva e Altimetro são variáveis e 645, é um literal.

Nem sempre, o uso de literais é uma boa prática de programação, pois eles podem mascarar o significado das instruções em que aparecem. Por exemplo, ao ler a instrução anterior, como o leitor poderá saber o que representa o valor 645? Além disso, os literais dificultam a tarefa de modificar o programa quando necessário. Se o nosso programa de tráfego aéreo for instalado em outro aeroporto, será necessário modificar todas as referências feitas à altitude do aeroporto. Se o literal 645 for usado em todas essas referências no programa, todas deverão ser localizadas e modificadas. O problema se agrava se o literal 645 também se referir a uma outra grandeza que não seja a altitude do aeroporto. Como saberemos quais ocorrências de 645 devem ser modificadas ou permanecer inalteradas?

Para resolver esses problemas, as linguagens de programação permitem que sejam atribuídos nomes descritivos a valores específicos e inalteráveis. Tais nomes são chamados **constantes**. Como exemplo, na linguagem Ada, a instrução declarativa

```
AltitudeDoAeroporto constant Integer:= 645;
```

associa à constante AltitudeDoAeroporto o valor 645 (que é considerado do tipo inteiro). O conceito similar em Java é expresso por

```
final int AltitudeDoAeroporto = 645;
```

e em C++ e C# a instrução apareceria como

```
const AltitudeDoAeroporto = 645;
```

Segundo essas declarações, o nome descritivo AltitudeDoAeroporto será utilizado no lugar do literal 645. Assim, usando esta constante em nosso pseudocódigo, a instrução

```
AltitudeEfetiva ← Altimetro + 645
```

seria reescrita como:

```
AltitudeEfetiva ← Altimetro + AltitudeDoAeroporto
```

a qual representa melhor o significado da instrução. Além disso, se tais constantes forem usadas em lugar de literais, e se o programa for instalado em outro aeroporto cuja altitude seja de 267 pés, então alterar a instrução declarativa na qual a constante é definida será tudo o que se precisa fazer para que todas as referências a tal altitude no programa representem corretamente a altitude do novo aeroporto.

Instruções de atribuição

Uma vez declarada a terminologia especial a ser usada em um programa (como a de variáveis e constantes), um programador pode iniciar a descrição do algoritmo envolvido. Isto é feito por meio das instruções imperativas, das quais a mais básica é a **instrução de atribuição**, que especifica um valor a ser atribuído a uma variável (ou, mais precisamente, a ser armazenado na área de memória identificada pela variável). Tal instrução em geral apresenta a forma sintática de uma variável, seguida por um símbolo que representa a operação de atribuição, seguido por uma expressão que indica o valor a ser atribuído. A semântica desta instrução significa que a expressão será avaliada e o resultado, associado como valor da variável. Por exemplo, a instrução

```
Z = X + Y;
```

em C, C++, C# e Java solicita que a soma de X e Y seja atribuída à variável Z. Em Ada e Pascal, a instrução equivalente apareceria como:

```
Z := X + Y;
```

Note-se que estas instruções só diferem em sintaxe quanto ao operador de atribuição, que nas linguagens C, C++ e Java é apenas o símbolo “=”, porém em Ada e Pascal é o símbolo “:=”. Talvez uma notação melhor para o operador de atribuição seja encontrada na APL, linguagem desenvolvida por Kenneth E. Iverson em 1962. (APL é abreviatura de A Programming Language). Ela usa uma seta para representar a atribuição. Assim, a instrução anterior seria expressa como

```
Z ← X + Y
```

em APL (bem como em nosso pseudocódigo do Capítulo 4).

Boa parte do poder das instruções de atribuição vem do escopo das expressões que podem aparecer na parte direita da instrução. Em geral, qualquer expressão algébrica pode ser usada com as operações aritméticas de adição, subtração, multiplicação e divisão, normalmente representadas pelos símbolos +, -, × e ÷, respectivamente. No entanto, as linguagens diferem quanto à forma como as expressões são interpretadas. Por exemplo, a expressão $2 \times 4 + 6 \div 2$ gera o valor 14 se for calculada da direita para a esquerda, ou 7, se processada da esquerda para a direita. Estas ambigüidades geralmente são resolvidas por regras de **precedência de operador**, o que significa que certas operações possuem prioridade em relação a outras. As regras tradicionais da álgebra ditam que as operações de multiplicação e divisão têm prioridade sobre as de adição e subtração, ou seja, são executadas antes das adições e subtrações. Seguindo esta convenção, a expressão anterior geraria o valor 11. Em muitas linguagens, são utilizados parênteses para alterar a prioridade dos operadores. Assim, da expressão $2 \times (4 + 6) \div 2$ resulta o valor 10.

Expressões em instruções de atribuição também podem envolver operações diferentes das algébricas tradicionais. Por exemplo, se `Primeiro` e `Ultimo` são variáveis associadas a cadeias de caracteres, então a instrução FORTRAN

```
Ambos = Primeiro // Ultimo
```

atribui à variável `Ambos` a cadeia de caracteres resultante da concatenação dos valores de `Primeiro` e de `Ultimo`. Assim, se `Primeiro` e `Ultimo` estão associados às cadeias *abra* e *cadabra*, respectivamente, é atribuída a `Ambos` a cadeia *abracadabra*.

Muitas linguagens de programação permitem que um mesmo símbolo represente mais de um tipo de operação. Nestes casos, o seu significado é determinado pelos tipos de dados dos operandos. Por exemplo, o símbolo + tradicionalmente indica adição, quando seus operandos são numéricos, mas em algumas linguagens, como Java, também indica concatenação, quando seus operandos são cadeias de caracteres. Tal multiplicidade de uso de um mesmo símbolo é chamada **sobrecarga**^{*}.

Instruções de controle

Instruções de controle são instruções imperativas destinadas a alterar a seqüência de execução das instruções de um programa. De todas as instruções, as provenientes deste grupo provavelmente foram as que receberam mais atenção e geraram as maiores controvérsias. A vilã principal é a mais simples de todas as instrução de controle, a instrução `goto`. Ela permite controlar a seqüência de execução mudando para outra posição do programa que tenha sido etiquetada, para este propósito, por meio de um nome ou número. Portanto, é apenas uma aplicação direta da instrução de desvio, em linguagem de máquina. Este recurso introduz, em uma linguagem de programação de alto nível, o problema de permitir que programadores escrevam ninhos de rato como:

```
goto 40
20  Aplicar o procedimento Evadir
```

*N. de T. Em inglês, *overloading*.

```

        goto 70
40   if (NivelCriptonita < DoseLetal) goto 60
        goto 20
60   Aplicar o procedimento SocorrerDonzela
70   ..

```

quando uma única instrução como:

```

if (NivelCriptonita < DoseLetal)
then (Aplicar o procedimento SocorrerDonzela)
else (Aplicar o procedimento Evadir)

```

seria suficiente.

Para evitar essas complicações, as linguagens modernas são projetadas com instruções de controle que permitem expressar certas estruturas de ramificação com uma única instrução. A Figura 5.8 apresenta algumas estruturas de ramificação comuns e as instruções de controle fornecidas por várias linguagens de programação para representar essas estruturas.

Note que as duas primeiras estruturas já foram encontradas no Capítulo 4. Elas são representadas por `se-senão` e `enquanto` do nosso pseudocódigo. A terceira estrutura, conhecida como estrutura `case`, pode ser vista como uma extensão da estrutura `se-senão`. Enquanto esta permite duas opções, o `case` permite selecionar entre muitas opções.

Outra estrutura muito conhecida, geralmente chamada estrutura `para`, bem como a sua representação em várias linguagens, está mostrada na Figura 5.9. Ela é uma estrutura iterativa semelhante à instrução `enquanto` do nosso pseudocódigo. A diferença é que as fases de iniciação, modificação e terminação desta estrutura iterativa estão incorporadas em uma única instrução, a qual é conveniente quando o corpo da iteração deve ser executado para cada elemento de um conjunto ordenado de valores. A instrução da Figura 5.9 faz com que o

Estrutura de controle C, C++, C# e Java

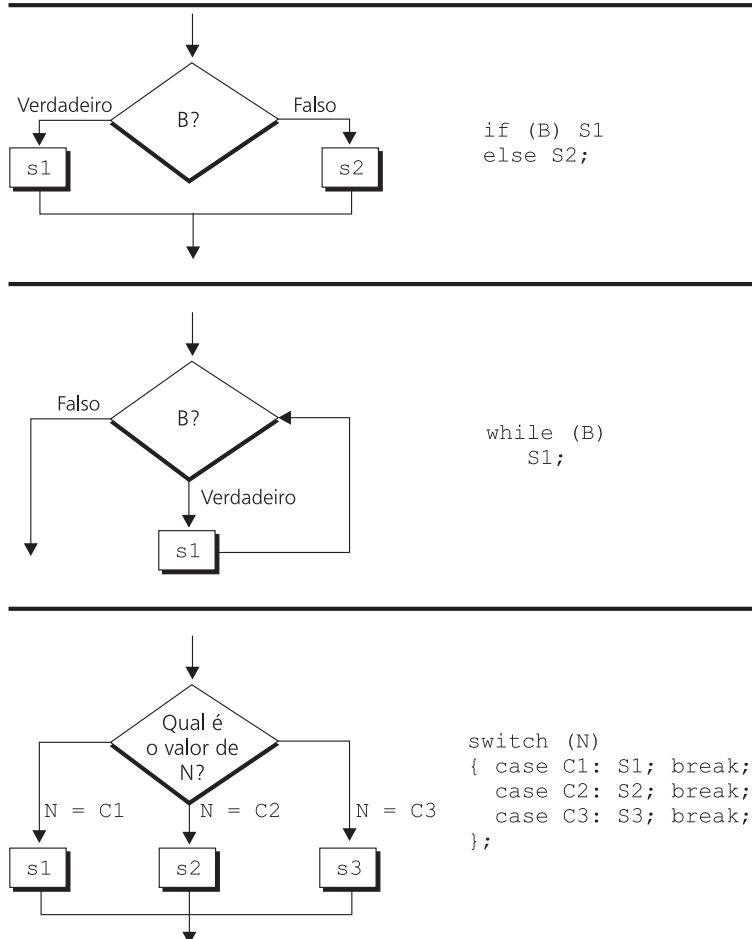


Figura 5.8 Estruturas de controle e suas representações em C, C++, C# e Java.

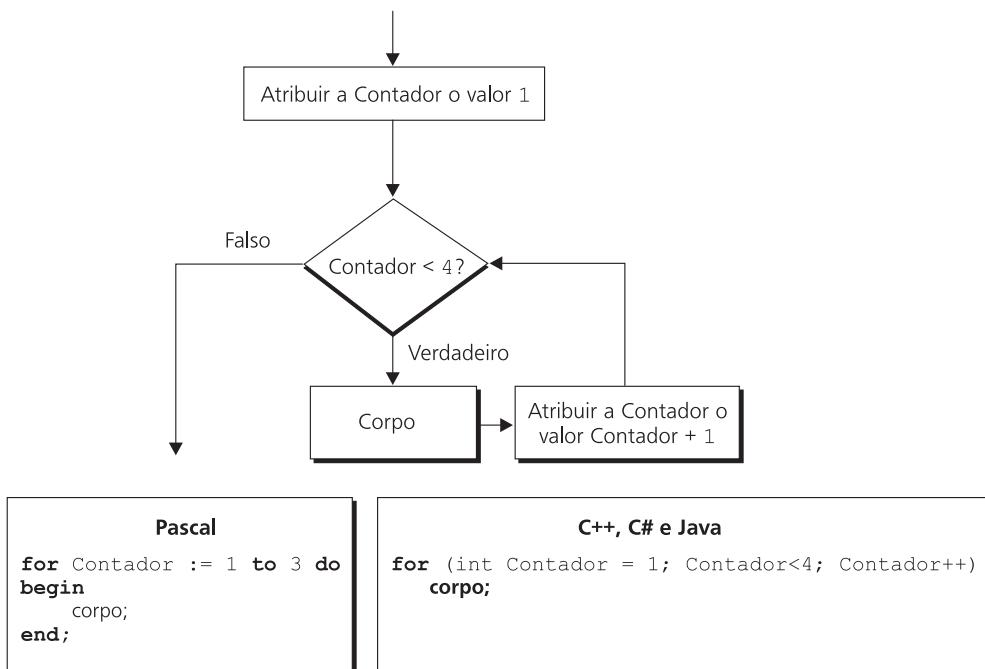


Figura 5.9 A estrutura para e sua representação nas linguagens Pascal, C++, C# e Java.

corpo da iteração seja executado de modo que a variável Contador inicialmente tenha o valor 1, em seguida o valor 2, e finalmente, o valor 3.

O objetivo desses exemplos é demonstrar que estruturas ramificadas genéricas aparecem, com pequenas variações, nas linguagens de programação imperativa e orientada a objeto. Um resultado teórico surpreendente da Ciência da Computação é que a capacidade de expressar somente algumas destas estruturas assegura que uma linguagem de programação tenha meios para expressar uma solução para qualquer problema que tenha solução algorítmica. Analisamos este fato no Capítulo 11. Por ora, mencionamos apenas que aprender uma linguagem de programação não é uma tarefa interminável de aprendizado de diferentes instruções de controle. Na verdade, as principais instruções de controle disponíveis nas linguagens de programação atuais são essencialmente variações das instruções aqui identificadas.

A escolha das estruturas de controle a serem incorporadas a uma linguagem é uma importante decisão de projeto. O objetivo é criar uma linguagem que expresse algoritmos de forma legível e que auxilie o programador nesta tarefa. Isto é possível restringindo o uso desses recursos — os quais, historicamente falando, têm conduzido a programações de baixa qualidade — e encorajando o uso de recursos mais bem projetados. O resultado é a **programação estruturada**, uma prática pouco compreendida que abrange uma metodologia organizada de projeto, combinada com o uso disciplinado das instruções de controle da linguagem. A idéia é produzir um programa que possa ser prontamente compreendido e demonstrar que ele atende às suas especificações.

Comentários

A experiência mostra que, a despeito do quanto uma linguagem de programação seja bem projetada, ou do quanto os seus recursos sejam bem utilizados, informações adicionais são necessárias, ou no mínimo úteis, para a compreensão humana de qualquer programa de porte razoável. Por isso, as linguagens de

Visual Basic

Visual Basic é uma linguagem orientada a objeto que foi desenvolvida pela Microsoft como ferramenta para que os usuários do sistema operacional Windows da Microsoft pudessem desenvolver suas próprias aplicações com interfaces gráficas (GUI). Na realidade, o Visual Basic é mais que uma linguagem – é um pacote completo de desenvolvimento de *software* que permite a um programador construir um formulário gráfico a partir de componentes predefinidos (como botões, caixas de verificação, caixas de texto, barras de rolagem etc.) e personalizar esses componentes por meio da descrição de como eles devem reagir a vários eventos. No caso de um botão, por exemplo, o programador descreve o que deve acontecer quando esse botão é clicado. No Capítulo 6, veremos que essa estratégia de construir *software* a partir de componentes predefinidos representa a tendência atual nas técnicas de desenvolvimento de *software*.

A popularidade do sistema operacional Windows combinada com a conveniência do pacote de desenvolvimento Visual Basic tem feito deste último a linguagem de programação mais utilizada atualmente. Por outro lado, o fato de o Visual Basic residir somente em ambiente de *software* da Microsoft impede que ele seja considerado uma linguagem de programação de propósito geral pela comunidade de computação.

programação incorporam formas sintáticas, chamadas **comentários**, que permitem a inserção de textos explicativos em um programa. Os comentários são ignorados pelo tradutor, portanto, sua presença ou ausência não afeta o programa sob o ponto de vista da máquina. A versão em linguagem de máquina do programa produzida por um tradutor será a mesma com ou sem comentários, mas a informação fornecida por essas declarações constitui parte importante do programa sob o ponto de vista humano. Sem essa documentação, programas grandes e complexos podem facilmente ultrapassar os limites da capacidade humana de compreensão de um programador.

Para permitir a inclusão de comentários internos em um programa, as linguagens de programação apresentam duas maneiras mais conhecidas para separar os comentários do restante do programa. Uma delas é delimitar todo o comentário por meio de marcadores especiais, um no início do comentário, e outro, no seu final. A outra forma consiste em marcar apenas o início do comentário e permitir que este ocupe todo o restante da linha, à direita do marcador. Encontramos exemplos das duas técnicas em C++, C# e Java. Essas linguagens permitem que os comentários sejam limitados por /* e */, mas também permitem que iniciem com // e se estendam até o final da linha. Assim, em C++, C# e Java, os textos

```
/* Isto é um comentário. */
```

e

```
// Isto é um comentário.
```

são declarações de comentário válidas.

Algumas palavras acerca do que constitui um comentário importante. Programadores iniciantes, quando solicitados a fazer comentários para documentação interna, tendem a acompanhar uma instrução como:

```
AnguloAproximado= AnguloDeslizamento + InclinacaoHiperEspaco
```

com um comentário do tipo “Calcule AnguloAproximado somando AnguloDeslizamento e InclinacaoHiperEspaco”. Tal redundância simplesmente aumenta o tamanho do programa, sem elucidá-lo. Convém lembrar que o objetivo de um comentário é explicar o programa, não repeti-lo. Um comentário mais apropriado, associado à instrução anterior, seria explicar por que o ângulo aproximado está sendo calculado (caso isso não seja óbvio). Por exemplo, o comentário “O ângulo aproximado será usado mais tarde para calcular a velocidade Jettison no campo de força, e depois não será mais necessário” é mais útil do que o anterior.

Além disso, um programa cujos comentários ficam espalhados entre as instruções pode ser mais difícil de compreender do que outro, sem qualquer comentário. Uma boa prática consiste em coletar em um só lugar, talvez no seu início, todos os comentários que se referem a uma determinada unidade de programa. Isto estabelece uma localização centralizada, em que o leitor poderá buscar as explicações de que necessita. É também um local ideal para declarar o objetivo e as características gerais deste módulo de programa. Se este arranjo for adotado para todos os módulos de programa, este irá apresentar uma boa uniformidade, pois cada módulo consiste em um bloco de instruções explicativas, seguido da apresentação formal do módulo de programa propriamente dito. Tal uniformidade em um programa facilita a sua leitura.



QUESTÕES/EXERCÍCIOS

1. Por que usar constantes em um programa é considerado um estilo de programação melhor do que empregar literais?
2. Qual é a diferença entre uma instrução declarativa e uma imperativa?
3. Cite alguns tipos de dados mais comuns.
4. Identifique algumas das estruturas de controle mais comuns, encontradas em linguagens de programação imperativas e orientadas a objeto.
5. Qual é a diferença entre um arranjo homogêneo e um heterogêneo?

5.3 Módulos de programas

Nos capítulos anteriores, vimos as vantagens de dividir programas grandes em módulos manipuláveis. Nesta seção, focalizaremos o conceito de procedimento, que é uma técnica básica para obter uma representação modular de um programa escrito em uma linguagem imperativa. Os procedimentos também são ferramentas usadas para descrever como os objetos devem responder aos vários estímulos em programas orientados a objeto.

Procedimentos

Um **procedimento**, em seu conceito geral, é um conjunto de instruções para realizar uma tarefa e pode ser usado como ferramenta abstrata por outras unidades de programa. O controle é transferido para o procedimento (por meio de uma instrução JUMP da linguagem de máquina) quando são solicitados os seus serviços e, mais tarde, devolvido ao módulo de programa original, após o término da execução do procedimento (Figura 5.10). O processo de transferir o controle para um procedimento geralmente é denominado *chamada*^{*} do procedimento. Referir-nos-emos a um módulo de programa que solicita a execução de um procedimento como módulo *chamador*.

Em muitos aspectos, um procedimento é um programa em miniatura e consiste em instruções declarativas que descrevem as variáveis usadas no procedimento seguidas de instruções imperativas, que descrevem os passos a serem realizados quando o procedimento é executado. Como regra geral, variáveis declaradas internamente em um procedimento são **variáveis locais**, ou seja, variáveis que só

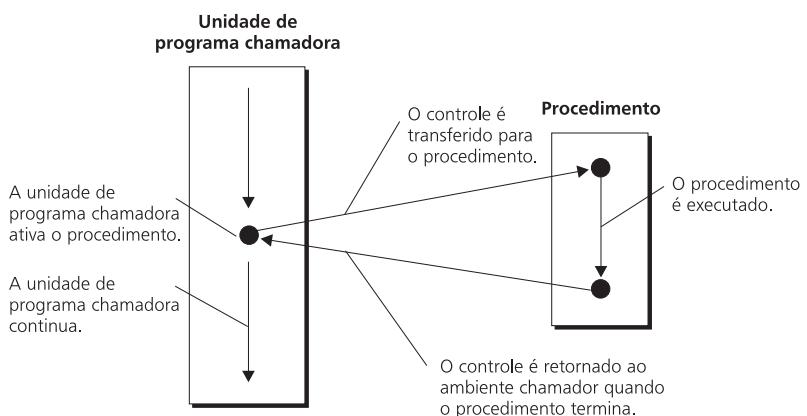


Figura 5.10

O fluxo de controle que envolve um procedimento.

*N. de T. Em inglês, *call*, *invocation*.

Sistemas de software dirigidos por eventos

No texto, consideramos casos em que os procedimentos são ativados como resultado de instruções em algum lugar no programa que explicitamente chamam o procedimento. Existem casos, contudo, em que os procedimentos são ativados implicitamente pela ocorrência de um evento. Exemplos são encontrados nas interfaces gráficas, onde o procedimento que descreve o que deve acontecer quando um botão é clicado não é ativado como resultado de uma chamada feita por outra unidade de programa, mas como resultado do clique no botão. Os sistemas de software nos quais os procedimentos são ativados dessa maneira são chamados sistemas **dirigidos por evento**. Em resumo, um sistema de software dirigido por evento consiste em procedimentos que descrevem o que deve acontecer como resultado de vários eventos. Quando o sistema é executado, esses procedimentos permanecem latentes até que os respectivos eventos ocorram — nesse momento, eles se tornam ativos, realizam suas tarefas e voltam à inatividade.

podem ser referenciadas no interior deste procedimento. Esta regra elimina qualquer confusão que possa surgir caso dois procedimentos independentes resolvam utilizar variáveis homônimas. (Variáveis que não se restringem a uma parte do programa são chamadas **variáveis globais**, no sentido em que são acessíveis em todo o programa. A maioria das linguagens fornece meios de declarar variáveis globais.)

Em nossas linguagens de programação tomadas como exemplos, os procedimentos são definidos praticamente da mesma maneira que em nosso pseudocódigo do Capítulo 4. A definição começa com uma declaração conhecida como **cabeçalho do procedimento**, que identifica, dentre outras coisas, o nome do procedimento. Seguindo esse cabeçalho, vêm as instruções que definem os detalhes do procedimento.

Contudo, em contraste com o nosso pseudocódigo informal do Capítulo 4, no qual solicitamos a execução de um procedimento por meio de uma instrução como “Aplicar o procedimento DesativarCriptônio”, a maioria das linguagens de programação modernas permite que os procedimentos sejam chamados simplesmente por seus nomes. Por exemplo, se `ObterNomes`, `OrdenarNomes` e `ImprimirNomes` forem procedimentos para obter, ordenar e imprimir uma lista de nomes, então um programa para obter, ordenar e depois imprimir a lista pode ser escrito como:

```
ObterNomes;
OrdenarNomes;
ImprimirNomes;
```

em vez de

```
Aplicar o procedimento ObterNomes;
Aplicar o procedimento OrdenarNomes;
Aplicar o procedimento ImprimirNomes;
```

Note que ao atribuir um nome a cada procedimento que indique a ação realizada por ele, essa forma condensada aparece como uma seqüência de comandos que refletem o significado do programa.

Parâmetros

Os parâmetros freqüentemente são escritos usando-se termos genéricos que são especificados quando o procedimento é aplicado. Por exemplo, a Figura 4.11 do capítulo anterior apresenta uma versão em pseudocódigo de um procedimento para ordenar listas, expresso em termos de listas genéricas, em vez de uma lista específica. Em nosso pseudocódigo, concordamos em identificar tais termos genéricos dentro de parênteses no cabeçalho do procedimento. Assim, o procedimento da Figura 4.11 começa com o cabeçalho

Procedimento Ordena (Lista)

e então prossegue com a descrição de como a lista deve ser ordenada, usando o termo *Lista* para se referir à lista em questão. Se queremos aplicar o procedimento para ordenar uma lista de convidados para um

casamento, precisamos meramente seguir as diretrizes do procedimento, assumindo que o termo genérico *Lista* se refere à lista de convidados. Se, porém, desejarmos ordenar uma lista de membros de uma corporação, deveremos interpretar o termo genérico *Lista* como uma referência à lista de membros.

Esses termos genéricos nos procedimentos são chamados **parâmetros**. Mais precisamente, os termos usados ao escrever o procedimento são chamados **parâmetros formais**, cujo significado preciso quando o procedimento é aplicado são chamados **parâmetros reais** ou **argumentos**. Em um certo sentido, os parâmetros formais representam encaixes do procedimento, nos quais os parâmetros reais são inseridos quando o procedimento é chamado. Na realidade, os parâmetros formais são variáveis do procedimento às quais são atribuídos valores (os parâmetros reais) quando o procedimento é executado.

Em geral, as linguagens de programação seguem o formato de nosso pseudocódigo para identificar os parâmetros formais em um procedimento, isto é, a maioria das linguagens de programação exige que, quando se define um procedimento, os parâmetros formais sejam listados entre parênteses no cabeçalho deste. Como exemplo, a Figura 5.11 apresenta a definição de um parâmetro chamado *PopulacaoProjetada*. Projetada como seria escrito na linguagem de programação C. O procedimento espera receber uma taxa de crescimento anual específica, quando é chamado. Baseado nessa taxa, o procedimento calcula a população projetada de uma espécie, pressupondo uma população inicial de 100, para os próximos 10 anos, e guarda esses valores em um vetor (matriz unidimensional) chamado *Populacao*.

A maioria das linguagens de programação também usa a notação entre parênteses para identificar os parâmetros reais quando um procedimento é chamado, isto é, a instrução que solicita a execução de um procedimento consiste no nome do procedimento seguido pela lista de parâmetros reais entre parênteses. Assim, uma instrução como

```
PopulacaoProjetada (0.03);
```

<p>Começar o cabeçalho com o termo "void" é a maneira como um programador C especifica que a unidade de programa é um procedimento e não uma função. Aprenderemos mais sobre as funções brevemente.</p>	<p>A lista de parâmetros formais. Note que o C, bem como muitas linguagens de programação, exige que o tipo de dados de cada parâmetro seja especificado.</p>
<pre>void PopulacaoProjetada (float TaxaCrescimento)</pre>	
<pre>{ int Ano;</pre> <p>Isto declara uma variável local denominada <i>Ano</i>.</p>	
<pre>Populacao[0] = 100.0; for (Ano = 0; Ano = < 10; Ano++) Populacao[Ano+1] = Populacao[Ano] + (Populacao[Ano] * TaxaCrescimento); }</pre>	
<p>Estas instruções descrevem como as populações são calculadas e armazenadas no vetor global chamado <i>Populacao</i>.</p>	

Figura 5.11 O procedimento *PopulacaoProjetada* escrito na linguagem de programação C.

pode ser usada em um programa C para chamar o procedimento PopulacaoProjetada da Figura 5.11 usando como TaxaCrescimento o valor 0,03.

Quando mais de um parâmetro está envolvido, os parâmetros reais são associados um a um com os parâmetros formais listados no cabeçalho do procedimento — o primeiro parâmetro real é associado com o primeiro parâmetro formal etc. Então, os valores dos parâmetros reais são efetivamente transferidos a seus parâmetros formais correspondentes. E o procedimento é executado.

Para enfatizar esse ponto, suponha que o procedimento `ImprimeCheque` tenha sido definido com o cabeçalho seguinte

```
procedure ImprimeCheque (Credor, Quantia)
```

onde `Credor` e `Quantia` são parâmetros formais usados dentro do procedimento para se referir à pessoa a quem o cheque deve ser pago e à quantia a ser paga, respectivamente. Então, chamar o procedimento com a instrução

```
ImprimeCheque ("John Doe", 150)
```

causa a execução do procedimento, com o parâmetro formal `Credor` sendo associado com o parâmetro real `John Doe`, e o parâmetro formal `Quantia`, com o valor 150. Entretanto, chamar o procedimento com a instrução

```
ImprimeCheque (150, "John Doe")
```

causaria a associação do valor 150 ao parâmetro formal `Credor` e do nome `John Doe` ao parâmetro formal `Quantia`, o que levaria a resultados errôneos.

A tarefa de transferir dados entre parâmetros reais e formais é feita de diversos modos por diferentes linguagens de programação. Em algumas linguagens, uma duplicata dos dados representados pelos parâmetros reais é produzida e passada ao procedimento. Usando essa estratégia, qualquer alteração de dados feita pelo procedimento é refletida apenas na duplicata — os dados na unidade de programa chamadora não se alteram. Dizemos que esses parâmetros são **passados por valor**. Note que a passagem de parâmetros por valor protege os dados da unidade chamadora de serem equivocadamente alterados por um procedimento mal projetado. Por exemplo, se uma unidade chamadora passa um nome de funcionário para um procedimento, ela não quer que o procedimento altere esse nome.

Infelizmente, a passagem de parâmetros por valor é inefficiente quando os parâmetros representam grandes blocos de dados. Uma maneira mais eficiente de passar parâmetros a um procedimento é dar-lhe acesso direto aos parâmetros reais fornecendo seus endereços na unidade chamadora. Nesse caso, dizemos que os parâmetros são **passados por referência**. Note que a passagem de parâmetros por referência permite que o procedimento modifique dados residentes no ambiente chamador. Essa estratégia é desejável no caso de um procedimento de ordenação em listas. Sem dúvida, o objetivo de se chamar tal procedimento é causar mudança nas listas.

Como exemplo, suponhamos que o procedimento `Demo` seja definido como

```
Procedure Demo(Formal)
```

```
Formal ← Formal + 1;
```

Além disso, suponha que a variável `Atual` tenha o valor 5 e chamemos `Demo` com a instrução

```
Demo(Atual)
```

Então, se os parâmetros forem passados por valor, a mudança em `Formal` dentro do procedimento não será refletida na variável `Atual` (Figura 5.12). Entretanto, se os parâmetros forem passados por referência, o valor de `Atual` será incrementado em 1 (Figura 5.13).

Linguagens de programação diferentes fornecem diferentes técnicas de passagem de parâmetros, mas em todos os casos o uso de parâmetros permite que um procedimento seja escrito em um contexto genérico e aplicado a dados específicos no momento apropriado.

Funções

Fazemos uma pausa para considerar uma ligeira variação do conceito de procedimento, que é encontrada em muitas linguagens de programação. Às vezes, o objetivo do procedimento é produzir um valor, em vez de realizar uma ação. (Considere a distinção sutil entre um procedimento cujo objetivo é estimar o número de produtos que serão vendidos e outro cujo objetivo é ordenar uma lista — a ênfase no primeiro é a produção de um valor, enquanto no segundo é a realização de uma ação.) Nesses casos, o “procedimento” pode ser implementado como uma função. Aqui, o termo **função** se refere a uma unidade de programa similar a um procedimento, exceto em que um valor é transferido de volta à unidade chamadora como o “valor da função”, isto é, como consequência da execução da função, um valor será calculado e enviado de volta à unidade de programa chamadora. Esse valor poderá então ser armazenado em uma variável para referência posterior, ou ser usado imediatamente em uma computação. Por exemplo, um programador C, C++, Java ou C# pode escrever

```
VendasProjetadasJan = VendasEstimadas(Janeiro);
```

para solicitar que a variável `VendasProjetadasJan` contenha o resultado da aplicação da função `VendasEstimadas`, que determina quantos produtos devem ser vendidos em janeiro. Ou então, o programador pode escrever

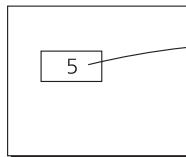
```
If (UltimaVendaJan < VendasEstimadas(Janeiro))...
else...
```

para especificar diferentes ações a serem realizadas, dependendo de as vendas esperadas de janeiro serem melhores do que as de janeiro passado. Note que, no segundo caso, o valor calculado pela função não é armazenado.

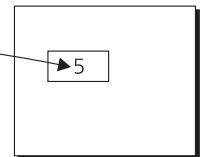
As funções são definidas em um programa de uma maneira muito parecida com a usada nos procedimentos. A diferença é que um cabeçalho de função geralmente começa com a especificação do

- a.** Quando o procedimento é chamado, uma cópia dos dados é passada para ele

Ambiente chamador

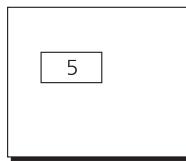


Ambiente do procedimento

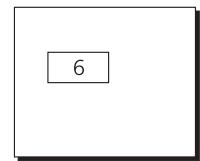


- b.** e o procedimento manipula a sua cópia.

Ambiente chamador



Ambiente do procedimento



- c.** Assim, quando o procedimento termina, o ambiente chamador não é alterado.

Ambiente chamador

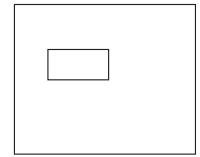
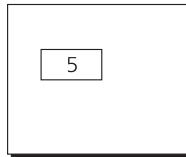
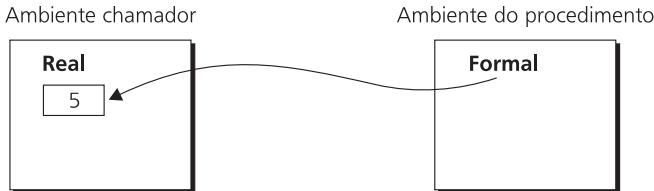
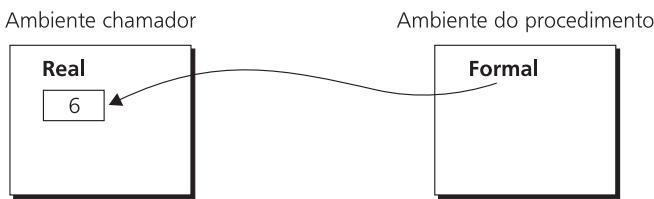


Figura 5.12 Execução do procedimento *Demo* com passagem de parâmetros por valor.

- a. Quando o procedimento é chamado, o parâmetro formal se torna uma referência ao real.



- b. Assim, mudanças indicadas pelo procedimento são feitas no parâmetro real



- c. e, portanto, preservadas após o término do procedimento.



Figura 5.13 Execução do procedimento *Demo* com passagem de parâmetros por referência.

do a operação for solicitada. Esta é a maneira como a maioria das linguagens implementa suas operações de entrada e saída, exceto em que os procedimentos e funções usados normalmente são fornecidos em um pacote pré-escrito chamado biblioteca, de onde são prontamente acessíveis.

Como exemplo, para ler um valor a partir do teclado e associá-lo a uma variável chamada *Valor*, um programador escreveria, em Pascal:

```
readln (Valor);
```

e, para escrever o valor na tela do monitor,

```
writeln (Valor);
```

Observe que a sintaxe é de chamada a um procedimento, com uma lista de parâmetros. De fato, *readln** e *writeln*** são os nomes de procedimentos pré-escritos para realizar as tarefas solicitadas.

De modo semelhante, um programador C usaria os procedimentos (teoricamente, eles são funções no vernáculo dos programadores C) *scanf* e *printf* para executar entrada e saída. Contudo,

tipo de dado do valor a ser retornado, e a definição normalmente termina com uma instrução de retorno na qual o valor a ser retornado é especificado. A Figura 5.14 apresenta a definição de como uma função chamada *VolumeCilindro* poderia ser escrita na linguagem C. (Na realidade, um programador C usaria uma forma mais sucinta, mas usaremos essa versão um tanto prolixas por razões pedagógicas). Quando chamada, a função recebe valores específicos para os parâmetros formais *Raio* e *Altura* e retorna o resultado do cálculo do volume de um cilindro com essas dimensões. Assim, a função poderia ser usada em qualquer parte do programa por meio de uma instrução como

```
Custo = CustoPorUnidadeVolume  
* VolumeCilindro(3.45, 12.7);
```

para determinar o custo do conteúdo de um cilindro com raio 3,45 e altura 12,7.

Instruções de entrada e saída

Procedimentos e funções proporcionam meios para ampliar as características de uma linguagem de programação. Se a linguagem não oferecer como primitiva uma determinada operação, pode-se á escrever um procedimento ou função para executar tal tarefa e depois chamar este módulo de programa quando a operação for solicitada.

*N. de T. Lê-se *read line* (em português, *leia uma linha*).

**N. de T. Lê-se *write line* (em português, *escreva uma linha*).

existe uma diferença significativa entre o `printf` do C e o `writeln` do Pascal, na maneira como os parâmetros são usados. O `writeln` do Pascal simplesmente exibe os parâmetros dados na ordem em que são listados, enquanto o `printf` do C espera que o primeiro parâmetro seja uma descrição de como os outros devem ser colocados na linha. Isto é, o primeiro parâmetro em uma chamada a `printf` é usado para descrever como os dados a serem exibidos serão formatados na tela do monitor. Essa abordagem freqüentemente é chamada de **E/S formatada**.

Por exemplo, um programador C escreveria

```
printf("%d %d\n", Valor1, Valor2);
```

para que os valores das variáveis `Valor1` e `Valor2` fossem exibidos, em notação decimal, em uma única linha. Note que o primeiro parâmetro é um esboço de como o texto deve aparecer na tela do monitor. Cada `%d` indica uma posição que será preenchida por um valor em notação decimal. O padrão `\n` indica que se deverá iniciar uma nova linha depois que tais valores tiverem sido mostrados. Assim, o primeiro parâmetro nesse exemplo indica que o texto a ser exibido consiste em um valor numérico em notação decimal, seguido por um espaço em branco, seguido por outro valor numérico, seguido por um retorno de carro e nova linha. Os parâmetros restantes fornecem os valores a serem inseridos na linha, na ordem em que são solicitados.

Como outro exemplo, se os valores de `Idade1` e `Idade2` forem 16 e 25, respectivamente, então a instrução:

```
printf("As idades são %d \n e %d.\n", Idade1, Idade2);
```

geraria a seguinte mensagem na tela do monitor:

```
As idades são 16
e 25.
```

Note que essas linhas foram produzidas com a inserção dos valores de `Idade1` e `Idade2` nas posições indicadas pelos marcadores `%d` no primeiro parâmetro (Figura 5.15). Sem a E/S formatada, um programador Pascal usaria a seqüência de duas instruções:

```
writeln ("As idades são", Idade1);
writeln ("e", Idade2, ".");
```

para obter o mesmo resultado.

Por serem linguagens orientadas a objeto, C++, Java e C# tratam as operações de entrada e saída como a transferência de dados de/para objeto. Em particular, o C fornece objetos pré-construídos, conhecidos como `cin` e `cout` (pronuncia-se *ci-in* e *ci-aut*) para representar o dispositivo padrão de entrada (provavelmente o teclado) e o dispositivo de saída (provavelmente a tela do monitor), respectivamente.

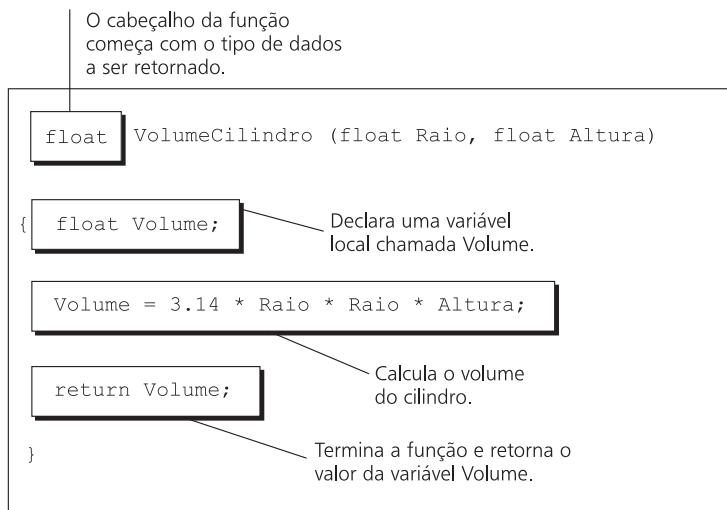


Figura 5.14 A função `VolumeCilindro` escrita na linguagem de programação C.

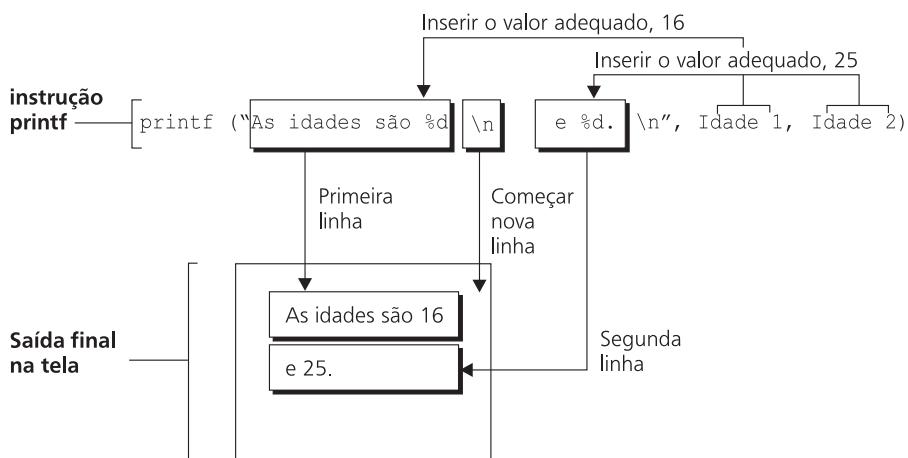


Figura 5.15 Um exemplo de saída formatada.

Os dados são então transferidos, destes objetos e para os mesmos, na forma de mensagens. Por exemplo, um valor pode ser obtido a partir do teclado e depositado na variável `Valor` com a seguinte instrução

```
cin >> Valor;
```

que informa ao objeto `cin` para atribuir o valor recebido à variável `Valor`. De maneira similar, a instrução:

```
cout << Valor;
```

diria ao objeto `cout` para exibir os dados contidos na variável `Valor` na tela do monitor.



QUESTÕES/EXERCÍCIOS

1. Qual é a diferença entre uma variável global e uma local?
2. Qual é a diferença entre um procedimento e uma função?
3. Por que muitas linguagens de programação implementam operações de entrada e saída na forma de chamadas de procedimentos?
4. Qual é a diferença entre um parâmetro formal e um argumento?
5. Quando escrevem em linguagens de programação modernas, os programadores tendem a usar verbos como nomes de procedimento e substantivos como nomes de função. Por quê?

5.4 Implementação de linguagens

Nesta seção, investigaremos o processo de conversão de um programa escrito em uma linguagem de alto nível para um formato de linguagem de máquina executável.

O processo de tradução

O processo de conversão de um programa de uma linguagem para outra é chamado **tradução**. O programa, em sua forma original, é chamado de **programa-fonte** e a versão traduzida, **programa-objeto**. O processo de tradução compreende três atividades — análise léxica, análise sintática e geração de código — que são processadas pelos módulos do tradutor conhecidos como **analisador léxico**, **analisador sintático** e **gerador de código** (Figura 5.16).

A análise léxica é o processo de reconhecer quais cadeias de símbolos do programa-fonte representam entidades indivisíveis. Por exemplo, os três caracteres 153 não devem ser interpretados como 1, seguido por 5, seguido por 3, mas são reconhecidos como um valor numérico apenas. Do mesmo modo, as palavras que aparecem no programa, embora compostas de caracteres individuais, devem ser interpretadas como unidades inseparáveis. A maioria dos seres humanos executa a atividade de análise léxica com um esforço consciente pequeno. Quando solicitados a ler em voz alta, pronunciamos palavras completas, em vez de caracteres individuais.

Assim, o analisador léxico lê o programa-fonte caractere a caractere identificando os agrupamentos de caracteres que representam unidades e classificando-as como valores numéricos, palavras, operadores aritméticos e assim por diante. À medida que a unidade é classificada, o analisador léxico gera um padrão de bits conhecido como **símbolo** (*token*) para representar a unidade e passa o símbolo para o analisador sintático. Durante esse processo, o analisador léxico ignora as declarações de comentário.

Assim, o analisador sintático vê o programa em termos de unidades léxicas (símbolos), em vez de caracteres individuais. O trabalho do analisador sintático é agrupar essas unidades em instruções. De fato, a análise sintática^{*} é o processo de identificação da estrutura gramatical do programa e de reconhecimento do papel de cada componente. São os detalhes técnicos da análise sintática que causam hesitação quando se lê:

The man the horse that won the race threw was not hurt.^{**}

Para simplificar o processo da análise sintática, as primeiras linguagens de programação exigiam que cada instrução do programa fosse posicionada de uma forma particular no texto impresso. Tais linguagens eram conhecidas como **linguagens de formato fixo**. Atualmente, a maioria é de **linguagens de formato livre**, o que significa que o posicionamento físico de instruções não é crítico para a sua interpretação. A vantagem dessas linguagens é propiciar aos programadores a possibilidade de organizar de modo legível o programa escrito, ou seja, de forma que simplifique sua análise. Para isso, é comum o uso da endentação para facilitar ao leitor a identificação da estrutura de uma instrução.

Em vez de:

```
if Custo < DinheiroVivo then pague em dinheiro vivo else use cartão de crédito
```



Figura 5.16 O processo de tradução.

*N. de T. Em inglês, *parsing*.

**N. de T. Em português, *O homem a quem o cavalo que venceu a corrida derrubou não foi ferido*.

é melhor que o programador escreva:

```
if Custo < DinheiroVivo
    then pague com dinheiro vivo
    else use cartão de crédito
```

Para a máquina efetuar a análise sintática de um programa escrito em uma linguagem de formato livre, a sintaxe da linguagem deve ser projetada de forma que a estrutura do programa possa ser identificada de maneira independente do espaçamento empregado no programa-fonte. Para isso, a maioria das linguagens de formato livre emprega símbolos de pontuação, como ponto-e-vírgula, para marcar o final de uma instrução, bem como **palavras-chave**, como *if*, *then* e *else* para o início de frases individuais. Essas palavras-chave muitas vezes são **palavras reservadas**, o que significa que não podem ser usadas no programa para outras finalidades.

O processo de análise sintática é feito com base em um conjunto de regras sintáticas, que definem a sintaxe da linguagem de programação. Uma forma de expressar tais regras é através de **diagramas de sintaxe**, representações gráficas da estrutura gramatical de um programa. A Figura 5.17 apresenta um diagrama de sintaxe da instrução *se-entao-senao* do nosso pseudocódigo do Capítulo 4. Este diagrama indica que uma estrutura *se-entao-senao* começa com a palavra *se*, seguida de uma *expressão booleana*, seguida pela palavra *entao* e, finalmente, por uma *Instrução*. Esta combinação pode ser ou não seguida por uma palavra *senao* e outra *Instrução*. Note que os termos de fato contidos em uma instrução *se-entao-senao* são representados em uma elipse, enquanto os que exigem explicação adicional, como *expressão booleana* e *instrução*, estão em retângulos. Termos que exigem explicação adicional (aqueles nos retângulos) são chamados **não-terminais**; os contidos nas elipses são os **terminais**. Na descrição completa da sintaxe de uma linguagem, os não-terminais são descritos por meio de diagramas adicionais.

Como um exemplo mais completo, a Figura 5.18 apresenta um conjunto de diagramas que descrevem a sintaxe de uma estrutura chamada *Expressão*, cuja finalidade é representar a estrutura de expressões aritméticas simples. O primeiro diagrama descreve uma *Expressão* formada por um *Termo* que pode aparecer isolado ou então seguido pelos símbolos + ou - e de outra *Expressão*. O segundo descreve um *termo* que consiste um único *Fator* ou um *Fator* seguido por um símbolo × ou ÷, seguido por outro termo. Finalmente, o último diagrama descreve um fator como um dos símbolos x, y ou z.

Pode-se representar, de forma gráfica, a maneira como uma cadeia adere à sintaxe descrita por um conjunto de diagramas de sintaxe, por meio de uma **árvore de sintaxe** (*parse tree*), como ilustrado na Figura 5.19, que apresenta uma árvore de sintaxe para a cadeia

$$x + y \times z$$

com base no conjunto de diagramas da Figura 5.18. Note-se que a raiz da árvore é o não-terminal *Expressão* e que cada nível mostra a maneira como os não-terminais daquele nível são decompostos, até que finalmente sejam obtidos os símbolos da própria cadeia analisada. A figura mostra exatamente como a cadeia $x + y \times z$ consiste em um *Termo* (que no caso é o *Fator* x), seguido do símbolo +, seguido de uma *Expressão* (que no caso é o *Termo* $y \times z$).

Implementação do Java

No caso de uma página Web com animação, o *software* de controle é transferido através da Internet juntamente com a página. Se ele for enviado na forma de programa-fonte, atrasos adicionais acontecerão na apresentação da página, porque o *software* será traduzido para a linguagem de máquina adequada. Entretanto, se ele for enviado em linguagem de máquina, isso implica que uma versão diferente da página deverá ser fornecida, dependendo da linguagem de máquina usada pelo computador que acessa a página.

A Sun Microsystems resolveu esse problema projetando uma “linguagem de máquina” universal chamada código de *bytes*, na qual os programas Java podem ser convertidos. Embora não seja de fato uma linguagem de máquina, ela pode ser executada rapidamente por meio de um interpretador adequado. Tais interpretadores vêm se tornando padrão nos *softwares* dos navegadores (*browsers*) atuais. Assim, se o *software* para controlar uma página da Web for escrito em Java e traduzido em código de *bytes*, então uma versão em código de *bytes* poderá ser transmitida aos navegadores, que exibirão a página Web para proporcionar uma animação eficiente.

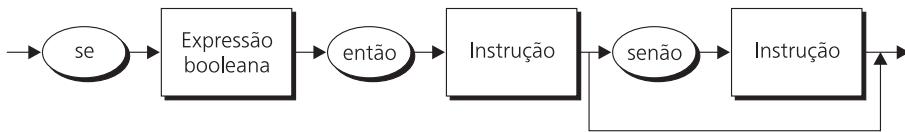


Figura 5.17 Um diagrama de sintaxe da instrução se-então-senão de nosso pseudocódigo.

O processo de análise sintática de um programa consiste, essencialmente, em construir uma árvore de sintaxe para o programa-fonte. De fato, uma árvore de sintaxe representa o entendimento do analisador a respeito da composição gramatical do programa. Por isso, as regras de sintaxe que descrevem a estrutura gramatical de um programa não devem propiciar que duas ou mais árvores de sintaxe distintas possam ser construídas para uma mesma cadeia, dado que isto levaria a ambigüidades no analisador sintático. Essa falha pode ser muito sutil. Na verdade, a própria regra da Figura 5.17 contém esse defeito, pois aceita as duas árvores de sintaxe mostradas na Figura 5.20 para a única instrução a seguir:

```
if B1 then if B2 then S1
else S2
```

Note-se que as duas interpretações são significativamente diferentes. A primeira implica que a instrução S2 será executada se *B1* for falso, enquanto a segunda implica que S2 só será executada se *B1* for verdadeiro e *B2*, falso.

As definições de sintaxe para linguagens formais de programação são projetadas para evitar tais ambigüidades. Em nosso pseudocódigo, evitamos esses problemas usando parênteses. Por exemplo, escreveríamos

```
se B1
  então (se B2 então S1)
  senão S2
```

e

```
se B1
  então (se B2 então S1)
  senão S2)
```

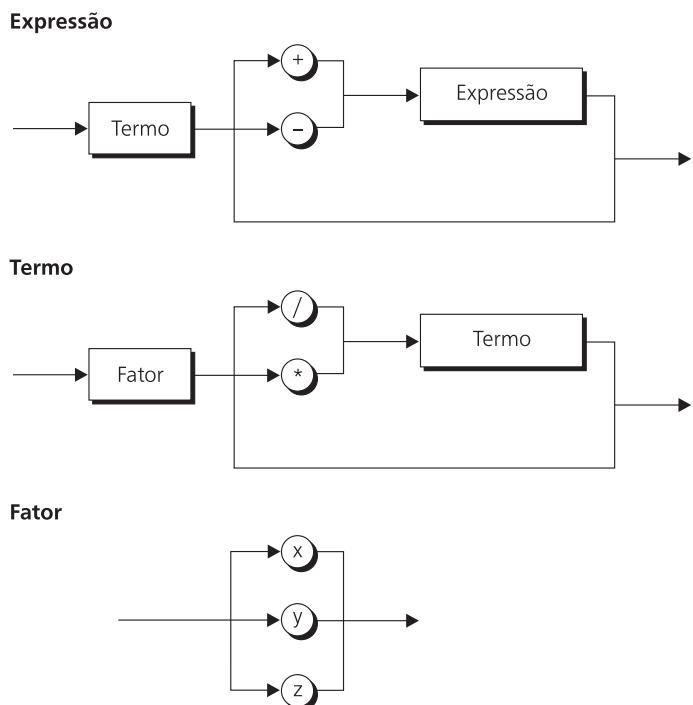


Figura 5.18 Diagramas de sintaxe que descrevem a estrutura de uma expressão algébrica simples.

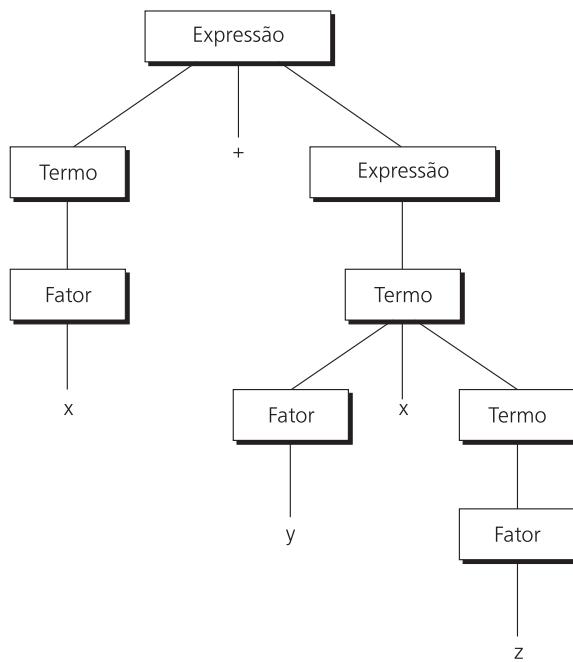


Figura 5.19 Árvore sintática da cadeia $x + y * z$ com base nos diagramas de sintaxe da Figura 5.18.

dois os dados envolvidos não são do mesmo tipo. Por exemplo, se X for inteiro e Y real, o conceito de adição ainda será aplicável. Neste caso, o analisador sintático poderá decidir que o gerador de código construa as instruções necessárias para converter um dado de um tipo para outro, antes de executar a adição. Tal conversão implícita entre tipos é denominada **coerção**.

As operações de coerção são malvistas por muitos projetistas de linguagens. Eles argumentam que a necessidade da coerção é um sintoma de falha no projeto do programa, e por isso não deve ser contornada pelo analisador sintático. O resultado é que a maioria das linguagens modernas é **fortemente tipificada**, o que significa que todas as ações solicitadas por um programa devem envolver dados de tipos compatíveis, sem coerção. Os analisadores sintáticos para estas linguagens consideram erros quaisquer incompatibilidades de tipo.

A geração de código, ação final do processo de tradução, é o processo de construção das instruções em linguagem de máquina responsáveis por implementar as instruções reconhecidas pelo analisador sintático. Este processo envolve numerosos problemas, um dos quais é a construção de um código eficiente. Por exemplo, consideremos a tarefa de traduzir a seguinte sequência de duas instruções:

$$\begin{aligned} x &\leftarrow y + z; \\ w &\leftarrow x + z; \end{aligned}$$

Elas poderiam ser traduzidas como instruções independentes. Todavia, esta interpretação tende a não produzir um código eficiente. O gerador de código deve ser construído de forma a ser capaz de identificar que, ao ser completada a primeira instrução, os valores de x e de z já se encontram em registradores de propósito geral do processador, de modo que não precisam ser carregados a partir da memória para o cálculo do valor de w . Melhorias como essa constituem a **otimização de código**, cuja realização constitui importante tarefa do gerador de código.

para distinguir as duas possíveis interpretações.

À medida que um analisador sintático analisa as instruções declarativas de um programa, ele registra essa informação em uma tabela conhecida como **tabela de símbolos**. Dessa forma, a tabela de símbolos guarda informações sobre as variáveis declaradas, os tipos de dados e as estruturas de dados associadas a tais variáveis. O analisador sintático então utiliza como base estas informações ao analisar instruções imperativas, como:

$$Z \leftarrow X + Y;$$

De fato, para determinar o significado do símbolo '+', o analisador sintático deverá saber qual o tipo de dados associado às variáveis X e Y. Se X for do tipo real e Y do tipo caractere, então somar X e Y faz pouco sentido e deverá ser considerado um erro. Se X e Y forem de tipo inteiro, então o analisador sintático solicitará ao gerador de código a construção de uma instrução em linguagem de máquina que utilize o código de operação correspondente à adição de inteiros. Se, porém, ambos forem de tipo real, o analisador solicitará o uso do código de operação correspondente à adição de valores em vírgula flutuante.

A instrução acima também faz sentido quando

os dados envolvidos não são do mesmo tipo. Por exemplo, se X for inteiro e Y real, o conceito de adição

ainda será aplicável. Neste caso, o analisador sintático poderá decidir que o gerador de código construa

as instruções necessárias para converter um dado de um tipo para outro, antes de executar a adição. Tal

conversão implícita entre tipos é denominada **coerção**.

As operações de coerção são malvistas por muitos projetistas de linguagens. Eles argumentam que

a necessidade da coerção é um sintoma de falha no projeto do programa, e por isso não deve ser contornada pelo analisador sintático. O resultado é que a maioria das linguagens modernas é **fortemente tipificada**, o que significa que todas as ações solicitadas por um programa devem envolver dados de tipos compatíveis, sem coerção. Os analisadores sintáticos para estas linguagens consideram erros quaisquer incompatibilidades de tipo.

A geração de código, ação final do processo de tradução, é o processo de construção das instruções em linguagem de máquina responsáveis por implementar as instruções reconhecidas pelo analisador sintático. Este processo envolve numerosos problemas, um dos quais é a construção de um

código eficiente. Por exemplo, consideremos a tarefa de traduzir a seguinte sequência de duas instruções:

$$x \leftarrow y + z;$$

$$w \leftarrow x + z;$$

Elas poderiam ser traduzidas como instruções independentes. Todavia, esta interpretação tende a

não produzir um código eficiente. O gerador de código deve ser construído de forma a ser capaz de

identificar que, ao ser completada a primeira instrução, os valores de x e de z já se encontram em

registradores de propósito geral do processador, de modo que não precisam ser carregados a partir da

memória para o cálculo do valor de w . Melhorias como essa constituem a **otimização de código**, cuja

realização constitui importante tarefa do gerador de código.

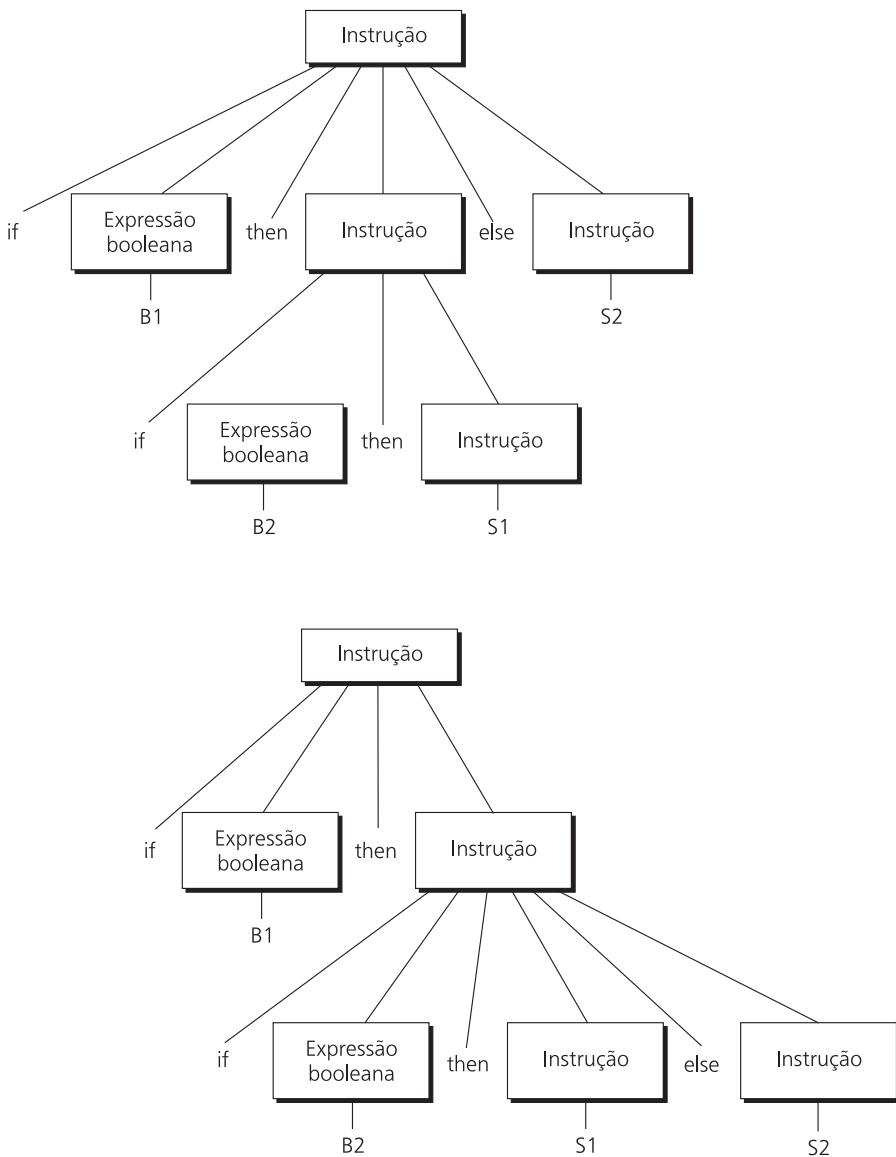


Figura 5.20 Duas árvores de sintaxe distintas para a instrução if B1 then B2 then S1 else S2.

Note-se que as análises léxica e sintática e a geração de código não são efetuadas em ordem estritamente seqüencial, mas de forma intercalada. O analisador léxico começa lendo os caracteres do programa-fonte e, ao identificar o primeiro símbolo, passa-o para o analisador sintático. Cada vez que este último recebe um símbolo do analisador léxico, ele analisa a estrutura gramatical que está sendo lida. Nesse ponto, ele pode solicitar um novo símbolo ao analisador léxico ou, se reconhecer que uma frase ou instrução completa foi lida, chamar o gerador de código para produzir as instruções de máquina apropriadas. Cada chamada faz com que o gerador de código construa instruções de máquina que são acrescentadas ao programa-objeto. Então, a tarefa de traduzir um programa de uma linguagem para

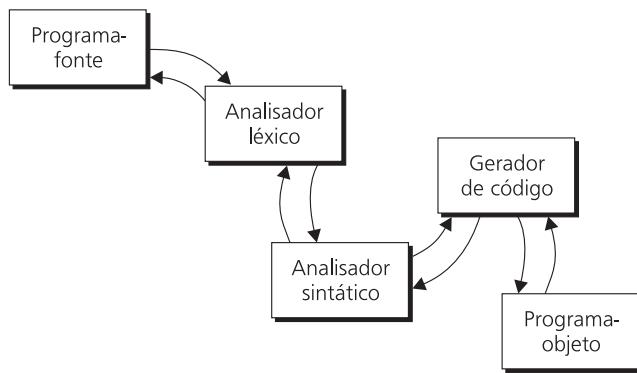


Figura 5.21 Uma abordagem orientada a objeto do processo de tradução.

para a forma de módulos individuais, separadamente, e em épocas diferentes, as várias partes de um programa (isso apóia a prática da construção modular de *software*). Assim, com freqüência, o programa-objeto produzido a partir de um único processo de tradução constitui apenas uma das várias peças de um programa completo, em que cada uma requisita serviços às demais para realizar as tarefas do sistema como um todo. Mesmo quando um programa completo é desenvolvido e traduzido como um único módulo, o programa-objeto correspondente raramente se encontra preparado para operar de forma autônoma no momento da execução, uma vez que provavelmente contém solicitações ao sistema operacional e aos softwares utilitários de serviços disponibilizados por intermédio do sistema operacional. Assim, um programa-objeto é na verdade um programa em linguagem de máquina que contém algumas “pontas soltas”, as quais devem ser devidamente conectadas a outros programas-objeto, antes de se conseguir um programa executável.

A tarefa de construir tais conexões é executada por um programa chamado **ligador**^{*}. Seu trabalho é unir alguns programas-objeto (resultantes de traduções prévias e independentes), rotinas do sistema operacional e outros softwares utilitários para produzir um programa completo, executável (conhecido como **módulo de carga**^{**}), o qual, por sua vez, é armazenado na forma de um arquivo, no sistema de armazenamento em massa do computador.

Finalmente, para que um programa já traduzido possa ser executado, o módulo de carga precisa ser transferido para a memória, o que é efetuado por um programa chamado **carregador**^{***}, o qual geralmente constitui uma parte do escalador do sistema operacional (Seção 3.3). A importância desta atividade é mais pronunciada no caso de sistemas multitarefa. Neles, uma vez que um programa deve compartilhar a memória com outros processos em execução, e essa mistura de processos varia a cada execução, a área exata de memória disponível ao programa não é conhecida até o momento da sua execução. Nesse contexto, a tarefa do carregador é simplesmente depositar o programa na área de memória indicada pelo sistema operacional e efetuar alguns ajustes de última hora (último microsegundo) que se fizerem necessários, ao se tornar conhecida a posição exata de memória em que o programa será executado. (Uma instrução de desvio no programa precisa desviar para o endereço correto dentro do programa.) No intuito de minimizar tais ajustes realizados pelo carregador, encorajou-se o desenvolvimento de técnicas para evitar que um programa efetue referências explícitas a

outra se encaixa naturalmente no paradigma orientado a objeto. O programa-fonte, os analisadores léxico e sintático, o gerador de código e o programa-objeto são objetos que interagem enviando mensagens à medida que cada uma realiza a sua tarefa (Figura 5.21).

Ligação e carregamento

Embora expresso em linguagem de máquina, o programa-objeto produzido pelo processo de tradução raramente se encontra em uma forma que possa ser diretamente executada pela máquina. Uma razão é que, na maioria dos ambientes de programação, é permitido desenvolver e traduzir

*N. de T. Em inglês, *linker*.

**N. de T. Às vezes conhecido como *módulo carregável*, ou *módulo executável*. Em inglês, *load module*.

***N. de T. Em inglês, *loader*.

endereços físicos de memória. Com isto, conseguiu-se um tipo de programa (chamado **módulo relocável**) que é capaz de ser corretamente executado sem modificações e de forma independente de sua posição física na memória.

Em resumo, a tarefa completa de preparar um programa escrito em linguagem de alto nível para a execução compreende uma sequência de três passos: tradução, ligação e carregamento, conforme descrito na Figura 5.22. Uma vez realizados os passos de tradução e ligação, o programa pode ser repetidamente carregado e executado, sem a necessidade de retornar à versão original. No entanto, se for preciso executar alguma alteração no programa, esta deve ser feita no programa-fonte, o qual, por sua vez, é novamente traduzido e ligado, resultando em um novo módulo de carga que incorpora as alterações efetuadas.

Pacotes para o desenvolvimento de software

A tendência atual é a de reunir em um pacote, que funcione como um sistema de *software* integrado, o tradutor e os demais programas utilizados no processo de desenvolvimento de *software*. No esquema de classificação da Seção 3.2, tal sistema seria classificado como um *software* de aplicação. Ao usar esse pacote de aplicação, um programador tem acesso direto a um editor, para escrever programas, a um tradutor, para converter os programas para linguagem de máquina, e a várias ferramentas de depuração que lhe permitem rastrear a execução de um programa que não esteja funcionando adequadamente, para localizar o ponto exato em que a perda do controle ocorreu.

As vantagens do uso de um sistema integrado são numerosas. Talvez a mais óvia seja a de que um programador pode usar um editor e as ferramentas de depuração, alternando com facilidade entre um e outro, quando são realizadas e testadas alterações no programa. Além disso, muitos pacotes de desenvolvimento de *software* permitem que módulos de programa que se inter-relacionem de alguma forma possam se conectar, ainda na fase de desenvolvimento, de forma simplificada. Alguns pacotes mantêm registros para mostrar, dentre um conjunto de módulos relacionados, aqueles que foram alterados desde o último teste de avaliação. Esses recursos são vantajosos no desenvolvimento de grandes sistemas de *software*, nos quais muitos módulos inter-relacionados são desenvolvidos por diferentes programadores.

Em uma escala menor, os editores encontrados em pacotes de desenvolvimento de *software* frequentemente são personalizados para a linguagem de programação em uso. Por exemplo, um editor em um pacote de desenvolvimento de *software* efetua automaticamente a operação de endentação das linhas, que é utilizada de forma padronizada na programação da linguagem desejada e, em alguns casos, pode reconhecer palavras-chave e completar automaticamente, após o programador ter digitado apenas alguns de seus caracteres iniciais.

Muitos pacotes de desenvolvimento de *software* usam interfaces gráficas que permitem a construção de programas a partir de blocos predesenvolvidos, representados por ícones na tela. Os blocos selecionados podem então ser personalizados por meio de um editor. Tais pacotes refletem a tendência geral de construir *software* a partir de grandes blocos pré-fabricados, em vez de escrevê-los instrução a instrução.

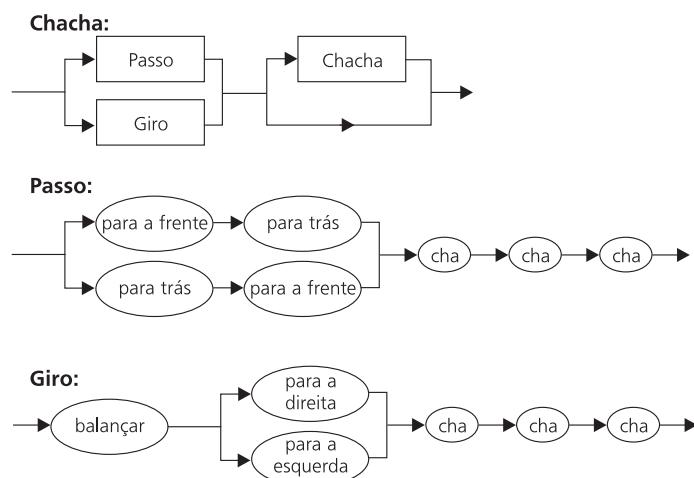


Figura 5.22 O processo completo de preparação de um programa.



QUESTÕES/EXERCÍCIOS

1. Descreva os três principais passos do processo de tradução.
2. O que é uma tabela de símbolos?
3. Construa uma árvore de sintaxe para a expressão
 $x \times y + x + z$
 com base nos diagramas de sintaxe da Figura 5.18.
4. Descreva as cadeias que obedecem à estrutura Chacha, definida de acordo com os seguintes diagramas de sintaxe.



5.5 Programação orientada a objeto

Vimos na Seção 5.1 que o paradigma da orientação a objeto leva ao desenvolvimento de unidades de programa ativas, chamadas objetos, em que cada uma contém procedimentos que descrevem como esses objetos devem responder a vários estímulos. A abordagem orientada a objeto a um problema é identificar os objetos envolvidos e descrevê-los como unidades autônomas. Por sua vez, as linguagens de programação orientadas a objeto fornecem instruções para expressar essas idéias. Nesta seção, introduzimos algumas dessas instruções como aparecem nas linguagens C++, Java e C#, que atualmente são as três linguagens orientadas a objeto mais proeminentes.

Classes e objetos

Considere a tarefa de desenvolver um jogo de computador no qual o jogador deve proteger a Terra de meteoros, atirando com raios *laser* de alta potência. Cada *laser* contém uma fonte de energia interna finita que é parcialmente consumida a cada disparo. Uma vez gasto, o *laser* se torna inútil. Cada *laser* deve responder aos comandos de apontar para a direita, apontar para a esquerda e de disparar o facho.

No paradigma orientado a objeto, cada *laser* no jogo de computador seria implementado como um objeto que contém um registro de sua energia disponível, bem como procedimentos para modificar a sua pontaria e disparar o facho de *laser*. Uma vez que todos os objetos *laser* possuem as mesmas propriedades, eles podem ser construídos a partir do mesmo modelo. No paradigma orientado a objeto, esse modelo é chamado **classe**. Nas linguagens C++, Java e C#, uma classe é descrita por uma declaração da forma

```
class Nome
{
    .
    .
    .
}
```

onde *Nome* identifica a classe em todo o programa. As propriedades da classe são descritas entre chaves.

Uma classe chamada *ClasseLaser*, que descreve a estrutura de um *laser* em nosso jogo de computador, é esboçada na Figura 5.23. (Uma descrição mais completa da classe será dada mais tarde, na Figura 5.25.) Qualquer objeto construído a partir desse modelo conterá uma variável chamada *EnergiaRestante*, do tipo inteiro, e três procedimentos chamados *virarDireita*, *virarEsquerda* e *disparar*, que descrevem os passos a serem efetuados para realizar a ação correspondente. Uma variável que reside em um objeto, tal como a *EnergiaRestante*, é chamada **variável de instância**, e os procedimentos do objeto são chamados **métodos** (ou funções-membro, no vernáculo do C++). Note que na Figura 5.23 a variável de instância *EnergiaRestante* é descrita usando-se uma instrução de declaração similar àquelas discutidas na Seção 5.2, e os métodos são descritos em uma forma reminiscente dos procedimentos e funções, como discutido na Seção 5.3. De fato, as declarações de variáveis de instância e as descrições de métodos são basicamente conceitos da programação imperativa.

Uma vez descrita a classe *ClasseLaser* do nosso programa de jogo, podemos declarar três variáveis *Laser1*, *Laser2* e *Laser3* como o “tipo” *ClasseLaser* com uma instrução da forma

```
ClasseLaser Laser1, Laser2, Laser3;
```

Note que é a mesma forma da instrução

```
int x, y, z;
```

que seria usada para criar três variáveis chamadas *x*, *y* e *z* do tipo inteiro, como já vimos na Seção 5.2. Ambas consistem no nome do “tipo” seguido da lista de variáveis a serem declaradas. A diferença é que qualquer entidade atribuída a *x* será construída de acordo com o modelo predefinido *Inteiro* (um tipo primitivo), enquanto qualquer entidade atribuída a *Laser1* será construída de acordo com o modelo definido pela classe dentro do programa.

Na linguagem C++, a instrução de declaração precedente cria três objetos do “tipo” *ClasseLaser* e os atribui às variáveis *Laser1*, *Laser2* e *Laser3*. Entretanto, em Java e em C#, a instrução meramente cria as variáveis, às quais poderão ser atribuídos valores. Para efetivamente criar um objeto do “tipo” *ClasseLaser* e atribuí-lo a *Laser1*, usariamos a instrução

```
ClasseLaser Laser1 = new ClasseLaser();
```

que não apenas declara que a variável é do tipo *ClasseLaser*, mas também cria um novo objeto usando o modelo *ClasseLaser* e o atribui a *Laser1*. Aqui, mais uma vez, a forma da instrução é similar à usada para declarar variáveis de tipos predefinidos. De fato, a instrução

CORBA e RMI

A conveniência de se implementar sistemas de *software* para redes de acordo com o paradigma orientado a objeto tem levado ao desenvolvimento de infra-estruturas de rede, por meio das quais os objetos podem trocar mensagens através da rede de uma maneira conveniente. Uma abordagem é o CORBA (Common Object Request Broker Architecture), que é um sistema aberto de especificações para implementar a troca de mensagens entre objetos de uma rede. Outra é o RMI (Remote Method Invocation), que é um sistema de especificações para tais comunicações, desenvolvido pela Sun Microsystems e suportado no ambiente de programação Java. (Aprenderemos a linguagem de programação Java mais tarde, neste capítulo.) Os desenvolvedores de *software* que usam o ambiente de programação Java têm fácil acesso ao RMI, portanto, ele é o método escolhido quando os objetos a serem desenvolvidos se comunicarão com outros “objetos Java”. Contudo, o CORBA permanece como padrão usado para objetos desenvolvidos em outras linguagens de programação.

```

class ClasseLaser
{
    int EnergiaRestante = 100;           Descrição dos dados que
                                         residirão em cada objeto
                                         deste "tipo".
    void virarDireita ()               Métodos que descrevem como
    { ... }                            um objeto deste "tipo" deve
                                         responder a várias mensagens.
    void virarEsquerda ()             {
    { ... }                            }
    void disparar ()                  {
    { ... }                            }
}

```

Figura 5.23 A estrutura de uma classe que descreve uma arma a laser em um jogo de computador.

execute seu método `virarEsquerda` com a instrução

```
Laser2.virarEsquerda();
```

Antes de continuar, devemos fazer uma pausa para enfatizar a distinção entre uma classe e um objeto. Uma classe é um modelo a partir do qual os objetos são construídos. Uma classe pode ser usada para criar numerosos objetos. Frequentemente nos referimos a um objeto como uma **instância** de uma classe a partir da qual ele foi construído. Assim, em nosso jogo de computador, `Laser1`, `Laser2` e `Laser3` são usadas para nos referirmos às instâncias da `ClasseLaser`.

```

class ClasseLaser
{
    int EnergiaRestante;
    Um construtor atribui um valor
    a EnergiaRestante quando o
    objeto é criado.

    ClasseLaser (EnergiaInicial)
    { EnergiaRestante = EnergiaInicial;
    }

    void virarDireita ()
    { ... }

    void virarEsquerda ()
    { ... }

    void disparar ()
    { ... }
}

```

Figura 5.24 Uma classe com um construtor.

```
int x = 3;
```

não apenas declara que a variável `x` é do tipo inteiro, mas também atribui à nova variável o valor 3.

Programando em C++, Java ou C#, após criar os objetos e atribuir-lhes os nomes `Laser1`, `Laser2` e `Laser3`, poderíamos continuar o nosso programa de jogo ativando os métodos apropriados nesses objetos (no vernáculo orientado a objeto, isso é chamado “enviando mensagens aos objetos”), como exigido. Podemos fazer com que `Laser1` execute seu método `disparar` usando a instrução

```
Laser1.disparar();
```

ou fazer com que o objeto `Laser2`

Construtores

Quando um objeto é construído, geralmente algumas atividades de personalização devem ser realizadas. Por exemplo, no nosso jogo de computador dos meteoros, podemos desejar que os diferentes *lasers* tenham diferentes reservas iniciais de energia, o que significa que as variáveis de instância chamadas `EnergiaRestante` dentro dos vários objetos devem ser criadas com diferentes valores iniciais. Essas necessidades de inicialização são supridas definindo-se métodos especiais chamados **construtores**, que são executados automaticamente quando o objeto é construído a partir da classe. Um construtor é identificado na definição da classe pelo fato de possuir o mesmo nome da classe.

A Figura 5.24 apresenta uma extensão da definição da `ClasseLaser` mostrada na Figura 5.23. Note que ela contém um construtor na forma de um método chamado `ClasseLaser`, o qual atribui à variável de ins-

tância EnergiaRestante o valor recebido como parâmetro. Assim, quando um objeto é construído a partir dessa classe, este método será executado, causando a inicialização adequada de EnergiaRestante.

Os parâmetros reais a serem usados pelo construtor normalmente são identificados com a lista de parâmetros na instrução causadora da criação do objeto. Assim, baseado na definição de classe da Figura 5.24, um programador C++ escreveria

```
ClasseLaser Laser1(50), Laser2(100);
```

para criar dois objetos do tipo ClasseLaser — um conhecido como Laser1, que possuiria reserva inicial de energia igual a 50; o outro, conhecido como Laser2, com reserva inicial de energia igual a 100. Os programadores Java e C# fariam a mesma coisa com as instruções

```
ClasseLaser Laser1 = new ClasseLaser(50);
ClasseLaser Laser2 = new ClasseLaser(100);
```

Recursos adicionais

Suponhamos agora que desejemos melhorar o nosso jogo de computador dos meteoros de tal forma que, ao alcançar um certo escore, um jogador seja premiado com um recarregamento de alguns dos lasers com sua energia inicial. Esses lasers teriam as mesmas propriedades dos outros, exceto em que seriam recarregáveis.

Para simplificar a descrição de objetos com características similares, embora diferentes, a maioria das linguagens orientadas a objeto permite que uma classe englobe as propriedades de outra por meio de um sistema conhecido como **herança**. Como exemplo, suponha que estejamos usando o Java para desenvolver o nosso programa de jogo. Poderíamos usar primeiramente a instrução mostrada antes para definir uma classe chamada ClasseLaser, que descreveria as propriedades comuns a todos os lasers do programa. Então, poderíamos usar a instrução

```
class LaserRecarregavel extends ClasseLaser
{
    .
    .
    .
}
```

para descrever outra classe chamada LaserRecarregavel. (Os programadores C++ e C# meramente substituiriam a palavra extends por dois pontos.) Aqui, a cláusula extends indica que essa classe deve herdar as características da ClasseLaser e também conter as características que aparecem entre chaves. Em nosso caso, as chaves conteriam um novo método (talvez chamado recarga) que descreveria os passos necessários para reajustar a variável de instância EnergiaRestante com o seu valor inicial. Uma vez definida essa classe, poderíamos usar a instrução

```
ClasseLaser Laser1, Laser2;
```

para declarar Laser1 e Laser2 como variáveis que se referem aos lasers tradicionais e usar a instrução

```
LaserRecarregavel Laser3, Laser4;
```

para declarar Laser3 e Laser4 como variáveis que se referem a lasers que possuem propriedades adicionais descritas na classe LaserRecarregavel.

O uso da herança leva à existência de diversos objetos com características similares, ainda que diferentes, que por sua vez levam a um fenômeno reminiscente da sobrecarga, que encontramos na Seção 5.2. (Lembre-se de que a sobrecarga se refere ao uso de um único símbolo, como o +, para representar diferentes operações, dependendo do tipo de seus operandos.) Suponha que um pacote

C# e .NET framework

Para simplificar o processo de desenvolvimento de *software* que atenda às necessidades atuais (como o desenvolvimento de sítios Web que se comunicam entre si para servir a seus clientes), a Microsoft desenvolveu um ambiente de *software* chamado *.NET framework*. Dentre outras coisas, esse *framework* contém uma coleção (chamada Biblioteca de Classes) de componentes que podem ser usados como ferramentas abstratas quando se desenvolve *software* de aplicação. Embora grande parte dessa biblioteca possa ser acessada por sistemas de desenvolvimento de *software* já fornecidos pela Microsoft (como o Visual Basic), a Microsoft projetou uma nova linguagem de programação, chamada C#, que foi produzida visando esse *framework*. Enquanto linguagem, o C# é um primo próximo do C++. A novidade do C# é que ele promete ser a principal linguagem para o desenvolvimento de *software* que utilize a *.NET framework*. Embora o C# e a *.NET framework* sejam propostos para ser usados apenas nas máquinas com sistemas operacionais da Microsoft, a popularidade desses sistemas implica que o C# será uma linguagem de programação popular em um futuro próximo.

Os componentes da classe são designados públicos ou privados, conforme forem acessíveis a outras unidades de programa ou não.

gráfico orientado a objeto consista em diversos objetos, e cada um represente uma forma (circunferência, retângulo, triângulo etc.) Uma específica imagem pode consistir em uma coleção desses objetos. Cada objeto sabe o seu tamanho, localização e cor, bem como as respostas à mensagens que solicitam, por exemplo, que ele se move para algum lugar, ou que seja exibido na tela do monitor. Para desenhar uma imagem, simplesmente enviamos uma mensagem “autodesenhe” a cada objeto da mesma. Contudo, a rotina usada para desenhar um objeto varia de acordo com a sua forma — desenhar um quadrado não é o mesmo processo que desenhar uma circunferência. Essa interpretação personalizada de uma mensagem é conhecida como **polimorfismo**; diz-se que a mensagem é polimórfica.

Outra característica associada com a programação orientada a objeto é o **encapsulamento**, que se refere ao acesso restrito às propriedades internas de um objeto. Dizer que certos recursos de um objeto são *encapsulados* significa que apenas o objeto pode ter acesso a eles. Diz-se que os recursos encapsulados são privados e os acessíveis de fora do objeto, públicos.

Como exemplo, retornemos à nossa ClasseLaser originalmente esboçada na Figura 5.23. Lembre-se de que ela descreve uma variável de instância EnergiaRestante e três métodos: virarDireita, virarEsquerda e disparar. Esses métodos devem ser chamados por

```
classe ClasseLaser
{private int EnergiaRestante;
public ClasseLaser (EnergiaInicial)
{EnergiaRestante = EnergiaInicial;
}
public void virarDireita ( )
{ ... }
public void virarEsquerda ( )
{ ... }
public void disparar ( )
{ ... }
}
```

Figura 5.25 Nossa definição de ClasseLaser utiliza o encapsulamento como se fosse aparecer em um programa Java ou C#.

outras unidades de programa para fazer com que uma instância de `ClasseLaser` realize as ações apropriadas. Entretanto, o valor de `EnergiaRestante` deve ser alterado apenas pelos métodos internos da instância. Nenhuma outra unidade de programa deverá ter acesso a esse valor diretamente. Para fazer cumprir essas regras, só precisamos designar `EnergiaRestante` como privada e `virarDireita`, `virarEsquerda` e `disparar` como públicos, como mostrado na Figura 5.25.

Como outro exemplo, considere um programa projetado para simular a interação de várias empresas. Ele pode envolver objetos construídos a partir de uma classe chamada `ClassePedidosAEmpresa`. Esses objetos provavelmente conterão itens de dados relacionados à atividade financeira das empresas, bem como um método que descreva como o objeto deve responder a um pedido. Os outros objetos precisarão ativar esse método quando quiserem fazer um pedido, mas não devem ter acesso aos registros financeiros. Assim, o acesso aos registros financeiros deve ser privado, enquanto o método para fazer um pedido deve ser público.



QUESTÕES/EXERCÍCIOS

1. Qual é a diferença entre um objeto e uma classe?
2. Que outras classes de objetos além da `ClasseLaser` podem ser encontradas no exemplo do jogo de computador usado nesta seção? Que variáveis de instância além de `EnergiaRestante` podem ser encontradas na `ClasseLaser`?
3. Suponha que as classes `FuncionarioTurnoParcial` e `FuncionarioTurnoIntegral` herdem as propriedades da classe `Funcionario`. Quais são as características que você espera encontrar em cada classe?
4. O que é construtor?

5.6 Programação de atividades concorrentes

Suponha que devamos projetar um programa para produzir animação para um jogo de computador que envolva múltiplas naves invasoras inimigas. Uma abordagem seria projetar um único programa que controlaria a animação na tela inteira. Tal programa seria penalizado com o desenho de cada nave, o que (se é para a animação parecer realista) significa que o programa teria de se manter a par das características individuais de numerosas naves. Uma abordagem alternativa seria projetar um programa para controlar a animação de uma única nave cujas características sejam determinadas por parâmetros atribuídos no início da execução do programa. Então, a animação poderá ser construída criando múltiplas ativações desse programa, cada uma com o seu conjunto de parâmetros. Com a execução simultânea dessas ativações, poderíamos obter a ilusão de muitas naves individuais percorrendo a tela ao mesmo tempo.

Essa execução simultânea de múltiplas ativações é chamada **processamento paralelo** ou **processamento concorrente**. O processamento verdadeiramente paralelo exige múltiplas UCPs, cada uma para executar uma ativação. Quando apenas uma UCP está disponível, a ilusão do processamento paralelo é obtida permitindo-se que as ativações compartilhem o tempo do processador, como discutido no Capítulo 3.

As linguagens de programação para a escrita de programas que envolvem o processamento paralelo foram originalmente projetadas para uso no desenvolvimento de sistemas operacionais. Contudo, como nossos exemplos introdutórios mostram, muitas aplicações modernas de computador são mais facilmente implementadas no contexto de processamento paralelo do que no contexto mais tradicional, que envolve uma única sequência de instruções. Assim, as linguagens de programação mais novas fornecem a sintaxe para expressar as estruturas semânticas envolvidas nas computações paralelas. O projeto

de tais linguagens exige a identificação dessas estruturas semânticas e o desenvolvimento de uma sintaxe para representá-las.

Cada linguagem de programação tende a abordar o paradigma do processamento paralelo a partir de seu ponto de vista, o que resulta em diferentes terminologias. Por exemplo, o que informalmente denominamos ativação é chamado tarefa no vernáculo da Ada e linha (*thread*) em Java. Isto é, em um programa Ada, as ações simultâneas são realizadas com a criação de múltiplas *tarefas*, enquanto em Java criamos múltiplas *linhas*. Em ambos os casos, o resultado é que múltiplas atividades são geradas e executadas de uma maneira muito próxima à dos processos sob controle de um sistema operacional multitarefa. Assim, adotamos a política de nos referirmos a ativação, tarefa e linha como um processo.

Talvez a ação mais básica que tenha de ser expressa em um programa que envolva processamento paralelo seja a da criação de novos processos. Se queremos ter ativações múltiplas do programa das naves que executem ao mesmo tempo, precisamos de uma sintaxe para especificar isso. Essa criação de novos processos geralmente é feita de uma maneira similar à da solicitação de execução de um procedimento tradicional. A diferença é que, no ambiente tradicional, a unidade de programa que solicita a ativação de um procedimento não progride até que o procedimento solicitado se encerre, enquanto no contexto paralelo, a unidade de programa solicitante continua a execução de suas instruções enquanto o procedimento solicitado realiza a sua tarefa. Assim, para criar múltiplas naves que percorram a tela, escreveríamos um programa principal, que simplesmente geraria múltiplas ativações do programa da nave, cada qual com os seus parâmetros descritivos das características distintas da nave.

Um tópico mais complexo, associado com o processamento paralelo, envolve o tratamento da comunicação entre os processos. No nosso exemplo das naves, os processos que representam naves diferentes precisam comunicar as suas localizações aos outros, a fim de coordenar as suas atividades. Em outros casos, um processo pode precisar esperar até que outro atinja um certo ponto em sua computação, ou um processo pode precisar interromper outro até que o primeiro realize uma tarefa específica.

Essas necessidades de comunicação são há muito tempo um tópico de estudo para os cientistas da computação, e muitas das linguagens de programação mais novas refletem as várias abordagens ao problema da comunicação entre processos. Como um exemplo, consideremos os problemas de comunicação encontrados quando dois processos manipulam os mesmos dados. (Esse exemplo é apresentado em mais detalhe na Seção opcional 3.4.) Se cada um dos processos que estão executando concorrentemente necessita adicionar o valor três a um item de dados comum, é necessária uma técnica para garantir que um processo possa completar a sua transação antes que o outro realize a sua tarefa. De outra forma, ambos poderiam iniciar a sua computação individual com o mesmo valor inicial, e o resultado seria incrementado apenas em três, em vez de seis. Os dados que podem ser acessíveis a apenas um processo por vez são denominados de acesso mutuamente exclusivo.

Uma abordagem para resolver esse problema é escrever as unidades de programa que descrevem os processos envolvidos de tal forma que, quando um processo estiver usando um dado compartilhado, ele bloqueie o acesso de outros processos aos dados, até que esse acesso esteja seguro. (Essa é a abordagem descrita na Seção opcional 3.4, onde identificamos a parte de um processo que tem acesso a dados compartilhados como a região crítica.) A experiência tem mostrado que essa abordagem tem como desvantagem a distribuição da tarefa de garantir a exclusão mútua em várias partes do programa — cada unidade de programa com acesso a dados compartilhados deve ser corretamente projetada para garantir a exclusão mútua e, assim, um equívoco em um simples segmento pode corromper o sistema inteiro. Por essa razão, muitos argumentam que uma solução melhor é incorporar ao item de dados o controle do acesso. Resumindo, em vez de delegar aos processos que compartilham dados a guarda contra acessos múltiplos, essa responsabilidade é atribuída ao próprio item de dados. O resultado é que o controle do acesso fica concentrado em um único ponto do programa, em vez de se dispersar em várias unidades de programa¹. Um item de dados com habilidade de controlar o acesso a si próprio freqüentemente é chamado **monitor**.

¹Este problema possui uma solução natural nas linguagens de programação orientadas a objeto, onde os processos que têm acesso aos dados são reunidos no mesmo objeto que os dados. Mais especificamente, você pode desejar explorar o uso da instrução *synchronize* em Java.

Vimos então que o projeto das linguagens de programação para processamento paralelo envolve o desenvolvimento de meios para expressar coisas como a criação de processos, as pausas e retomadas de processos, a identificação de regiões críticas e a composição de monitores.

Por fim, devemos notar que embora a animação proveja um ambiente interessante no qual podemos explorar tópicos da computação paralela, ela é apenas uma de muitas áreas que se beneficiam das técnicas de processamento paralelo. Outras áreas incluem a previsão meteorológica, o controle de tráfego aéreo, a simulação de sistemas complexos (desde reações nucleares até tráfego de pedestres), as redes de computadores e a manutenção de bancos de dados.



QUESTÕES/EXERCÍCIOS

1. Cite algumas propriedades que devem ser encontradas em uma linguagem de programação para processamento concorrente e que não constam em uma linguagem mais tradicional.
2. Descreva dois métodos para garantir o acesso mutuamente exclusivo aos dados.

5.7 Programação declarativa

Argumentamos previamente que a lógica formal disponibiliza um algoritmo geral de resolução de problemas, com o qual um sistema de programação declarativa pode ser construído. Nesta seção, estaremos esta afirmação, introduzindo em primeiro lugar os fundamentos de tal algoritmo e, em seguida, analisaremos brevemente uma linguagem declarativa de programação baseada nele.

Dedução lógica

Suponha que saibamos que ou Caco está no palco ou está doente, e sejamos informados de que ele não está no palco. Concluímos então que Caco está doente. Isto é um exemplo de um princípio de raciocínio dedutivo, chamado **resolução**.

Para entender melhor este princípio, convencionemos primeiro representar afirmações simples por letras isoladas e a negação de uma proposição pelo símbolo \neg . Por exemplo, poderíamos representar a proposição “Caco é um príncipe” por A e “Miss Piggy é uma atriz” por B . Então, a expressão

$A \text{ OR } B$

significará “Caco é um príncipe ou Miss Piggy é uma atriz”, e

$B \text{ AND } \neg A$

significará “Miss Piggy é uma atriz e Caco não é um príncipe”. Usaremos uma seta para indicar “implica”. Por exemplo, a expressão

$A \rightarrow B$

significa “Se Caco é um príncipe, então Miss Piggy é uma atriz”.

Em sua forma geral, o princípio de resolução estabelece que, dadas duas proposições, da forma

$P \text{ OR } Q$

e

$R \text{ OR } \neg Q$

podemos concluir a proposição

$P \text{ OR } R$

Neste caso, as duas proposições originais formam uma terceira, que chamamos de **resolvente**. É importante observar que o resolvente é uma consequência lógica das afirmações originais. Isto é, se as proposições originais forem verdadeiras, o resolvente também deverá ser verdadeiro. (Se Q for verdadeiro, então R deverá ser, mas se Q for falso, então P deverá ser verdadeiro. Assim, independentemente da veracidade de Q , ou P ou R deverá ser verdadeiro.)

Representamos graficamente a resolução de duas proposições na Figura 5.26, na qual descrevemos as proposições originais a partir das quais são traçadas, para baixo, linhas em direção a seus resolventes. Note-se que as resoluções só podem ser aplicadas a pares de proposições que aparecem na **forma de cláusulas** — isto é, proposições cujos elementos básicos são conectados pela operação booleana OR. Assim,

$$P \text{ OR } Q$$

está em forma de cláusula, mas

$$P \rightarrow Q$$

não. O fato de este problema potencial não causar qualquer preocupação séria é uma consequência do teorema da lógica matemática, resultante do fato de que qualquer proposição, expressa em lógica de predicados de primeira ordem (um sistema extensivamente expressivo, usado para a representação de proposições), pode ser reescrita na forma de cláusulas. Não analisaremos este teorema importante, mas para futuras referências observemos que a proposição

$$P \rightarrow Q$$

é equivalente à cláusula

$$Q \text{ OR } \neg P$$

Um conjunto de proposições será considerado **inconsistente** se for impossível que todas sejam verdadeiras ao mesmo tempo. Em outras palavras, um conjunto inconsistente de proposições é um conjunto de proposições contraditórias. Um exemplo simples seria uma proposição da forma P combinada com outra, da forma $\neg P$. Os lógicos demonstraram que uma resolução repetida constitui um método sistemático para confirmar a inconsistência de um conjunto de cláusulas contraditórias. A regra é que se a aplicação de resolução repetida produz uma cláusula vazia (resultado obtido ao solucionar uma cláusula da forma P com uma da forma $\neg P$), então o conjunto original de proposições deve ser inconsistente. Como ilustração, a Figura 5.27 mostra que o conjunto de proposições

$$\begin{aligned} &P \text{ OR } Q \\ &R \text{ OR } \neg Q \\ &\neg R \\ &\neg P \end{aligned}$$

é inconsistente.

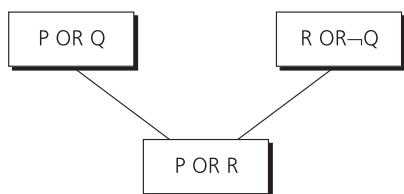


Figura 5.26 Resolução das proposições ($P \text{ OR } Q$) e ($R \text{ OR } \neg Q$) que resulta em ($P \text{ OR } R$).

Suponhamos agora que queiramos confirmar que um conjunto de proposições implica uma outra proposição, a qual denotaremos por P . Implicar a proposição P é o mesmo que contradizer a proposição $\neg P$. Assim, para demonstrar que o conjunto original de proposições implica P , tudo o que precisamos fazer é expressar as proposições originais e a proposição $\neg P$ no formato de cláusulas, e em seguida aplicar a resolução até o aparecimento de uma cláusula vazia. Isto feito, podemos concluir que a proposição $\neg P$ é inconsistente com as proposições originais, e assim estas devem implicar P .

Resta ainda um ponto a ser analisado antes de estarmos prontos para aplicar a resolução em ambientes reais de programação. Suponhamos que haja duas proposições

$$(O\ carneirinho\ de\ Mary\ está\ em\ X) \rightarrow (O\ carneirinho\ de\ Mary\ está\ em\ X)$$

onde X representa um local qualquer, e

Mary está em casa

Estas duas proposições, na forma de cláusulas, podem ser expressas como:

$$(O\ carneirinho\ de\ Mary\ está\ em\ X)\ OR\ \neg(O\ carneirinho\ de\ Mary\ está\ em\ X)$$

e

(Mary está em casa)

que, à primeira vista, não apresentam elementos a serem resolvidos. Por outro lado, os elementos (Mary está em casa) e \neg (Mary está em X) são muito parecidos para serem contraditórios. O problema está em reconhecer que, como X representa localidades genéricas, também é uma proposição acerca de uma *casa*, em particular. Assim, um caso particular da primeira proposição é

$$(O\ carneirinho\ de\ Mary\ está\ em\ casa)\ OR\ \neg(O\ carneirinho\ de\ Mary\ está\ em\ casa)$$

que pode ser resolvido com a proposição

(Mary está em casa)

para produzir a proposição

(O carneirinho de Mary está em casa)

O processo de atribuição de valores a variáveis (tal como associar o valor *casa* a X), de forma que a resolução possa ser executada, é chamado **unificação**. É este processo que permite que, em um sistema de dedução, proposições gerais sejam utilizadas em aplicações específicas.

Prolog

A linguagem Prolog (abreviação de PROGramming in LOGic*) é uma linguagem de programação declarativa cujo algoritmo subjacente de solução de problemas se baseia em resoluções repetitivas. Em Prolog, um programa é formado de um conjunto de proposições iniciais, nas quais o algoritmo subjacente baseia seu raciocínio dedutivo. Os elementos dos quais estas proposições são constituídas são chamados *predicados*. Um predicado consiste em um identificador do predicado, seguido de uma lista dos seus argumentos, entre parênteses. Um único predicado representa um fato acerca dos seus argumentos, e seu identificador em geral é escolhido de forma que lembre a semântica subjacente. Portanto, se quisermos expressar o fato de Bill ser o pai de Mary, podemos utilizar a forma de predicado

pai (bill, mary).

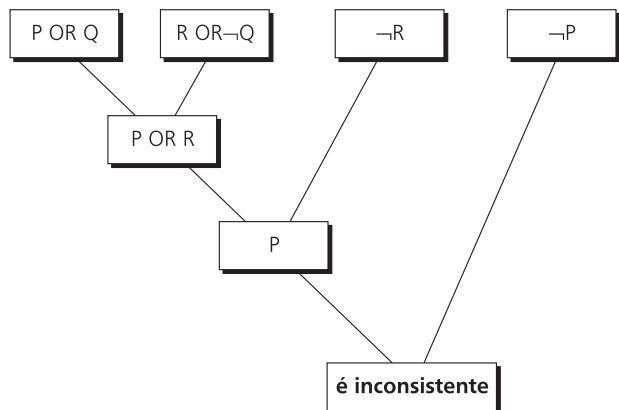


Figura 5.27 Resolução das proposições $(P \text{ OR } Q)$, $(R \text{ OR } \neg Q)$, $\neg R$ e $\neg P$.

*N. de T. Em português, *Programação em Lógica*.

Note que os argumentos neste predicado começam com letras minúsculas, embora representem nomes próprios. Isto ocorre porque a linguagem Prolog faz distinção entre constantes e variáveis, impondo que as constantes sejam iniciadas com letras minúsculas, enquanto as variáveis, com maiúsculas.

Em um programa escrito em Prolog, as proposições ou são fatos ou são regras, cada qual finalizada por um ponto. Um *fato* consiste em um único predicado. Por exemplo, o fato de uma tartaruga ser mais rápida que um caracol pode ser representado em Prolog por:

```
maisrapido (tartaruga, caracol).
```

e o fato de um coelho ser mais rápido que uma tartaruga, por

```
maisrapido (coelho, tartaruga).
```

Uma regra em Prolog corresponde a uma proposição “implica”. Em lugar de escrever tal proposição na forma $X \rightarrow Y$, um programador de Prolog escreve “ Y if X ”, exceto em que o símbolo :- (dois pontos seguidos de um traço) é usado no lugar da palavra *if*. Assim, a regra “ X é velho implica que X é justo” pode ser expressa, em lógica, como:

```
velho(X) → justo(X)
```

mas seria expressa em Prolog como:

```
justo(X) :- velho(X).
```

Como outro exemplo, a regra

```
maisrápido(X,Y) AND maisrápido(Y,Z) → maisrápido(X,Z)
```

seria expressa em Prolog como:

```
maisrapido(X, Z) :- maisrapido(X, Y), maisrapido(Y, Z).
```

A vírgula que separa `maisrapido (X, Y)` e `maisrapido (Y, Z)` representa a conjunção AND. Embora tais regras não estejam na forma de cláusulas, elas são permitidas em Prolog porque podem ser facilmente transformadas em cláusulas.

Convém lembrar que o sistema Prolog desconhece o significado dos predicados de um programa; ele simplesmente manipula as proposições de uma forma totalmente simbólica, de acordo com a regra de inferência da resolução. Desta maneira, é tarefa do programador descrever, em termos de fatos e regras, todas as características relativas aos predicados. Assim, os fatos são usados em Prolog para identificar determinadas instâncias de um predicado, enquanto as regras são usadas para descrever princípios gerais. Este foi o enfoque adotado na construção dos comandos vistos anteriormente, relativos ao predicado `maisrapido`. Os dois fatos descrevem instâncias particulares de “rapidez”, enquanto a regra descreve uma propriedade geral. Note-se que o fato de um coelho ser mais rápido do que um caracol, embora não declarado explicitamente, decorre da combinação dos dois fatos com a regra.

Quando se desenvolve *software* usando o Prolog, a tarefa do programador é desenvolver uma coleção de fatos e regras que descrevam a informação conhecida. Esses fatos e regras constituem o conjunto de proposições iniciais a ser usado no sistema dedutivo. Uma vez estabelecida esta coleção, as conjecturas (chamadas metas na terminologia Prolog) podem ser propostas ao sistema — geralmente por intermédio do teclado do computador. Apresentada a meta a um sistema Prolog, este aplica uma resolução para confirmar se a meta é consequência das proposições iniciais. Com base no conjunto de proposições que descrevem a relação `maisrapido` do nosso exemplo, todas as metas

```
maisrapido(tartaruga, caracol).
```

```
maisrapido(coelho, tartaruga).
```

```
maisrapido(coelho, caracol).
```

são confirmadas, dado que cada uma é consequência lógica das proposições iniciais. As duas primeiras são idênticas aos fatos que aparecem nas proposições iniciais, enquanto a terceira exige algum trabalho de dedução por parte do sistema.

Exemplos mais interessantes poderão ser obtidos se fornecermos variáveis como argumentos das metas, em vez de constantes. Nestes casos, o Prolog tenta deduzir a meta a partir das proposições iniciais, mantendo, ao mesmo tempo, um histórico das unificações aplicadas com essa finalidade. Então, se conseguir atingir a meta, o Prolog fornecerá como resultado essas unificações. Por exemplo, consideremos a meta:

```
maisrapido(W, caracol).
```

Em resposta a isto, o Prolog informa:

```
maisrapido(tartaruga, caracol).
```

De fato, isto é uma consequência das proposições iniciais, e é compatível com a meta através da unificação das proposições. Além disso, indo mais adiante neste processo, o resultado obtido seria:

```
maisrapido(coelho, caracol).
```

Entretanto, é possível solicitar ao Prolog que encontre instâncias de animais que sejam mais lentos que um coelho, propondo a meta:

```
maisrapido(coelho, W).
```

De fato, se iniciarmos com a meta:

```
maisrapido(V, W).
```

o Prolog informará todas as relações de `maisrapido` que possam ser derivadas a partir das proposições iniciais. Assim, um único programa, escrito em Prolog, pode ser utilizado para confirmar se um dado animal é mais rápido do que outro, encontrar os animais que são mais rápidos e os que são mais lentos que um determinado animal, ou achar todas as relações referentes à meta `maisrapido`. Esta versatilidade é um dos recursos que vem prendendo a imaginação dos cientistas da computação.



QUESTÕES/EXERCÍCIOS

1. Quais das proposições R , S , T , U e V são consequências lógicas do conjunto de proposições $(\neg R \text{ OR } T \text{ OR } S)$, $(\neg S \text{ OR } V)$, $(\neg V \text{ OR } R)$, $(U \text{ OR } \neg S)$, $(T \text{ OR } \neg U)$ e $(S \text{ OR } V)$?
2. O seguinte conjunto de proposições é consistente? Explique sua resposta.
 $P \text{ OR } Q \text{ OR } R \quad \neg R \text{ OR } Q \quad R \text{ OR } \neg P \quad \neg Q$
3. Suponha um programa em Prolog que consista nas seguintes proposições:
`maiseconomico(carol, john).`
`maiseconomico(bill, sue).`
`maiseconomico(sue, carol).`
`maiseconomico(X, Z) :- maiseconomico(X, Y), maiseconomico(Y, Z).`
 Liste os possíveis resultados das seguintes metas:
 - a. `maiseconomico(sue, V).`
 - b. `maiseconomico(U, carol).`
 - c. `maiseconomico(U, V).`
4. Complete as duas regras no final do programa Prolog abaixo de forma que o predicado `mae(X, Y)` signifique que X é a mãe de Y e o predicado `pai(X, Y)`, que X é o pai de Y .
`femea(carol).`

```
femea(sue).
macho(bill).
macho(john).
genitor(john, carol).
genitor(sue, carol).
mae(X,Y) :- 
    pai(X,Y) :-
```

Problemas de revisão de capítulo

(Os problemas marcados com asterisco se referem às seções opcionais.)

1. O que significa dizer que uma linguagem de programação é independente de máquina?

2. Traduza o seguinte programa em pseudocódigo para a linguagem de máquina descrita no Apêndice C.

```
x ← 0;
enquanto (x < 3) faça
    (x ← x + 1)
```

3. Traduza a instrução

Meiocaminho ← Comprimento + Largura
para a linguagem de máquina do Apêndice C, pressupondo que Comprimento, Largura e Meiocaminho sejam representados em notação de vírgula flutuante.

4. Traduza a instrução de alto nível

```
if (X = 0)
    then Z ← Y + W
    else Z ← Y + X
```

para a linguagem de máquina do Apêndice C, pressupondo que W, X, Y e Z são representados por números em notação de complemento de dois que ocupa um byte de memória.

5. Por que era necessário identificar o tipo de dados associado às variáveis do Problema 4 para traduzir as instruções? Por que muitas linguagens de alto nível exigem que o programador identifique o tipo de dado de cada variável no início de um programa?

6. Cite e descreva quatro paradigmas diferentes de programação.

7. Suponha que a função f receba dois valores numéricos como parâmetros, determine o menor deles e retorne o resultado como valor de saída. Se w, x, y e z representam valores numéricos,

qual será o resultado obtido ao calcular $f(f(w,x), f(y,z))$?

8. Seja f uma função que forneça o resultado da reversão de uma cadeia de símbolos e g uma função que retorne a concatenação de duas cadeias dadas como entrada. Se x for a cadeia abcd, qual será o resultado de $g(f(x),x)$?

9. Suponha que sua conta-corrente seja representada como um objeto, em um programa orientado a objeto projetado para manter o seu histórico financeiro. Que dados são armazenados neste objeto? Quais mensagens este objeto pode receber? Como são as respostas do objeto a tais mensagens? Que outros objetos poderiam ser usados no programa?

10. Resuma a diferença entre linguagem de máquina e linguagem de montagem.

11. Projete uma linguagem de montagem para a máquina descrita no Apêndice C.

12. O programador John argumenta que o recurso de declarar constantes dentro de um programa não é necessário, pois poderiam ser utilizadas variáveis em seu lugar. Por exemplo, o problema da altitude do aeroporto AltitudedoAeroporto da Secção 5.2 pode ser reescrito declarando AltitudedoAeroporto como variável e então atribuindo a ela o valor adequado no início do programa. Por que este método não é tão bom quanto o uso de uma constante?

13. Resuma a diferença entre as instruções declarativas e as imperativas.

14. Explique as diferenças entre um literal, uma constante e uma variável.

15. O que é precedência de operador?

- 16.** O que é programação estruturada?
- 17.** Qual é a diferença de significado entre os símbolos “igual” da instrução
`if (X = 5) then (...)`
e da instrução de atribuição
 $X = 2 + Y$
- 18.** Faça um fluxograma que represente a estrutura expressa pela seguinte instrução em C, C++ e Java.
`for (x = 2; x < 8; ++x)
{ . . . }`
- 19.** Faça um fluxograma que represente a estrutura expressa pela seguinte instrução em C, C++ e Java.
`switch (naipe)
{ case "paus": lance(1);
 case "ouros": lance(2);
 case "copas": lance(3);
 case "espadas": lance(4);
}`
- 20.** Se você estiver familiarizado com notação musical, analise-a do ponto de vista de uma linguagem de programação. Quais são as estruturas de controle? Qual é a sintaxe para a inserção de comentários? Qual notação musical se assemelha à instrução `for` da Figura 5.9?
- 21.** Reescreva o seguinte segmento de programa usando uma única instrução `case` no lugar das instruções aninhadas `if-then-else`:
`if (W = 5)
 then (Z ← 7)
 else (if (W = 6)
 then (Y ← 7)
 else (if (W = 7)
 then (X ← 7)
)
)`
- 22.** Resuma a seguinte rotina de ninho de rato com uma única instrução `if-then-else`.
`if X > 5 then goto 80
 X = X + 1
 goto 90
80 X = X + 2
90 stop`
- 23.** Resuma as estruturas básicas de controle encontradas nas linguagens de programação

imperativas e orientadas a objeto para realizar as seguintes atividades:

- Determinar que instrução deve ser executada a seguir.
 - Repetir uma coleção de instruções.
 - Mudar o valor de uma variável.
- 24.** Resuma as distinções entre um tradutor e um interpretador.
- 25.** Suponha que uma variável `X` em um programa tenha sido declarada como do tipo inteiro. Que erro ocorreria na execução da instrução
 $X \leftarrow 2.5$
- 26.** O que significa dizer que uma linguagem de programação é fortemente tipificada?
- 27.** Por que uma matriz grande dificilmente seria passada por valor a um procedimento?
- 28.** Suponha que o procedimento `modificar` seja definido por
`procedure Modificar(Y)
 Y ← 7;
 imprimir o valor de Y.`
Se os parâmetros forem passados por valor, o que será impresso quando o seguinte segmento de programa for executado? E se os parâmetros forem passados por referência?
 $X \leftarrow 5;$
aplicar o procedimento `Modificar` para `X`;
imprimir o valor de `X`;
- 29.** Suponha que o procedimento `modificar` seja definido por
`procedure Modificar(Y)
 Y ← 9;
 imprimir o valor de X;
 imprimir o valor de Y.`
Suponha também que `X` seja uma variável global. Se os parâmetros forem passados por valor, o que será impresso quando o seguinte segmento de programa for executado? E se os parâmetros forem passados por referência?
 $X \leftarrow 5;$
aplicar o procedimento `Modificar` para `X`;
imprimir o valor de `X`;
- 30.** Algumas vezes, um parâmetro real é passado para um procedimento por meio da produção de uma duplicata a ser usada pelo procedimento

(como quando os parâmetros são passados por valor), mas quando o procedimento termina, o valor na cópia do procedimento é transferido para o parâmetro real antes que o procedimento chamador continue. Nesses casos, diz-se que a passagem de parâmetros é por valor-resultado. O que seria impresso pelo segmento de programa do exercício 28 se os parâmetros fossem passados por valor-resultado?

31. Que ambigüidade existe na instrução:

$X \leftarrow 3 + 2 \times 5$

32. Suponha que uma companhia pequena tenha cinco funcionários e planeje aumentar para seis. A seguir, são apresentados trechos de dois programas equivalentes, utilizados pela companhia, que devem ser modificados para corresponder à alteração do número de funcionários. Ambos estão escritos em uma linguagem semelhante a Ada. Indique quais mudanças devem ser feitas em cada um. Que complicações existentes no programa 1 foram evitadas no programa 2 pela utilização de constantes?

Programa 1

```

.
.
.

SalarioDiario:= SalarioTotal/5;
SalarioMedio:= SalarioTotal/5;
VendasDiarias:= VendasTotal/5;
VendasMedio:= VendasTotal/5;

.
.
.
```

Programa 2

```

.
.
.

NumeroDeFuncionarios constant
  Integer:= 5;
DiasPorSemana constant Integer:= 5;
.

.
```

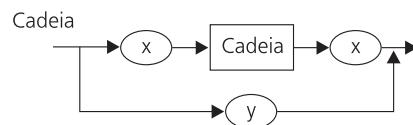
```

SalarioDiario:= SalarioTotal/
  DiasPorSemana;
SalarioMedio:= SalarioTotal/
  NumeroDeFuncionarios;
```

```

VendasDiarias:= VendasTotal/
  DiasPorSemana;
VendasMedio:= VendasTotal/
  NumeroDeFuncionarios;
.
.
```

33. Faça um diagrama de sintaxe que represente a estrutura da instrução enquanto, do pseudocódigo do Capítulo 4.
34. Projete um conjunto de diagramas de sintaxe para descrever a sintaxe dos números de telefone escritos na forma do seguinte exemplo: (444) 555-1234.
35. Projete um conjunto de diagramas de sintaxe para descrever sentenças simples, escritas na língua portuguesa.
36. Projete um conjunto de diagramas de sintaxe que descreva a estrutura gramatical de “sentenças” que consistam em ocorrências da palavra *sim* seguidas pelo mesmo número de palavras *não*. Por exemplo, “sim sim não não” seria uma sentença, enquanto “não sim”, “sim não não” e “sim não sim” não seriam.
37. Dê um argumento que explique por que não se pode projetar um conjunto de diagramas de sintaxe que descreva a estrutura gramatical de “sentenças” que consistem em ocorrências da palavra *sim* seguidas pelo mesmo número de ocorrências da palavra *não*, seguidas pelo mesmo número de ocorrências da palavra *talvez*. Por exemplo, “sim não talvez” e “sim sim não não talvez talvez” seriam sentenças, enquanto “sim talvez”, “sim não não talvez talvez” e “talvez não” não seriam.
38. Escreva uma frase definindo a estrutura de uma cadeia como o foi no diagrama abaixo, e desenhe a árvore de sintaxe para a cadeia *xyyxx*.



39. Acrescente diagramas de sintaxe aos itens da Questão 4 da Seção 5.4 para obter um conjunto de diagramas que definam a estrutura *Dance*

como ou um Chacha ou uma Valsa, onde Valsa consiste em uma ou mais cópias do padrão

frente diagonal fechar

ou

atrás diagonal fechar

- 40.** Desenhe a árvore de sintaxe para a expressão $x \times y + y \div x$

com base nos diagramas de sintaxe da Figura 5.19.

- 41.** Qual otimização pode ser executada por um gerador de código ao construir o código de máquina que representa a instrução
if ($X = 5$) **then** ($Z \leftarrow X + 2$)
else ($Z \leftarrow X + 4$)

- 42.** Simplifique o seguinte segmento de programa:
 $Y \leftarrow 5;$
if ($Y = 7$)
then ($Z \leftarrow 8$)
else ($Z \leftarrow 9$)

- 43.** Simplifique o seguinte segmento de programa:
while ($X \neq 5$) **do**
 $(X \leftarrow 5)$

- *44.** Em um ambiente de programação orientada a objeto, como os tipos são similares às classes? Como eles diferem?

- *45.** Descreva como a herança pode ser usada para desenvolver classes que descrevam vários tipos de edificações.

- *46.** Dê uma definição de encapsulamento com suas próprias palavras.

- *47.** Qual é a diferença entre as partes pública e privada de uma classe?

- *48.** Faça um diagrama (semelhante ao da Figura 5.27) que represente as resoluções necessárias para provar que o conjunto de proposições ($Q \text{ OR}$

$\neg R), (T \text{ OR } R), \neg P, (P \text{ OR } \neg T) \text{ e } (P \text{ OR } \neg Q)$ é inconsistente.

- *49.** O conjunto de proposições $\neg R, (T \text{ OR } R), (P \text{ OR } \neg Q), (Q \text{ OR } \neg T) \text{ e } (R \text{ OR } \neg P)$ é consistente? Explique a sua resposta.

- *50.** Quais conclusões podem ser obtidas pelo sistema Prolog para a meta

`maior(X, lassie).`

com as proposições iniciais:

`maior(rex, lassie).`

`maior(fido, rex).`

`maior(spot, rex).`

`maior(X, Z) :-`

`maior(X, Y),`

`maior(Y, Z).`

- *51.** Quais conclusões podem ser obtidas pelo sistema Prolog para a meta

`igual(X, Y).`

com as proposições iniciais:

`maiorouigual(a,b).`

`maiorouigual(b,c).`

`maiorouigual(c,a).`

`maiorouigual(U,W) :-`

`maiorouigual(U,V), maiorouigual(V,W).`

`igual(X,Y) :-`

`maiorouigual(X,Y), maiorouigual(Y,X).`

- *52.** Que problemas aconteceriam se o seguinte segmento de programa fosse executado em uma máquina na qual os valores fossem representados em notação de vírgula flutuante com oito bits descrito na Seção 1.7?

$X \leftarrow 0.01;$

while ($X \neq 1.00$) **do**

(imprimir o valor de X;

$X \leftarrow X + 0.01)$

Questões sociais

As seguintes questões procuram auxiliar o leitor no entendimento de alguns assuntos éticos, sociais e legais no campo da computação. O objetivo não é meramente o de fornecer respostas a tais questões. O leitor também deve justificá-las e verificar se as justificativas apresentadas preservam sua consistência de uma questão para outra.

1. Em geral, as leis de direitos autorais apóiam direitos de propriedade associados à expressão de uma idéia, mas não à idéia propriamente dita. Como resultado, um parágrafo de um livro é passível de direitos autorais, mas as idéias apresentadas nesse parágrafo, não. Como este direito poderia ser estendido a programas-fonte e aos algoritmos expressos por eles? Até que ponto você acredita que uma pessoa que conheça os algoritmos utilizados em um pacote comercial de *software* está autorizada a escrever o seu próprio programa com estes algoritmos e a comercializar esta nova versão?
2. Ao usar uma linguagem de programação de alto nível, um programador se habilita a expressar algoritmos usando palavras como *if*, *then* e *while*. Até que ponto o computador entende o significado de tais palavras? A habilidade de responder corretamente ao uso de palavras implica em entendimento das mesmas? Como você sabe quando outra pessoa entendeu o que você disse?
3. Uma pessoa que desenvolveu uma linguagem de programação nova e útil deve ter direito a lucro pelo uso dessa linguagem? Nesse caso, como esse direito deve ser protegido? Até que ponto uma linguagem pode ser propriedade de alguém? Até que ponto uma empresa pode ter direito de propriedade em relação às realizações criativas e intelectuais de seus funcionários?
4. Até que ponto um programador que participa do desenvolvimento de um jogo eletrônico violento é responsável pelas suas consequências? O acesso a jogos eletrônicos violentos a crianças deveria ser restringido? Nesse caso, de que forma e por quem? E quanto aos outros grupos da sociedade, como criminosos condenados?
5. Com a aproximação do prazo final, é aceitável que um programador negligencie a documentação, na forma de comentários, para que o programa seja terminado a tempo? (Estudantes calouros freqüentemente se surpreendem com a importância atribuída à documentação pelos profissionais de desenvolvimento de *software*.)
6. Boa parte da pesquisa em linguagens de programação tem sido para desenvolver linguagens que permitam ao programador escrever programas fáceis de ler e entender. Até que ponto deve-se exigir que um programador use essas facilidades? Isto é, até que ponto importa se o programa executa corretamente, ainda que não seja bem escrito sob uma perspectiva humana?
7. Suponha que um programador amador escreva um programa para o seu uso pessoal e por isso seja relaxado em sua construção. O programa não usa os recursos da linguagem de programação que o fariam mais legível, não é eficiente e contém simplificações que tiram vantagem de situações particulares nas quais o programador pretende usar o programa. Ao longo do tempo, o programador fornece cópias do programa a seus amigos, que também distribuem a seus amigos. Até que ponto o programador é responsável por problemas que possam ocorrer?
8. Até que nível um profissional da computação deve conhecer os vários paradigmas de programação? Algumas companhias impõem que todo *software* por elas desenvolvido seja escrito em uma mesma linguagem de programação predeterminada. A sua resposta para a pergunta inicial seria diferente para um profissional que trabalhasse nesta companhia?

Leituras adicionais

Aho, A. V., R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.
Archer, T. *Inside C#*. Redmond, WA: Microsoft Press, 2001.

Barnes, J. *Programming in Ada 95*. 2nd ed. Reading, MA: Addison-Wesley, 1998.

Bergin, T. J., and R. G. Gibson. *History of Programming Languages*. New York: ACM Press, 1996.

Clocksin, W. F., and C. S. Mellish. *Programming in Prolog*, 4th ed. New York: Springer-Verlag, 1997.

Graham, P. *ANSI Common Lisp*. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Kelley, A. and I. Pohl. *C by Dissection: The Essentials of C Programming*, 4th ed. Boston: Addison-Wesley, 2001.

Lewis, J. and W. Loftus. *Java Software Solutions: Foundations of Program Design*, 2nd ed. Boston: Addison-Wesley, 2002.

Metcalf, M., and J. Reid. *Fortran 90/95 Explained*. 2nd ed. Oxford, England: Oxford University Press, 1999.

Noonan, R. and A. Tucker. *Programming Languages: Principles and Paradigms*, Burr Ridge, IL: McGraw-Hill, 2001.

Pratt, T. W., and M. V. Zelkowitz. *Programming Languages, Design and Implementation*, 4nd ed. Englewood Cliffs, NJ: Prentice-Hall, 2001.

Savitch, W. *Problem Solving with C++*, 3rd ed. Boston: Addison-Wesley, 2001.

Sebesta, R. W. *Concepts of Programming Languages*, 5th ed. Boston: Addison-Wesley, 2001.

6

C A P Í T U L O

Engenharia de software

Neste capítulo, analisaremos tópicos relativos ao processo global de desenvolvimento de *software* e sua manutenção. O assunto é chamado engenharia de *software* porque o desenvolvimento de *software* é um processo de engenharia. Contudo, a engenharia de *software* possui muitas características que fazem com que ela seja única na disciplina de engenharia.

A nossa discussão se refere a sistemas de *software* de grande porte. Os problemas encontrados no desenvolvimento de tais sistemas são mais que uma versão amplificada dos problemas encontrados quando se escreve pequenos programas. Para exemplificar, o desenvolvimento de tais sistemas exige os esforços de diversas pessoas por um longo tempo, durante o qual as necessidades do sistema proposto podem ser modificadas e o pessoal encarregado do projeto pode sofrer alterações. Por conseguinte, a Engenharia de *Software* inclui tópicos, como pessoal e administração de projeto, que dizem respeito mais diretamente à administração empresarial do que à Ciência da Computação. Entretanto, focalizaremos os tópicos diretamente relacionados à Ciência da Computação.

6.1 A disciplina da engenharia de *software*

6.2 O ciclo de vida do *software*

O ciclo como um todo

A fase tradicional de desenvolvimento

Tendências modernas

6.3 Modularidade

Implementação modular

Acoplamento

Coesão

6.4 Metodologias de projeto

Cima-baixo versus baixo-cima

Padrões de projeto

Desenvolvimento de código aberto

Ferramentas do ofício

6.5 Ferramentas do ofício

6.6 Testes

6.7 Documentação

6.8 Propriedade e responsabilidade de *software*

6.1 A disciplina da engenharia de *software*

Para apreciar os problemas envolvidos na engenharia de *software*, é útil selecionar um dispositivo complexo qualquer (um automóvel, um edifício comercial com muitas lojas, ou talvez uma catedral), imaginar o seu projeto e a supervisão de sua construção. Como estimar o custo, o tempo necessário e outros recursos para completar o projeto? Como dividir o projeto em partes operacionais? Como assegurar que tais partes, uma vez implementadas, serão compatíveis entre si? Como será a comunicação dos envolvidos nas diversas partes? Como medir seu progresso? Como trabalhar com a imensa gama de detalhes (a seleção das maçanetas das portas, o projeto das carrancas, a disponibilidade de vidro azul para os vitrais, a capacidade de sustentação dos pilares, o projeto do tubo utilizado no sistema de aquecimento)? Questões do mesmo escopo devem ser respondidas durante o desenvolvimento de um sistema de *software* de grande porte.

Como a engenharia é uma área bem definida, pode dar a impressão de que existe uma infinidade de técnicas úteis, já desenvolvidas, para responder a tais questões. Este raciocínio é parcialmente verdadeiro, mas despreza as muitas diferenças existentes entre as características do *software* e as dos outros campos da engenharia.

Uma distinção se refere à possibilidade de construir sistemas a partir de componentes genéricos pré-fabricados. Os campos tradicionais da engenharia há muito tempo se beneficiam do uso de componentes “na prateleira”, como módulos na construção de dispositivos complexos. O projetista de um novo carro não terá de projetar o motor, o rádio, o ar-condicionado e as trancas das portas. Em vez disso, ele usa versões previamente projetadas desses componentes. No contexto do *software*, os componentes previamente projetados tendem a ser específicos a cada domínio — isto é, seu projeto interno depende de uma aplicação específica. Reusar tal componente exige, portanto, que ele seja reprojeto. O resultado é que os sistemas complexos de *software* têm sido historicamente construídos a partir do zero.

Outra distinção entre a engenharia de *software* e a tradicional se refere à tolerância. Áreas tradicionais da engenharia cuidam do desenvolvimento de produtos considerados aceitáveis se executarem suas tarefas dentro de certos limites. Uma lavadora de roupa que gira em torno do seu eixo com 2% de tolerância em relação ao tempo especificado é considerada aceitável. O *software*, ao contrário, apresenta um funcionamento correto ou incorreto. Um sistema de contabilidade que apresente uma precisão com tolerância de 2% não é aceitável.

Outra diferença se refere à escassez de sistemas quantitativos, chamados **métricas**, para medir as características de *software*. A qualidade de um dispositivo mecânico freqüentemente é medida em termos do tempo médio entre suas falhas, que é uma medida da capacidade do dispositivo de suportar desgaste. O *software*, ao contrário, não se desgasta, portanto, tal método de medição de qualidade não pode ser aproveitado pela engenharia de *software*.

Esta dificuldade para medir características de *software* de uma maneira quantitativa é uma das principais razões pelas quais a engenharia de *software* ainda não encontrou uma base rigorosa de sustentação, como a que existe em engenharia mecânica ou elétrica. Enquanto esses assuntos se fundamentam na ciência estabelecida da física, a engenharia de *software* continua a procurar suas próprias raízes. Sem dúvida, o seu estudo é similar ao da engenharia mecânica no início do século XVII, antes que Isaac Newton e outros descobrissem que propriedades como massa, aceleração e força podem ser medidas e relacionadas matematicamente.

Association for Computing Machinery

A Association for Computing Machinery (ACM) foi fundada em 1947 como uma organização internacional, científica e educacional dedicada ao avanço da arte, ciência e aplicações da tecnologia da informação. Sediada em Nova Iorque, engloba numerosos grupos de interesse especial, enfocando tópicos como arquitetura de computadores, inteligência artificial, computação biomédica, computadores e sociedade, educação na ciência da computação, computação gráfica, hipertexto/hipermídia, sistemas operacionais, linguagens de programação, simulação e modelagem e engenharia de *software*. Seu sítio na Web é <http://www.acm.org>.

Assim, no momento, pesquisar em engenharia de *software* é avançar em dois níveis: alguns pesquisadores, algumas vezes chamados práticos, trabalham para o desenvolvimento de técnicas para aplicação imediata, enquanto outros, chamados teóricos, buscam princípios subjacentes e teorias, com os quais algum dia serão construídas técnicas mais estáveis. Por estarem apoiadas em bases subjetivas, muitas metodologias desenvolvidas e promovidas pelos práticos no passado foram substituídas por outras abordagens que, por sua vez, poderão ficar obsoletas com o tempo. Enquanto isso, os avanços dos teóricos continuam sendo evasivos.

A necessidade de avanços práticos e teóricos é imensa. Nossa sociedade se mostra dependente dos sistemas computacionais e dos *softwares* correspondentes. Nossa economia, planos de saúde, governo, cumprimento das leis, transporte e defesa dependem de sistemas de *software* de grande porte, cuja confiabilidade continua sendo um dos principais desafios. Erros de *software* têm causado muitos problemas: um nascer da lua, interpretado como um ataque nuclear; a perda de \$5 milhões pelo Banco de Nova Iorque em um único dia; a perda da sonda espacial Mariner 18; sobrecargas de radiação, que causaram mortes e paralisias; e a queda simultânea das comunicações telefônicas em vastas regiões.

Enquanto a ciência continua procurando métodos de desenvolver *software* de melhor qualidade, organizações profissionais contribuíram, indiretamente, ao promover altos padrões de ética e conduta profissional entre seus membros. Por exemplo, A Association of Computing Machinery (ACM) e o Institute of Electrical and Electronics Engineers (IEEE) adotaram códigos de conduta profissional e de ética que aperfeiçoam o profissionalismo dos projetistas de *software*, promovendo as responsabilidades de cada indivíduo.

Neste capítulo, introduziremos alguns dos resultados provenientes das pesquisas em engenharia de *software*, incluindo alguns dos seus princípios básicos (ciclo de vida, modularidade e padrões de projeto), bem como algumas ferramentas e técnicas de desenvolvimento utilizadas em nossos dias.



QUESTÕES/EXERCÍCIOS

1. Por que o número de linhas em um programa não é uma boa medida da sua complexidade?
2. Qual técnica pode ser utilizada para determinar o número de erros existentes em um trecho de *software*?
3. Sugira uma métrica para medir a qualidade de um *software*. Quais são as fragilidades desta métrica?

Institute of Electrical and Electronics Engineers

O Institute of Electrical and Electronics Engineers (IEEE, pronuncia-se “ai-trípou- i”) foi formado em 1963 como resultado da fusão do American Institute of Electrical Engineers (fundado em 1884 por 25 engenheiros eletricistas, inclusive Thomas Edison) e o Institute of Radio Engineers (fundado em 1912). Sediado em Londres, o IEEE é uma organização internacional de engenheiros eletricistas, eletrônicos e industriais. Ele engloba 36 sociedades técnicas tais como a Aerospace and Electronic Systems Society, a Lasers and Electro-Optics Society, a Robotics and Automation Society, a Vehicular Technology Society, e (a mais importante para o nosso estudo) a Computer Society. Dentre suas atividades, o IEEE se envolve no desenvolvimento de padrões. Em particular, foram os esforços do IEEE que levaram à padronização da notação de vírgula flutuante.

Você encontrará a página Web do IEEE em <http://www.ieee.org> e da Computer Society em <http://www.computer.org>

6.2 O ciclo de vida do software

O conceito mais fundamental em engenharia de *software* é o ciclo de vida do *software*.

O ciclo como um todo

O ciclo de vida do *software* é mostrado na Figura 6.1. Ela comprova que, uma vez desenvolvido, o *software* inicia um ciclo no qual é utilizado, depois modificado e assim por diante, pelo resto de sua vida. Este modo de funcionamento é observado em muitos produtos manufaturados. A diferença é que, no caso de outros produtos, sua fase de modificação é mais precisamente denominada fase de reparo ou manutenção, pois tais produtos saem do estado de utilização para o de modificação quando alguma de suas partes se desgasta.

Entretanto, um *software* não se desgasta. Ele entra em fase de modificação porque são descobertos erros, porque ocorrem mudanças da aplicação do programa, que exigem modificações correspondentes no *software*, ou porque se descobriu que alterações feitas em uma modificação anterior induziram ao aparecimento de problemas em alguma outra parte do *software*. Por exemplo, mudanças nas leis de arrecadação de impostos em geral exigem modificações nos programas de folha de pagamento que calculam impostos retidos na fonte, o que normalmente acarreta efeitos adversos em outras partes do programa, os quais podem permanecer despercebidos por algum tempo.

Independentemente do motivo pelo qual um *software* entra na fase de modificação, este processo requer que uma pessoa (que raramente é o próprio autor do programa original) estude e compreenda o programa, ou parte dele, e sua documentação, caso contrário, qualquer modificação poderia introduzir problemas adicionais. Esta é uma tarefa muito difícil, mesmo com softwares bem projetados e documentados. Em geral, é nesta fase que trechos de *software* são descartados sob o pretexto (frequentemente verdadeiro) de ser mais fácil desenvolver um sistema novo a partir do zero do que conseguir modificar um pacote já existente.

Experiências mostraram que um pequeno esforço adicional durante a fase de desenvolvimento de um *software* pode representar uma grande diferença, no caso de modificação. Por exemplo, na nossa discussão acerca das instruções para a descrição de dados, no Capítulo 5, vimos como o nome Altitude do Aeroporto poderia ser usado no lugar do valor não-descritível 645 do programa e argumentamos que se alguma modificação fosse necessária, seria mais fácil alterar o valor associado ao nome em vez de encontrar e alterar todas as inúmeras ocorrências deste valor. Por sua vez, a maioria das pesquisas na engenharia de *software* se concentra na fase de desenvolvimento do ciclo de vida do *software*, com o objetivo de aproveitar ao máximo a relação custo/benefício associada a tais esforços.

A fase tradicional de desenvolvimento

Os estágios da fase de desenvolvimento do ciclo de vida de um *software* são análise, projeto, implementação e teste (Figura 6.2).

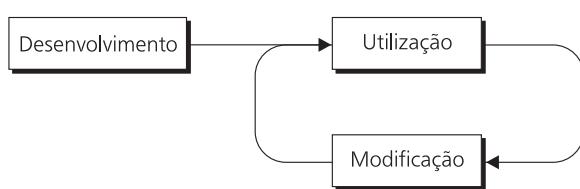


Figura 6.1 O ciclo de vida do *software*.

Análise A fase de desenvolvimento do ciclo de vida de um *software* começa com a análise — a meta principal é identificar o quê o sistema proposto deve fazer. Se o sistema deve ser um produto genérico, vendido em um mercado competitivo, essa análise envolve uma extensa investigação para identificar as necessidades dos consumidores em potencial. Entretanto, se o sistema for projetado para um usuário específico, então o processo deverá ser uma investigação mais estreita.

Uma vez identificadas as necessidades do usuário em potencial, elas são reunidas para formar um conjunto de **requisitos** a que o novo sistema deve satisfazer. Esses requisitos são expressos na linguagem do usuário, e não na terminologia técnica usada pela comunidade de processamento de dados. Um dos requisitos é que o acesso aos dados fique restrito apenas ao pessoal autorizado. Outro, que os dados representem o estado corrente do cadastro, como, por exemplo, no final do último dia de negócios, ou que a apresentação dos dados na tela do computador seja compatível com o formato do formulário em uso.

Depois de identificados, os requisitos do sistema são traduzidos para a forma de **especificações** mais técnicas. Por exemplo, o requisito de os dados serem restritos apenas ao pessoal autorizado passa a ser uma especificação em que o sistema não responderá se não for digitada, no terminal, uma senha autorizada de oito dígitos; caso contrário, tais dados serão apresentados em forma codificada, até que sejam pré-processados por uma rotina, conhecida apenas por pessoal autorizado.

Projeto Enquanto a análise se concentra no *quê* o sistema proposto deve fazer, o projeto se concentra em *como* o sistema atingirá as metas. Aqui a estrutura do sistema de software é estabelecida.

É consenso que a melhor estrutura para um grande sistema de software é a modular. De fato, é por meio dessa decomposição modular que se torna possível a implementação de sistemas de grande porte. Sem ela, os detalhes técnicos necessários à sua implementação excederiam a capacidade da compreensão humana. Com um projeto modular, porém, somente os detalhes pertinentes ao módulo em questão são considerados de cada vez. Esta modularização também se aplica em futuras manutenções, pois permite fazer alterações com base nos módulos. (Se for feita uma alteração no método de cálculo dos benefícios do plano de saúde de cada funcionário, então apenas os módulos relacionados com este assunto serão considerados.)

Contudo, existem distinções em relação ao conceito de módulo. Se abordarmos a tarefa de projeto em termos do paradigma imperativo tradicional, os módulos irão consistir em procedimentos e o desenvolvimento de um projeto modular passará a ser a identificação das várias tarefas que o sistema proposto deve realizar. Em contraste, se abordarmos a tarefa de projeto a partir da perspectiva orientada a objeto, os módulos

A engenharia de software no mundo real

O seguinte cenário é comum quanto aos problemas encontrados pelos engenheiros de software do mundo real. A Companhia XYZ contrata uma firma de engenharia de software para desenvolver e instalar um sistema de software integrado para suprir as suas necessidades de processamento de dados. Como parte do sistema produzido pela Companhia XYZ, uma rede de PCs é usada para que os funcionários tenham acesso ao sistema geral da companhia. Assim, cada funcionário tem um PC na sua mesa. Pouco tempo depois, esses PCs são usados não apenas para acessar o novo sistema de gerência de dados, mas também como ferramentas personalizadas, com as quais cada funcionário procura aumentar a sua produtividade. Por exemplo, um funcionário desenvolve um programa de planilha que acelera a execução de suas tarefas. Infelizmente, essas aplicações personalizadas podem não ser bem projetadas, muitas vezes não são completamente testadas e podem incluir recursos que não sejam bem entendidos pelo funcionário. Com o passar dos anos, o uso dessas ferramentas precárias se integra aos procedimentos internos da empresa. (O funcionário que desenvolveu o programa de planilha pode sair da companhia, deixando os outros com um programa que eles não conhecem bem.) O resultado é que o sistema inicial, bem projetado e coerente, pode tornar-se dependente de aplicações sujeitas a erros, mal documentadas e mal projetadas.

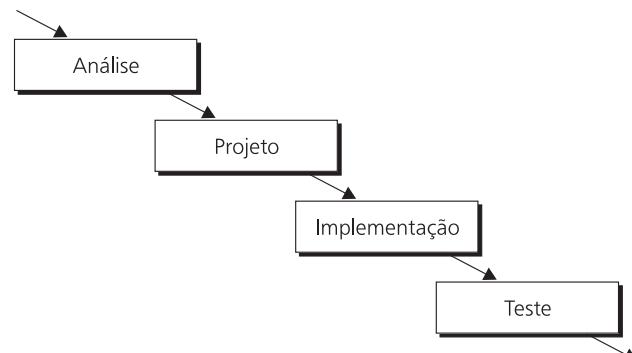


Figura 6.2 O ciclo de vida do software.

serão vistos como objetos e o processo de projeto passará a ser a identificação das entidades (objetos) no sistema proposto, bem como a maneira como elas devem se comportar.

Implementação Esta fase envolve a elaboração dos programas propriamente ditos, a criação dos arquivos de dados e o desenvolvimento de bancos de dados.

Teste Esta fase está intimamente associada à anterior, pois cada módulo do sistema normalmente é testado de acordo com o progresso de sua implementação. De fato, em um sistema bem projetado, cada módulo pode ser testado independentemente, substituindo os demais por versões simplificadas, chamadas **terminações**^{*}, encarregadas de simular a interação entre o módulo em estudo e o restante do sistema. É óbvio que esse teste individual deverá ser substituído pelo teste global do sistema, na época em que os vários módulos existentes estiverem concluídos e inter-relacionados.

Infelizmente, é difícil demais executar com sucesso o teste e a depuração de um sistema. A experiência mostra que sistemas de *software* de grande porte podem conter muitos erros, mesmo depois de aprovados em testes rigorosos. Muitos desses erros não são detectados em toda a vida útil do sistema, e outros podem causar falhas extremamente críticas. A eliminação de tais erros é uma das metas da engenharia de *software*, e o fato de eles continuarem subsistindo significa que ainda há muita pesquisa a fazer.

Tendências modernas

Os primeiros métodos, em engenharia de *software*, baseavam-se em uma rígida obediência à seqüência de desenvolvimento de *software*: análise, projeto, implementação e teste. Achava-se que, durante o desenvolvimento de um sistema de *software* de grande porte, muita coisa seria posta em risco se fosse permitido utilizar técnicas de tentativa e erro. Como resultado, os engenheiros de *software* insistiram em que a análise completa do sistema fosse feita antes da fase do projeto e, do mesmo modo, que o projeto fosse concluído antes da fase de implementação. O resultado era um processo de desenvolvimento denominado **modelo de cachoeira**^{**}, uma alusão ao fato de ser permitido ao processo de desenvolvimento fluir em uma única direção.

Note uma semelhança entre as quatro fases de resolução de problemas, identificadas por Polya (Seção 4.3), e as fases de desenvolvimento de *software*: análise, projeto, implementação e teste. Afinal de contas, desenvolver um sistema de *software* de grande porte é resolver um problema. Entretanto, o tradicional método da cachoeira, para o desenvolvimento de *software*, é totalmente contrário ao processo de livre escolha por tentativa e erro, que em geral se mostra vital para soluções criativas. Enquanto o método da cachoeira busca definir um ambiente altamente estruturado, no qual o desenvolvimento avança de modo seqüencial, resoluções criativas buscam um ambiente não-estruturado, no qual o programador possa fazer suas primeiras tentativas, baseadas na intuição, sem saber explicar exatamente o motivo de agir assim.

Recentemente, as técnicas da engenharia de *software* começaram a refletir essa contradição subjacente, como ilustrado pela emergência do **modelo incremental** para desenvolvimento de *software*. Segundo esse modelo, o sistema de *software* desejado é construído por incrementos — o primeiro é uma versão simplificada do produto final com funcionalidade limitada. Uma vez testada essa versão e talvez analisada pelos futuros usuários, novos recursos são incorporados e testados de uma maneira incremental até que o sistema esteja completo. Por exemplo, se o sistema a ser desenvolvido for de registro de estudantes a ser utilizado por uma universidade, o primeiro incremento poderá incorporar apenas a facilidade de visualizar os registros de estudantes. Uma vez operacional, serão adicionadas a essa versão habilidades como a de acrescentar e atualizar registros, uma adição feita passo a passo.

*N. de T. Em inglês, *stubs*.

**N. de T. Em inglês, *waterfall model*.

O modelo incremental evidencia a tendência no desenvolvimento de *software* para a **prototipação**, na qual versões incompletas do sistema proposto, chamadas **protótipos**, são construídas e analisadas. No caso do modelo incremental, esses protótipos evoluem para o sistema completo final — um processo conhecido como **prototipação evolucionária**. Em outros casos, os protótipos são descartados, dando lugar a novas implementações do projeto final. Essa abordagem é conhecida como **prototipação descartável** (*throwaway prototyping*). Um exemplo que normalmente se enquadra nessa categoria descartável é a prototipação rápida, na qual um exemplar simples do sistema proposto é rapidamente construído no estágio inicial do desenvolvimento. Tal protótipo pode consistir apenas em algumas imagens na tela que indiquem como o sistema deverá interagir com o usuário e que recursos ele terá. O objetivo não é produzir uma versão funcional do produto, mas obter uma ferramenta de demonstração que possa ser usada para esclarecer a comunicação entre as partes envolvidas. Por exemplo, protótipos rápidos têm sido vantajosos na confirmação dos requisitos durante a fase de análise ou como ajuda nas apresentações de venda aos clientes em potencial.

Atualmente, as metodologias da engenharia de *software* variam em um amplo espectro de filosofias. Em uma extremidade, está o tradicional modelo da cachoeira, caracterizado de certa forma pela imagem de gerentes e programadores que trabalham em escritórios individuais — cada qual realizando uma parte bem definida da tarefa de desenvolvimento do *software*. Na outra extremidade, está o modelo (carinhosamente conhecido como **programação extrema**) no qual um grupo de menos que uma dúzia de indivíduos divide um ambiente de trabalho onde livremente compartilham idéias e assistência à medida que desenvolvem incrementalmente um pacote de *software*, repetindo diariamente o ciclo de projeto, implementação e teste.

Outro desenvolvimento na engenharia de *software* tem sido a aplicação da tecnologia da computação ao processo de desenvolvimento de *software*, que resultou na chamada **Engenharia de Software Assistida por Computador (CASE)**^{*}. Esses sistemas mecanizados são conhecidos como ferramentas CASE e incluem ferramentas de planejamento de projeto (para estimativas de custo, no escalonamento do projeto e na alocação de pessoal), ferramentas de gerência de projeto (contribuem na monitoração do progresso no desenvolvimento do projeto), ferramentas de documentação (para a escrita e a organização da documentação), ferramentas de prototipação e simulação (que auxiliam no desenvolvimento de protótipos), ferramentas de projeto de interfaces (que auxiliam no desenvolvimento de GUIs), e ferramentas de programação (para a escrita e depuração de programas). Algumas dessas ferramentas são pouco mais do que processadores de texto, sistemas de planilhas e sistemas de comunicação por correio eletrônico, usados em outras aplicações. Outras são pacotes sofisticados projetados primordialmente para o ambiente da engenharia de *software*. Por exemplo, algumas ferramentas CASE incluem geradores de código que, quando alimentados com as especificações de um pacote de *software*, produzem programas em linguagem de alto nível que implementam parte do sistema.



QUESTÕES/EXERCÍCIOS

1. Qual é a diferença entre requisitos de sistema e especificações de sistema?
2. Resuma as quatro etapas (análise, projeto, implementação e teste) da fase de desenvolvimento do ciclo de vida do *software*.
3. Resuma a diferença entre o tradicional modelo da cachoeira, para o desenvolvimento de *software*, e o mais recente paradigma da prototipação.

*N. de T. Sigla do inglês, Computer-Aided Software Engineering.

6.3 Modularidade

Uma das afirmações centrais da Seção 6.2 sustenta que, para modificar um *software*, é necessário compreender o programa, ou pelo menos as partes do programa que mais interessam. Tal compreensão em geral já é suficientemente difícil no caso de pequenos programas, e seria quase impossível em sistemas de *software* de grande porte, não fosse o conceito de **modularidade**, prática de dividir o *software* em módulos manipuláveis, cada qual projetado para executar somente uma parte da tarefa global.

Implementação modular

A modularidade pode ser alcançada de diversas formas. No Capítulo 5, vimos como os módulos podem ser formados em termos de procedimentos e usados como blocos construtores (ferramentas abstratas) na implementação de sistemas maiores. No Capítulo 5, também encontramos a modularidade no contexto do paradigma orientado a objeto. Nesse caso, os módulos se apresentam na forma de objetos, cada qual com suas próprias estruturas internas, que são independentes dos conteúdos de outros objetos. É essa estrutura modular inerente que deu popularidade ao enfoque orientado a objeto para o desenvolvimento de *software*.

O **diagrama estrutural**^{*} é a ferramenta tradicional para a representação da estrutura modular obtida por meio de procedimentos. Nesse diagrama, cada módulo (procedimento) é representado por um retângulo, e as dependências entre os módulos, por setas que conectam os retângulos. O diagrama estrutural da Figura 6.3 representa a estrutura modular de uma loja virtual da Internet, na qual os clientes acessam o sítio da companhia, pesquisam o catálogo de produtos e fazem pedidos. O diagrama indica que o módulo chamado VisãoGeralSítio usa os módulos ProcessaNovoCliente, VisãoGeralCatálogo e ProcessaPedido como ferramentas abstratas para realizar a sua tarefa. Mais precisamente, o procedimento VisãoGeralSítio chama o procedimento ProcessaNovoCliente para realizar os passos necessários para obter as informações dos clientes que acessam o sítio. Ele então chama o procedimento VisãoGeralCatálogo para permitir a consulta ao catálogo e a produção do pedido pelos clientes. Uma vez produzido o pedido, VisãoGeralSítio chama o procedimento ProcessaPedido para coordenar as atividades de embarque e cobrança.

Enquanto os diagramas estruturais são usados para representar a organização de procedimentos de um sistema de *software*, os **diagramas de classes** freqüentemente são usados para representar a estrutura de sistemas orientados a objeto em termos de classes de objetos e das relações entre eles. A Figura 6.4 é um diagrama de classes simples da nossa loja virtual. Ele indica que o sistema é composto de três tipos de objetos, Cliente, Catálogo e Formulário Pedido, e das relações chamadas Pesquisando e Construindo.

A idéia subjacente é que um objeto do tipo Cliente contém os dados e os procedimentos específicos de um cliente; os objetos do tipo Catálogo, os dados e os procedimentos necessários para apresentar os produtos da companhia aos clientes; e um objeto do tipo Formulário Pedido, para manipular os itens que o cliente deseja comprar. A relação Pesquisando representa a associação entre os clientes e o catálogo e a relação Construindo, entre um cliente e seu formulário de pedidos. Os números que aparecem nas extremidades das linhas que representam a relação Construindo indicam que apenas um cliente pode ser associado a um formulário.

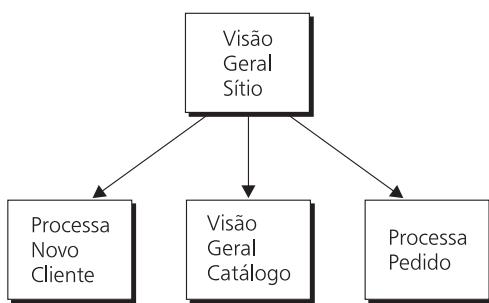


Figura 6.3 Um diagrama estrutural para uma loja virtual na Internet.

*N. de T. Em inglês, *structure chart*.

rio de pedidos e vice-versa. Essa relação um para um é diferente da relação muitos para um Pesquisando, na qual a notação indica que vários clientes podem pesquisar o catálogo simultaneamente. (Retornaremos a essas idéias quando discutirmos os diagramas de entidades e relacionamentos na Seção 6.5.)

Nos últimos anos, um progresso significativo ocorreu na padronização de sistemas de notação para representar os projetos orientados a objeto — o mais proeminente exemplo é a **UML (Unified Modeling Language)**, um sistema para representar diversos conceitos orientados a objeto. A notação usada na Figura 6.4 é baseada na convenção UML.

Acoplamento

Introduzimos o conceito de modularidade como forma de obtenção de um *software* administrável. A idéia baseia-se na alta probabilidade de somente alguns dos módulos do *software* terem necessidade de modificação posterior, de forma que podemos restringir as atenções apenas a essa parte do sistema durante o processo de alteração. Isto obviamente depende da validade da hipótese de que as alterações em um dado módulo não irão afetar inadvertidamente outros módulos do sistema. Em consequência, o objetivo principal do projeto de um sistema modular deveria ser a maximização da independência intermódulos. Um obstáculo a esta meta reside no fato de serem necessárias conexões entre os módulos para que seja criado um sistema coerente. Tal conexão é denominada **acoplamento**. Maximizar a independência entre os módulos corresponde, portanto, a minimizar seu acoplamento.

O acoplamento entre módulos ocorre, na verdade, de várias maneiras. Uma delas é o **acoplamento de controle**, que ocorre quando um módulo passa o controle a outro, como na relação de transferência e retorno entre um programa e sub-rotinas ou funções. Outro caso é o do **acoplamento de dados**, referente ao compartilhamento de dados entre módulos.

O diagrama estrutural da Figura 6.3 já continha uma representação do acoplamento de controle existente na abordagem de procedimento do nosso exemplo de loja virtual. O acoplamento de dados é tradicionalmente representado em um diagrama estrutural por meio de setas adicionais, conforme ilustra a Figura 6.5. Este quadro indica os elementos de dados que são passados para um módulo quando seus serviços são solicitados pela primeira vez, assim como os elementos de dados retornados ao módulo original quando a tarefa requisitada for completada. O diagrama indica que o módulo VisãoGeralSítio receberá informação sobre cada cliente a partir do procedimento ProcessaNovoCliente e passará essa informação ao procedimento VisãoGeralCatálogo. Além disso,

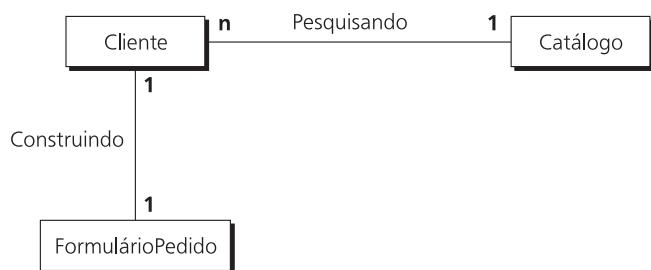


Figura 6.4 Um diagrama de classes para uma loja virtual na Internet.

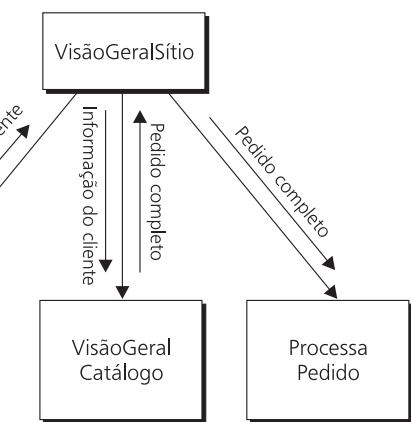


Figura 6.5 Um diagrama estrutural, que mostra o acoplamento de dados.

esse procedimento irá tratar o formulário de pedidos produzido, que será passado ao procedimento `ProcessaPedido`.

A minimização do acoplamento de dados é um dos principais benefícios da abordagem orientada a objeto. De fato, um dos conceitos motivadores dos objetos é a prática de confinar em um único módulo as rotinas que manipulam um item de dados específico. Assim, em um sistema orientado a objeto, a maioria dos acoplamentos entre módulos toma a forma da comunicação entre objetos, que normalmente é implementada como acoplamento de controle, isto é, a solicitação para que um objeto realize uma tarefa corresponde essencialmente a uma solicitação para a execução de um método (procedimento) dentro do objeto. A solicitação então resulta na passagem do controle para o objeto, de forma semelhante a uma chamada de procedimento no paradigma imperativo.

Uma forma de representar a comunicação entre objetos em um projeto é acrescentar a informação a um diagrama de classes para formar um **diagrama de colaboração** — que é essencialmente um diagrama de classes que também mostra como os vários objetos no sistema colaboram entre si. Um diagrama de colaboração simples (que usa notação compatível com a UML) para nossa loja virtual é mostrado na Figura 6.6. Ele indica que um objeto do tipo `Cliente` pode enviar uma mensagem a um objeto `FormulárioPedido`, dizendo-lhe para acrescentar um item no pedido, e que um objeto do tipo `Catálogo` pode enviar uma mensagem a um objeto `Cliente` com informações relativas a uma mercadoria específica.

Independentemente do tipo de acoplamento envolvido, convém evidenciá-lo na versão escrita do programa. Acoplamentos disfarçados são conhecidos como **acoplamentos implícitos**, e são responsáveis por muitos erros de software. Uma forma comum de acoplamento implícito decorre do uso de **dados globais** — elementos de dados que estão disponíveis automaticamente aos diversos módulos, por todo o sistema, enquanto os elementos locais de dados só são acessíveis internamente em um determinado módulo, a menos que sejam explicitamente passados para outro. A maioria das linguagens de alto nível dispõe de recursos para a implementação tanto de dados globais como locais. O problema dos dados globais é que as mudanças nos dados freqüentemente ocorrem como efeitos colaterais da execução de um módulo, em vez de serem explicitamente indicadas. (No vernáculo de software, o termo **efeito colateral** se refere a uma ação realizada pelo software que não é imediatamente aparente a partir do programa escrito.) Assim, uma pessoa que leia o programa pode não se dar conta de que um módulo altera uma unidade global, a não ser que os detalhes internos dos módulos sejam verificados. Isso degrada a utilidade dos módulos como ferramentas abstratas. (Aqui está um exemplo de onde os comentários devem ser empregados para ajudar o leitor a entender um programa.)

Coesão

Assim como é importante minimizar o acoplamento entre módulos, maximizar a ligação interna em cada módulo também é. O termo **coesão** foi adotado para designar essa ligação interna, o grau de relacionamento entre as partes internas de um módulo. Para compreender a importância da coesão, devemos olhar para além do desenvolvimento inicial do sistema e considerar integralmente o ciclo de

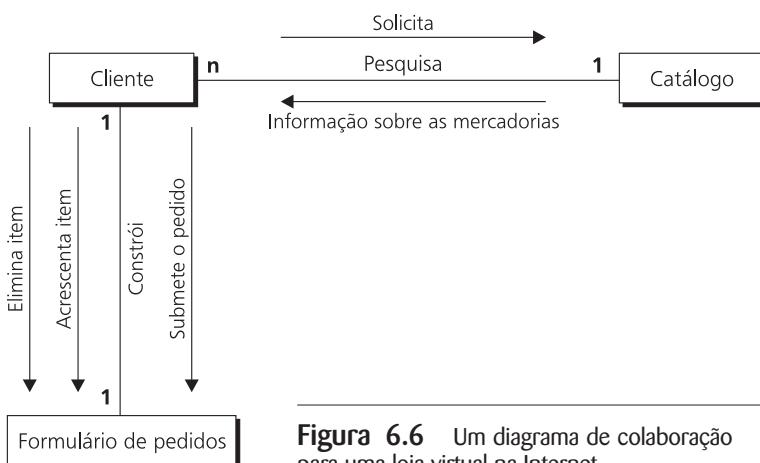


Figura 6.6 Um diagrama de colaboração para uma loja virtual na Internet.

vida do *software*. Se for necessário efetuar modificações em um módulo, a existência de uma diversidade de ações internas complicará o processo, que, de outra forma, seria simples. Assim, além de ter como meta manter um baixo acoplamento entre módulos, os projetistas de *software* se esforçam para obter uma forte coesão interna nos mesmos.

Uma forma fraca de coesão é conhecida como **coesão lógica**. Trata-se da coesão interna de um módulo, pois, em termos lógicos, os elementos que o constituem executam ações similares. Por exemplo, consideremos um módulo que execute toda a comunicação de um sistema com o mundo externo. O que mantém a união de tal módulo é o fato de todas as suas ações internas se referirem à comunicação. Entretanto, os assuntos tratados na comunicação podem variar amplamente. Enquanto alguns se dedicam à obtenção de dados, outros se reportam à ocorrência de erros.

Uma forma mais forte de coesão é conhecida como **coesão funcional**, ou seja, todas as partes do módulo estão voltadas para o desempenho de uma mesma atividade. O módulo VisãoGeralCatálogo na Figura 6.3 não possui coesão funcional, pois contém os detalhes da comunicação com o cliente, obtendo e exibindo informação de produtos e construindo o pedido do cliente. Contudo, se esses detalhes estiverem isolados em outros módulos e forem usados como ferramentas abstratas, cada passo no módulo VisãoGeralCatálogo poderá ser apresentado no contexto geral de supervisionar as ações associadas com o processo de folhear o catálogo, aumentando assim a coesão funcional do módulo.

Nos projetos orientados a objeto, os objetos geralmente possuem apenas a coesão lógica, uma vez que seus métodos freqüentemente realizam atividades fracamente relacionadas — a única vinculação comum é o fato de que as atividades são realizadas no mesmo objeto. Por exemplo, em nossa loja virtual, cada objeto do formulário de pedidos provavelmente conterá um método para acrescentar um item ao pedido, outro para eliminar um item e outro para submeter o pedido. Portanto, tal objeto será um módulo apenas com coesão lógica. No entanto, o projetista de *software* deve esforçar-se para que cada método de um objeto tenha coesão funcional, isto é, ainda que o objeto como um todo possua apenas a coesão lógica, cada método deve realizar apenas uma tarefa, de modo a possuir coesão funcional (Figura 6.7).

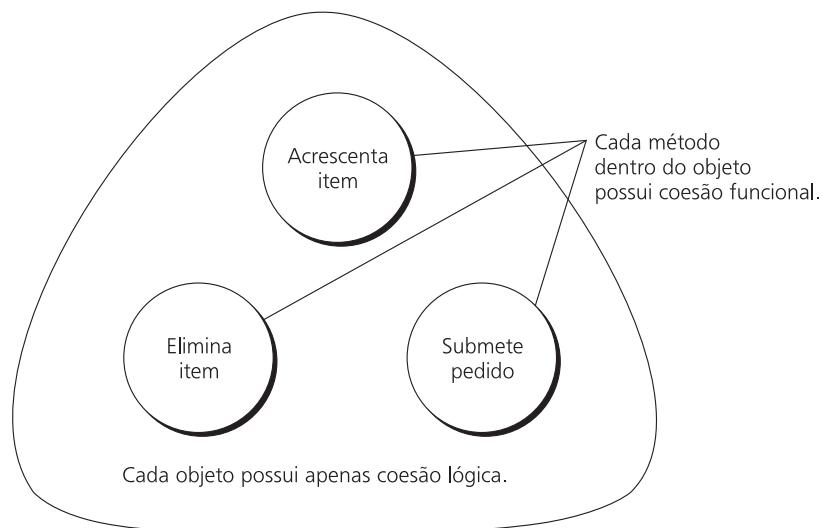


Figura 6.7 Coesão lógica e funcional dentro de um objeto que representa um formulário de pedidos em uma loja virtual na Internet.



QUESTÕES/EXERCÍCIOS

1. Como um romance difere de uma enclopédia, em termos de grau de acoplamento entre suas unidades (como capítulos, seções ou introduções)? E quanto à coesão?
2. Uma partida de um jogo de cartas é dividida em duas fases: os lances e o jogo de cartas propriamente dito. Analise o acoplamento entre estas fases, identificando a informação que é passada explicitamente da primeira fase à segunda. Qual informação é passada implicitamente?
3. O objetivo de maximizar a coesão é compatível com o de minimizar o acoplamento? Em outros termos, à medida que aumenta a coesão, o acoplamento tende naturalmente a diminuir?
4. Estenda o diagrama de colaboração na Figura 6.6 para incluir outras mensagens que devem ser passadas entre os objetos do tipo `Cliente` e `Catálogo`.

6.4 Metodologias de projeto

O desenvolvimento de metodologias para projetar sistemas de *software* é uma investigação fundamental na engenharia de *software*. Já encontramos alguns tópicos nessa área, incluindo o modelo da cachoeira e a prototipação. Nesta seção, exploraremos alguns tópicos adicionais, bem como algumas diretrizes da pesquisa corrente.

Cima-baixo versus baixo-cima

Talvez a estratégia mais comum associada ao projeto de sistemas seja a metodologia cima-baixo*, cuja essência é que não se deve tentar resolver um problema complexo em um único passo. Em vez disso, deve-se começar dividindo o problema em subproblemas menores e mais manejáveis. Então, deve-se prosseguir dividindo esses subproblemas em outros ainda menores. Dessa maneira, um problema complexo torna-se um conjunto de problemas mais simples, cujas soluções conjuntas resolvem o problema original.

O resultado de um projeto cima-baixo tende a ser um sistema hierárquico de refinamentos que muitas vezes pode ser traduzido diretamente em uma estrutura modular, compatível com o paradigma imperativo de programação. As soluções desses pequenos problemas na hierarquia se tornam módulos procedimentais que realizam tarefas simples e são usados como ferramentas abstratas pelos módulos superiores para resolver os problemas mais complexos no sistema.

Em contraste à metodologia de projeto cima-baixo, está a abordagem baixo-cima**, na qual se inicia o projeto de um sistema identificando tarefas individuais no sistema e então considerando como as soluções dessas tarefas podem ser usadas como ferramentas abstratas na solução de problemas mais complexos. Por muitos anos, este método foi considerado inferior ao do para-

O ambiente de programação Java

A linguagem de programação Java seria apenas mais uma linguagem orientada a objeto, não fosse por seus recursos, como a coleção de armações, que tem sido desenvolvida para facilitar o processo de programação. Essas armações são chamadas coletivamente Interface de Programas de Aplicação e estão disponíveis como parte do Kit de Desenvolvimento Java da Sun Microsystems. As armações fornecem gabaritos para o desenvolvimento de GUIs, manipulação de áudio e vídeo, transferência de dados pela Internet, desenvolvimento de poderosos servidores e de páginas da Web com animação. Assim, o ambiente de programação Java provê um exemplo do movimento para a construção de *software* a partir de componentes pré-fabricados.

*N. de T. Em inglês, *top-down*.

**N. de T. Em inglês, *bottom-up*.

digma cima-baixo. Atualmente, porém, a metodologia de projeto baixo-cima está ganhando apoio. Uma razão para esta mudança de opinião é que a metodologia cima-baixo busca uma solução na qual um módulo dominante usa submódulos, e cada um destes se sustenta em sub-submódulos e assim por diante. Contudo, o melhor projeto para muitos sistemas não possui natureza hierárquica. De fato, um projeto que consiste em dois ou mais módulos em interação de igual para igual, como exemplificado no modelo cliente-servidor, ou sistemas que envolvem aplicações de processamento paralelo podem ser melhores soluções que um sistema que consiste em um módulo superior que se apóia em subordinados para realizar a sua tarefa.

Outra razão para o interesse crescente no projeto baixo-cima é que ele é mais consistente com a meta de construir sistemas de *software* complexos a partir de componentes pré-fabricados — uma abordagem que é a tendência atual na engenharia de *software*.

Padrões de projeto

Em um esforço para encontrar os modos de construir *software* a partir de componentes da prateleira, os engenheiros de *software* têm se voltado para o campo da arquitetura em busca de inspiração. Especialmente interessante é o livro *A Pattern Language*, de Christopher Alexander et al., que descreve um conjunto de padrões para projetar comunidades. Cada padrão consiste no enunciado de um problema genérico seguido por uma solução proposta. Os problemas pretendem ser universais e as soluções são genéricas, no sentido de buscarem a natureza universal do problema, em vez de resolver um caso particular.

Por exemplo, um padrão chamado Fundos Tranquilos (*Quiet Backs*) aborda a necessidade de escape da confusão de um centro comercial para pequenos períodos de relaxamento. A solução proposta é projetar “fundos tranquilos” nos distritos comerciais. Em alguns casos, o distrito pode ser projetado ao longo de uma avenida na qual se situam todos os prédios — o que proporciona ruas tranquilas atrás dos prédios. Em outros casos, os “fundos tranquilos” podem ser obtidos por meio de parques, rios ou catedrais.

O ponto importante para a nossa discussão é que o trabalho de Alexander tenta identificar problemas universais e propor padrões para resolvê-los. Atualmente, muitos engenheiros de *software* vêm tentando aplicar a mesma abordagem no projeto de grandes sistemas. Em particular, os pesquisadores estão aplicando padrões de projeto como uma maneira de prover blocos genéricos com os quais os sistemas de *software* podem ser construídos.

Um exemplo de padrão é o editor-assinante, que consiste em um módulo (o editor) que deve enviar cópias de suas “publicações” para outros módulos (os assinantes) (Figura 6.8). Como um exemplo específico, considere um conjunto de dados que deve ser exibido simultaneamente na tela de um computador em mais de um formato — talvez como gráfico circular ou um gráfico de barras. Nesse contexto, qualquer mudança nos dados deve aparecer nos dois gráficos. Assim, o módulo de *software* encarregado de desenhar os gráficos deve ser notificado quando ocorrem mudanças nos dados. Nesse caso, o módulo de *software* que mantém os dados faz o papel do editor, que deve enviar mensagens de atualização aos assinantes, que são os módulos encarregados de desenhar os gráficos.

Outro exemplo de padrão de projeto de *software* é o padrão recipiente-componente. Ele capta o conceito geral de um recipiente que contém componentes que, por sua vez, podem ser recipientes (Figura 6.9). Tal padrão é exemplificado pelos diretórios ou pastas usados pelo gerente de arquivos de

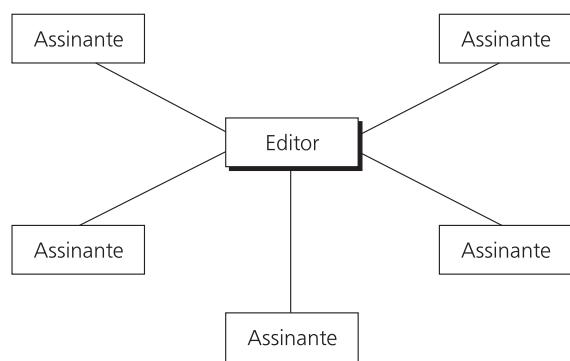


Figura 6.8 O padrão editor-assinante.

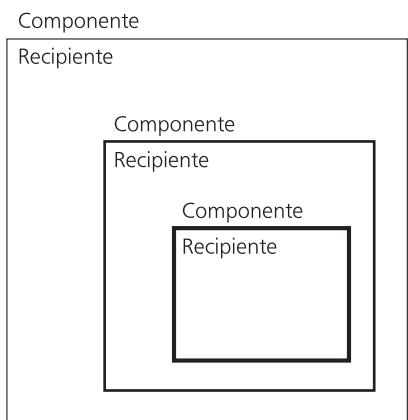


Figura 6.9 O padrão recipiente-componente.

um sistema operacional. Cada diretório normalmente contém outros diretórios, os quais podem conter ainda outros. Em resumo, o padrão recipiente-componente pretende captar o conceito recursivo de recipientes que contêm recipientes.

Uma vez identificado um padrão como o do editor-assinante ou do recipiente-componente, os engenheiros de *software* propõem o desenvolvimento de unidades de programa esquemáticas chamadas **armações** (*frameworks*) que implementam os recursos pertinentes para a solução do padrão, deixando as características específicas das aplicações particulares como encaixes a serem preenchidos mais tarde. Para acompanhar as armações, os engenheiros propõem uma documentação que descreve como os encaixes devem ser preenchidos para obter uma implementação completa do padrão subjacente no contexto específico. Essa documentação algumas vezes é chamada receita. As coleções de documentações com suas receitas são conhecidas carinhosamente por alguns como *livros de culinária*. Os pesquisadores acreditam que por meio desses livros os engenheiros de *software* finalmente estarão aptos a construir sistemas grandes e complexos a partir de componentes de prateleira — componentes esses que são as armações.

Embora os padrões de projeto tenham gerado grande entusiasmo na comunidade de engenharia de *software*, é interessante notar que Alexander não ficou satisfeito com o resultado de seus padrões na arquitetura. Ele considerou que os sistemas projetados a partir dos padrões perdião o caráter e seu trabalho, desde o início dos anos 1980, tem focado as maneiras de capturar essa qualidade evasiva. Entretanto, os engenheiros de *software* argumentam que a meta no desenvolvimento de *software* não envolve qualidades como beleza e caráter, mas correção e eficiência. Assim, os padrões de projeto terão mais sucesso na área da engenharia de *software* do que na arquitetura. De fato, como testemunha do sucesso na abordagem dos padrões de projeto, a popularidade das armações na engenharia de *software* continua a crescer.

Desenvolvimento de código aberto

Algumas vezes, os avanços significativos nas culturas se desenvolvem fora do eixo principal, onde se tornam doutrinas bem definidas antes de ser finalmente reconhecidos como contribuições importantes pela comunidade formalmente estabelecida. Por exemplo, o *jazz* atualmente é reconhecido como uma contribuição valiosa ao mundo da música; grande parte da arte moderna foi originalmente desdenhada pelos críticos, e algum dia o golfe pode se tornar um esporte olímpico.

Esse é o caso de uma técnica um tanto improvisada para desenvolvimento de *software* que vem sendo usada há anos por entusiastas da computação e que agora está sendo reconhecida como metodologia legítima de desenvolvimento de *software*. Na verdade, a técnica é uma forma de prototipação evolucionária, mas ao contrário da abordagem tradicional, a prototipação é realizada em um fórum público. A metodologia é chamada **desenvolvimento de código aberto** (*open-source development*), embora o nome mais adequado fosse “prototipação evolucionária pública”.

Em um certo sentido, o desenvolvimento de código aberto é uma extensão do teste beta (Seção 6.6) — mas ele permite que os “testadores beta” modifiquem o *software* e comuniquem essas modificações, em vez de meramente comunicar problemas, como no teste beta tradicional. O processo completo é o seguinte: o autor original escreve uma versão inicial do *software* (provavelmente para suprir as suas necessidades) e divulga o código-fonte e sua documentação pela Internet, de onde pode ser baixado e usado por outros. Uma vez que esses outros usuários possuem o programa-fonte e a documentação, estão aptos a modificar ou melhorar o *software* para ajustá-lo às suas necessidades e corrigir erros que encontram. Eles comunicam essas mudanças ao autor original, o qual as incorpora à versão divulgada do *software*. Essa versão estendida estará, portanto, disponível para futuras modifi-

cações. Na prática, é possível que um pacote de *software* se desenvolva por meio de várias extensões em uma única semana.

Como os participantes do desenvolvimento de código aberto não estão organizados e não recebem remuneração, pode-se pensar que não se dedicam o bastante para produzir sistemas grandes e com alto nível de qualidade. Contudo, esta conjectura tem demonstrado ser falsa. O sistema operacional Linux, reconhecido como um dos sistemas operacionais mais confiáveis disponíveis atualmente, constitui um importante exemplo. De fato, Linus Torvald, o criador original do Linux, pode ser considerado o pai do desenvolvimento de código aberto, pois foi com essa metodologia que ele dirigiu o desenvolvimento inicial do Linux.

Um problema que impede o desenvolvimento de código aberto de se tornar uma metodologia mais popular de desenvolvimento de *software* é a dificuldade de uma única entidade manter a propriedade do produto final. De fato, exercer o desenvolvimento de código aberto é liberar o *software* para o domínio público.

Outro problema é a dificuldade de os gerentes treinados na comunidade de negócios aceitarem o estilo “livre” que é aparentemente exigido para que haja sucesso no desenvolvimento do código aberto. Tal desenvolvimento parece apoiar-se em um ambiente no qual o entusiasmo dos indivíduos é fomentado pela expressão de suas habilidades criativas e pelo sentimento de realização e orgulho. Os gerentes, porém, têm mostrado uma tendência para exercer um nível de controle que atenua esse entusiasmo. Por exemplo, um gerente pode querer reter parte do sistema de *software* para garantir a propriedade, mas o desenvolvimento de código aberto se apóia justamente no fornecimento do sistema completo para um indivíduo baixar, usar e eventualmente modificar de acordo com suas necessidades. Em outros casos, um gerente pode atrasar a liberação de novas versões do *software* para evitar erros embaraçosos, quando na realidade a descoberta desses erros é que faz aumentar o apetite de muitos contribuidores ao desenvolvimento de código aberto.

Independentemente de o desenvolvimento de código aberto tornar-se um modelo popular para o desenvolvimento de *software* comercial, ele deve ser reconhecido como uma metodologia existente e viável que merece a atenção dos pesquisadores desse campo da engenharia de *software*. Sem dúvida, o aprendizado sobre o que faz o desenvolvimento de código aberto funcionar em algumas situações, ainda que falhe em outras, deve levar às investigações que contribuem para o campo como um todo.



QUESTÕES/EXERCÍCIOS

1. Identifique algumas estruturas de *software* discutidas nos capítulos anteriores que poderiam ser consideradas padrões de projeto.
2. Que papel no processo da engenharia de *software* os pesquisadores esperam que as armações desempenhem?
3. Qual é a distinção entre a prototipação evolucionária tradicional e o desenvolvimento de código aberto?

6.5 Ferramentas do ofício

A Engenharia de Software produziu diversos sistemas notacionais para auxiliar na análise e projeto de sistemas. Desses, já vimos os diagramas estruturais, de classes e de colaboração. Um outro é o **diagrama de fluxo de dados**, representação gráfica das trajetórias de dados em um sistema, isto é, ele identifica a origem, o destino e os pontos de processamento dos dados em um sistema. Os vários símbolos existentes em um diagrama possuem significados específicos: as setas representam as trajetórias dos dados; os retângulos, as fontes e os sorvedouros de dados; os círculos (bolhas), os pontos de manipulação de dados; e as linhas retas grossas, o armazenamento de dados. Em cada caso, o símbolo é

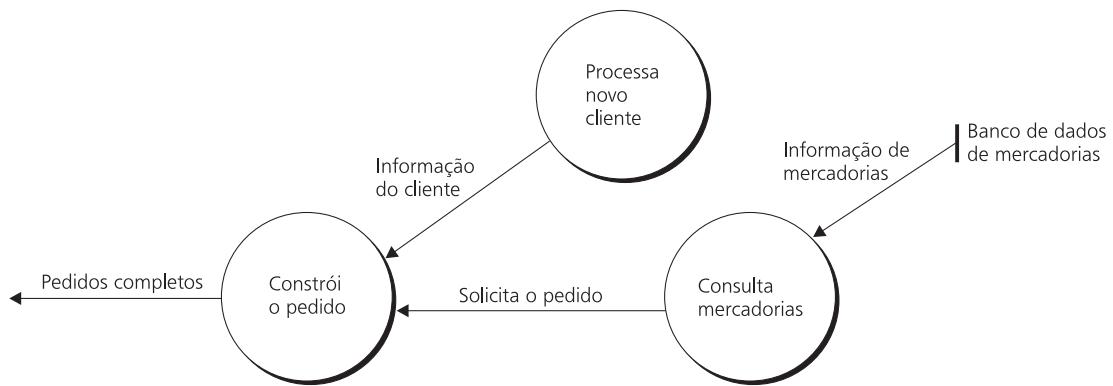


Figura 6.10 Um diagrama de fluxo de dados para uma loja virtual na Internet.

rotulado com o nome da entidade representada. Um diagrama de fluxo de dados para a nossa loja virtual é mostrado na Figura 6.10.

Os diagramas de fluxo de dados para o desenvolvimento de um *software* têm sua origem no paradigma imperativo de programação. A idéia baseia-se no fato de que, ao seguir as trajetórias dos dados ao longo do sistema proposto, vão-se descobrindo os pontos em que os elementos de dados se fundem, se separam ou são alterados. Como é necessária nessas posições do sistema a execução de ações computacionais, tais ações, ou conjuntos de ações, vão formando os módulos do sistema. Por conseguinte, é possível obter uma estrutura modular para um sistema com fluxos de dados.

Embora desenvolvida de acordo com o paradigma de programação imperativo, a análise do fluxo de dados encontrou aplicações em ambientes orientados a objeto. Em particular, a identificação de itens de dados em um sistema ajuda a identificar os objetos, e a identificação das mudanças feitas nos dados, a identificar as ações que esses objetos devem realizar.

Outra ferramenta usada na análise e projeto de sistemas de *software* é o **diagrama de entidades e relacionamentos**, uma representação gráfica dos elementos de informação (entidades), interiores do sistema e de suas relações. Para ilustrar, consideremos uma parte de um diagrama de entidades e relacionamentos para um sistema de *software* que mantenha informações sobre professores, estudantes e classes em uma universidade.

Primeiro, identifiquemos as entidades de dados envolvidas. Elas incluem a entidade Professor, que representa um único professor da universidade; a entidade Estudante, que representa um único estudante; e a entidade Classe, que representa uma seção de um determinado curso. A cada ocorrência da entidade Professor, é associado um nome, endereço, número de identificação do funcionário, salário e assim por diante; a cada ocorrência da entidade Estudante, um nome, endereço, número de identificação do estudante, média das notas e assim por diante; e a cada ocorrência da entidade Classe, é associada uma identificação da disciplina (História 101), semestre e ano, sala de aula, horário etc.

Identificadas as entidades em nosso sistema, consideremos agora as relações entre elas. Primeiro note-se que cada professor leciona e cada estudante freqüenta as aulas. Identifiquemos então a relação entre as entidades Professor e Classe como a relação Leciona e a existente entre Estudante e Classe como a relação Freqüenta. (Note-se que as entidades são associadas a substantivos, enquanto as relações, a verbos.)

Para representar tais entidades e relações, utilizemos o diagrama de entidades e relacionamentos da Figura 6.11. Aqui, cada entidade é representada por um retângulo e cada relação, por um losango. O



Figura 6.11 Um diagrama de entidades e relacionamentos.

diagrama mostra nitidamente que os professores estão relacionados às classes por meio da relação Leciona e os estudantes, pela relação Freqüenta.

Entretanto, há uma estrutura diferente associada a cada relação do nosso exemplo. A relação entre Professor e Classe é de um para muitos, em que cada professor leciona em várias classes, mas cada classe recebe ensinamentos de um único professor. Ao contrário, a relação entre Estudante e Classe é de muitos para muitos, porque cada estudante freqüenta várias aulas e cada aula é freqüentada por vários estudantes. Assim, existem três tipos fundamentais de relacionamento que podem ocorrer entre entidades — um para um, um para muitos (ou muitos para um, dependendo do ponto de vista), e muitos para muitos, como mostrado na Figura 6.12. Encontramos um relacionamento um para um na Figura 6.6, na qual cada Cliente era associado a um único Formulário de Pedido e cada formulário, a um cliente. Outros exemplos de relacionamento de um para muitos e muitos para muitos existem entre autores e livros. O relacionamento tradicional entre autores e romances é de um para muitos, já que um romancista pode escrever vários romances, mas cada um possui um só autor. Entretanto, o relacionamento entre autores de livro-texto e este é muitos para muitos, porque um único livro-texto freqüentemente é de muitos autores.

Em um diagrama de entidades e relacionamentos, o tipo de relacionamento freqüentemente é representado pela presença de ponteiros nas linhas que conectam as relações entre entidades. Um único ponteiro para uma entidade significa que esta acontece somente uma vez a cada ocorrência da relação, enquanto um ponteiro duplo indica que mais de uma ocorrência pode estar envolvida. Assim, o ponteiro para a entidade Professor da Figura 6.11 indica que apenas um professor leciona para uma classe,

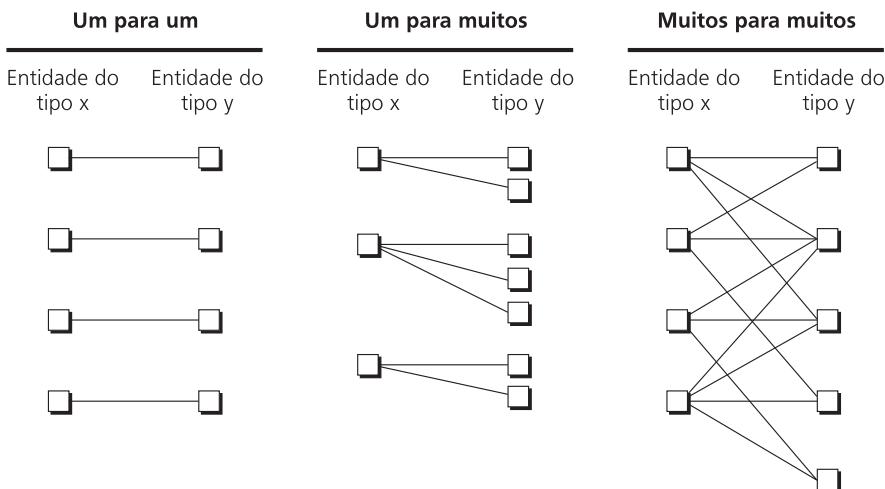


Figura 6.12 Relacionamentos um para um, um para muitos e muitos para muitos entre entidades dos tipos X e Y.

enquanto o ponteiro duplo para `Classe` da relação `Leciona` indica que cada professor pode ensinar mais de uma classe.

Dentre as várias ferramentas utilizadas pelos engenheiros de *software* no passado, os diagramas de entidades e relacionamentos provavelmente serão das poucas que sobreviverão à transição para as metodologias orientadas a objeto. Isto se deve ao fato de que a identificação de entidades corresponde essencialmente à identificação de objetos e de a classificação de seus relacionamentos ser o primeiro passo para a identificação das relações e das trajetórias de comunicação necessárias entre objetos. De fato, o diagrama tradicional de entidades e relacionamentos é prontamente identificado como o predecessor do diagrama de classes UML (Figura 6.4) usado no ambiente de projeto orientado a objeto.

Outra ferramenta para o desenvolvimento de um sistema de *software* é o **dicionário de dados** — um repositório central de informações sobre os dados que aparecem no sistema. Essas informações incluem o identificador associado a cada item, a discriminação de quais dados constituem elementos válidos (por exemplo, se o dado sempre é numérico ou alfabético, ou qual faixa de valores pode ser associada a este dado), a posição em que ficam armazenados os dados (por exemplo, em qual arquivo ou banco de dados o dado foi armazenado) e os pontos do *software* em que o item é referenciado (por exemplo, quais módulos fazem referência a esse dado).

Há várias metas associadas ao desenvolvimento de um dicionário de dados. Uma delas é aumentar a comunicação entre o potencial usuário do sistema e o analista encarregado de transformar suas necessidades em requisitos e especificações. Seria descorajador descobrir, após a implementação de um sistema, que a identificação de uma peça (número de ordem) não é de fato numérica, ou que o tamanho do cadastro real excede o máximo permitido pelo sistema. A construção de um dicionário de dados evita tais enganos.

Outra meta associada ao dicionário de dados é o estabelecimento de uma uniformidade ao longo do sistema. Normalmente, por meio da construção de dicionários de dados, afloram diversas redundâncias e contradições. Por exemplo, o dado chamado `NúmeroDaPeça` dos registros de cadastro pode ser o mesmo que foi chamado `IdPeça` nos registros de venda. Além disso, o departamento de pessoal pode usar o item `Nome` para se referir a um funcionário, enquanto os registros de cadastro podem conter o item `Nome` associado a alguma peça.

Finalmente, devemos mencionar as fichas CRC (*class-responsibility-collaboration*) que são úteis no projeto de sistemas orientados a objeto. Uma ficha CRC é essencialmente uma ficha tradicional de indexação, na qual é escrita a descrição de um objeto. A metodologia das fichas CRC faz com que o projetista produza uma ficha para cada objeto de um sistema proposto e então use as fichas para simular as atividades do sistema — talvez em sua mesa de trabalho ou via experimento “teatral”, onde cada membro da equipe de projetistas faz o papel de um objeto como descrito na ficha apropriada. Tais simulações têm se mostrado úteis na identificação de falhas no projeto antes de sua implementação.



QUESTÕES/EXERCÍCIOS

1. Suponha que uma recepcionista receba solicitações para agendar compromissos. A resposta a tais solicitações ou é um horário para uma futura reunião ou, então, uma reunião imediata. Faça um diagrama de fluxo de dados que represente esse trabalho da recepcionista.
2. Faça um diagrama de entidades e relacionamentos que represente companhias aéreas, os vôos que cada uma realiza e os passageiros nesses vários vôos.
3. Qual é a diferença entre implementar um relacionamento um para um e um relacionamento de um para muitos?

6.6 Testes

Na Seção 4.6, consideramos as técnicas para verificação e correção de algoritmos de maneira rigorosamente matemática, mas concluímos que a maioria dos softwares atualmente é “verificada” por meio de testes. Infelizmente, os testes são no máximo uma ciência inexata. Não podemos declarar que um módulo de software está correto a partir deles, a menos que façamos testes suficientes para exaurir todos os cenários possíveis. Existem mais de 1.000 caminhos diferentes em uma simples estrutura de laço que contém uma instrução `se-então-senão` e podem ser percorridos quando o laço é repetido 10 vezes. Assim, o teste de todos os caminhos possíveis em um programa complexo é uma tarefa impossível.

No entanto, os engenheiros de software têm desenvolvido metodologias de teste que ajudam a revelar os erros no software. Uma delas se baseia na observação de que os erros no software tendem a se agrupar. Isto é, a experiência tem mostrado que um pequeno número de módulos em um grande sistema de software tende a ser mais problemático do que os restantes. Assim, uma vez identificados esses módulos e testados exaustivamente, a maioria dos erros no sistema pode ser descoberta sem precisar testar todos os módulos de uma maneira uniforme e menos exaustiva. Essa é uma instância de uma proposição conhecida como **princípio de Pareto**, em referência ao economista e sociólogo Vilfredo Pareto (1848-1923) que observou que uma pequena parte da população da Itália controlava a maioria das riquezas do país. Em geral, o princípio de Pareto declara que os resultados freqüentemente tendem a ser alcançados mais rapidamente pela aplicação concentrada de esforços em uma área.

Outra metodologia de teste para software, chamada **teste do caminho base** (*basis path testing*) é desenvolver um conjunto de dados de teste que garanta a execução de cada instrução no software pelo menos uma vez. As técnicas, que usam uma área da matemática conhecida como teoria dos grafos, têm sido desenvolvidas para identificar esses conjuntos de dados de teste. Assim, embora seja impossível garantir que qualquer caminho em um sistema de software seja testado, é possível garantir que cada instrução no sistema seja executada pelo menos uma vez durante o processo de teste.

As técnicas baseadas no princípio de Pareto e no teste do caminho base se apóiam no conhecimento da composição interna do software a ser testado. Portanto, elas se enquadram na categoria **teste na caixa de vidro** — o que significa que o interior do software é visível ao testador. Em contraste, está a categoria **teste na caixa preta**, que se refere a testes que não se apóiam no conhecimento da composição interna do software. Resumindo, o teste na caixa preta é realizado a partir do ponto de vista do usuário. Não se investiga como o software desempenha a sua tarefa, mas apenas se ele a realiza corretamente em termos de exatidão e rapidez.

Uma metodologia que freqüentemente é associada com o teste na caixa preta, chamada análise de valores de fronteira, é identificar os pontos de fronteira nas especificações do software e testá-lo nesses pontos extremos. Por exemplo, se o software deve aceitar valores de entrada em uma faixa especificada, então ele deve ser testado com o menor e o maior valor da faixa, ou então, se ele se propõe a coordenar múltiplas atividades, deve ser testado com uma coleção de atividades que implique uma maior demanda.

Outra metodologia de teste na caixa preta é aplicar a redundância. Seguindo essa abordagem, dois sistemas de software que realizam a mesma tarefa são desenvolvidos independentemente por equipes diferentes ou mesmo por diferentes empresas. Então os dois sistemas são testados aplicando-se a eles o mesmo conjunto de dados e comparando os seus resultados. Os erros serão indicados pelas discrepâncias. Essas técnicas de redundância freqüentemente são aplicadas nos sistemas de exploração espacial.

Outra metodologia que se enquadra na categoria da caixa preta e vem sendo usada de maneira crescente pelos desenvolvedores de software “encolher-embrulhar” voltado para o mercado de PCs é fornecer a um segmento do público pretendido uma versão preliminar, chamada **versão beta**. A meta principal é aprender como o software se comporta em situações da vida real antes que a versão final do produto se solidifique e seja liberada ao mercado.

As vantagens desse **teste beta** se estendem além da tradicional descoberta de erros. O retorno dos clientes em potencial (seja positivo ou negativo) é obtido e pode ajudar a refinar as estratégias de venda. Além disso, a distribuição antecipada do software beta ajuda outros desenvolvedores de software

a projetar pacotes compatíveis. Por exemplo, no caso de um novo sistema operacional, a distribuição de uma versão beta estimula o desenvolvimento de programas utilitários compatíveis, de forma que o sistema operacional final aparecerá nas prateleiras das lojas cercado de produtos relacionados. Finalmente, a existência do *software* beta pode gerar um sentimento de antecipação no mercado — uma atmosfera que aumenta a publicidade e as vendas.



QUESTÕES/EXERCÍCIOS

1. Quando se testa um *software*, o teste é bem-sucedido quando se encontram erros ou quando não se encontram?
2. Que técnicas você propõe usar para a identificação dos módulos em um sistema que devem ser testados mais exaustivamente?
3. Qual seria um bom teste a ser realizado em um pacote de *software* projetado para ordenar uma lista com 100 elementos no máximo?

6.7 Documentação

Um sistema de *software* tem pouca utilidade, a menos que as pessoas possam entendê-lo e mantê-lo. Conseqüentemente, a documentação se mostra uma parte importante de um pacote de *software*, logo o seu desenvolvimento é um tópico não menos importante da engenharia de *software*.

Em geral, a documentação de um pacote de *software* é elaborada com dupla finalidade. A primeira destina-se a explicar as características do *software* e a descrever como usá-lo. Esta é conhecida como a **documentação de usuário**, pois é projetada para ser lida pelo usuário do *software*. Conseqüentemente, a documentação de usuário tende a ser não-técnica.

Atualmente, a documentação de usuário é reconhecida como importante ferramenta de *marketing*. Uma boa documentação de usuário (combinada com uma interface de usuário bem projetada) torna acessível o pacote de *software* e gera, portanto, um aumento em sua vendagem. Reconhecido este ponto, muitos fornecedores de *software* contratam escritores técnicos para trabalhar este lado dos seus produtos, ou fornecem versões preliminares dos mesmos para escritores independentes de manuais, de forma que manuais didáticos, que ensinam o funcionamento do *software*, estejam disponíveis nas livrarias na época do seu lançamento no mercado.

A documentação de usuário é tradicionalmente elaborada na forma de um livro ou manual, mas em muitos casos, a mesma informação é incluída no próprio *software*. Isto permite ao usuário consultar partes do manual através do monitor, enquanto continua a processar o *software*. Neste caso, a informação pode ser dividida em pequenos módulos, às vezes chamados de pacotes de ajuda*, que são apresentados como partes do próprio sistema de *software*. Isto significa que o recurso de acesso a um dos pacotes de ajuda é uma opção construída como parte do sistema de *software*. Em alguns sistemas, os pacotes de ajuda podem entrar automaticamente em atividade na tela do monitor, caso o usuário dê, por muito tempo, a impressão de estar perdido entre os comandos do sistema.

O outro propósito da documentação é o de descrever a composição interna do *software* de forma que, mais tarde, seja possível efetuar a sua manutenção, durante o seu ciclo de vida. Conhecida como **documentação de sistema**, é inherentemente mais técnica do que a documentação de usuário. O componente principal da documentação é a versão fonte de todos os programas do sistema. É essencial

*N. de T. Em inglês, *help*.

que tais programas sejam apresentados em um formato legível, motivo pelo qual os engenheiros de software apóiam o uso de linguagens de programação de alto nível bem projetadas, o emprego de comentários para documentar o programa e um projeto modular, que apresente de forma coerente cada módulo do sistema. De fato, muitas companhias que produzem software têm adotado convenções que seus funcionários devem seguir quando escrevem os programas. Elas incluem indentação para organizar o programa na página, convenção de nomenclatura para distinguir nomes de variáveis, constantes, objetos, classes etc. e convenção de documentação, para assegurar que todos os programas sejam suficientemente documentados. Essas convenções garantem uniformidade ao software da companhia, o que em última análise simplifica o processo de manutenção.

Outro componente da documentação de um sistema é o registro dos documentos de projeto que descrevem as especificações do sistema e como elas foram obtidas. A criação dessa documentação é um processo contínuo, que começa na análise inicial, durante o desenvolvimento do software e, por isso, leva a um conflito entre as metas da engenharia de software e a natureza humana. É altamente improvável que as especificações iniciais e o projeto inicial do software permaneçam inalterados à medida que o seu desenvolvimento evoluí. Em casos deste tipo, é uma tentação fazer alterações contínuas no projeto do sistema sem atualizar sua antiga documentação. O resultado é uma grande possibilidade de tais documentos ficarem incorretos, inutilizando, consequentemente, a documentação final.

Nisto se fundamenta outro argumento em favor das ferramentas CASE, que refazem os diagramas e atualizam dicionários com mais facilidade do que mediante os antigos métodos manuais. Desta maneira, aumenta-se a probabilidade de as atualizações serem feitas corretamente e também é mais provável que a documentação final se torne mais precisa.

Finalizaremos enfatizando que o exemplo da atualização de documentos é apenas uma das muitas instâncias em que a engenharia de software se vê obrigada a conciliar os fatos frios e difíceis de uma ciência com a compreensão realística da natureza humana. Outras incluem os inevitáveis conflitos de personalidade, ciúmes e explosões de ego que surgem em trabalhos de equipe. Assim, como mencionamos anteriormente, o assunto da engenharia de software é muito mais abrangente, não se restringindo apenas a esses assuntos diretamente associados à Ciência da Computação.



QUESTÕES/EXERCÍCIOS

1. Quais as maneiras de se documentar um software?
2. Em que fase (ou fases) do ciclo de vida do software é preparada a documentação?
3. O que é mais importante, um programa ou a sua documentação?

6.8 Propriedade e responsabilidade de software

As companhias e as pessoas físicas têm reivindicado uma forma de recuperar o enorme investimento aplicado no desenvolvimento de softwares de alta qualidade e auferir lucros com eles, já que, por não haver meios de proteção de tal investimento, poucos estão dispostos a produzir o software de que a comunidade necessita. No entanto, questões acerca da propriedade de software e dos direitos de propriedade em geral recaem nas brechas das leis bem estabelecidas de direitos autorais e de patentes. Embora tais leis tenham sido desenvolvidas para permitir a quem desenvolve um “produto” liberar esse produto ao público salvaguardando seus direitos de propriedade, as características do software têm desafiado continuamente os tribunais em seus esforços de aplicar os princípios de direitos autorais e de patentes a assuntos ligados à propriedade de software.

O Therac-25

A necessidade de boas disciplinas de projeto é exemplificada pelos problemas encontrados no Therac-25, que era um sistema de terapia por aceleração/radiação eletrônica controlado por computador, usado pela comunidade médica na década de 1980. Falhas no projeto da máquina contribuíram para seis casos de *overdose* de radiação – três dos quais resultaram em morte. Essas falhas incluíam um projeto precário para a interface da máquina, que permitia ao operador iniciar a radiação antes que a máquina fosse ajustada para a dosagem correta, e má coordenação entre o projeto do *hardware* e *software*, que resultou na ausência de certas características de segurança. Você pode aprender mais sobre tais problemas no Fórum de Riscos, cuja página está localizada em <http://catless.ncl.ac.uk/Risks>

As leis de direitos autorais foram definidas originalmente para proteger os direitos de propriedade de autores de trabalhos literários. Nesse caso, o valor está na forma como as idéias são expressas, e não nas idéias propriamente ditas. O valor de um poema está no seu ritmo, estilo e formato, e não no assunto tratado; o valor de um romance está na forma como o autor apresenta a história, e não na história em si. Assim, o investimento de um poeta ou de um autor pode ser protegido atribuindo-lhe o direito de propriedade daquela expressão particular da idéia, mas não da própria idéia. Outra pessoa é livre para expressar a mesma idéia, contanto que tal expressão não seja “significativamente similar” à original.

Resumindo, as leis do direito autoral foram desenvolvidas para proteger a forma e não a função, mas o valor de um programa em geral está em sua função, e não em sua forma. Assim, uma aplicação direta da lei de direito autoral tenderia a não proteger o investimento do desenvolvedor do *software*. Em geral, os tribunais têm reconhecido esse problema e sido receptivos às tentativas de dar aos desenvolvedores de *software* justa proteção

ção segundo a lei de direito autoral existente. O problema está no estabelecimento do significado de “significativamente similar” no caso de *software*.

Seria fútil considerar dois programas “significativamente similares” apenas porque realizam a mesma tarefa. Isso levaria à conclusão de que deve haver um único sistema operacional, uma vez que a tarefa de qualquer sistema operacional é coordenar as atividades da máquina e a sua alocação de recursos. Entretanto, o que acontece quando a estrutura subjacente (representada por diagramas estruturais ou de colaboração) de dois programas é a mesma? Isso indica um plágio ou não? Se, por exemplo, a estrutura em comum resultou da aplicação de um padrão de projeto bem conhecido, então a similaridade pode meramente refletir o fato de os dois programas terem sido bem projetados. De maneira semelhante, o fato de dois programas executarem uma tarefa na mesma maneira pode simplesmente refletir a realidade de que existe um único e melhor algoritmo para a aplicação específica, e não indica brecha na lei do direito autoral.

Para enfrentar tais problemas, os tribunais vêm aplicando há muito tempo as técnicas de filtragem para selecionar as similaridades que não indicam transgressão à lei. Em termos de *software*, alguns itens que têm sido argumentados como filtros incluem as características determinadas por padrões, as características que são essencialmente ditadas pelas consequências lógicas dos objetivos do programa e os componentes que fazem parte do domínio público. (Essa abordagem para determinar a similaridade significativa é a idéia fundamental por trás do procedimento mais preciso, conhecido no jargão jurídico por filtragens sucessivas e testes de abstrações.)

Assim, se as similaridades nestas áreas não constituem transgressão à lei de direito autoral, que tipo de similaridade infringe a lei? Alguns queixosos têm argumentado com sucesso que a aparência e o jeito de um sistema de *software* devem ser protegidos pela lei do direito autoral. Embora os termos *aparência* e *jeito* não tenham sido utilizados até 1985, os conceitos estão enraizados desde os anos 1960, quando a IBM introduziu a sua série Sistema /360 de máquinas. Essa série consistia em diversas máquinas voltadas para aplicações tanto de pequenas empresas como de grandes corporações com significativa demanda de processamento. Todas as máquinas eram fornecidas com sistemas operacionais cuja comunicação com o ambiente era essencialmente a mesma, isto é, a série completa de máquinas possuía uma interface padronizada com o usuário. Logo, à medida que a empresa crescia, ela podia migrar para uma máquina mais possante na série /360 sem muita reprogramação e sem esforços de treinamento. Sem

dúvida, a *aparência* (ou seja, a imagem projetada pelo *software* do sistema) e o *jeito* (que significa a maneira como o usuário interagia com o *software* do sistema) eram os mesmos para todas as máquinas da série /360.

Atualmente, as vantagens das interfaces padronizadas são bem reconhecidas e solicitadas no espectro de *software*. Quando a interface projetada por uma companhia se torna popular, passa a ser vantajoso para as companhias concorrentes projetar seus sistemas com a aparência e o jeito daquela bem conhecida. Essa similaridade facilita aos usuários do sistema bem conhecido a sua conversão para o sistema do concorrente, ainda que os projetos internos dos dois sistemas sejam bem diferentes. As companhias que enfrentam esse tipo de competição têm solicitado proteção sob a lei do direito autoral, alegando propriedade da aparência e do jeito do sistema original. Afinal de contas, a aparência e o jeito de um pacote de *software* têm muitas das características de propriedades protegidas pela lei de direito autoral.

Um primeiro teste da argumentação de aparência e jeito ocorreu em 1987, quando a Lotus Development Corporation processou a Mosaic Software, alegando que esta havia copiado a aparência e o jeito de seu sistema de planilhas Lotus 1-2-3. A causa foi ganha. Contudo, em casos mais recentes, a aparência e o jeito têm tido resultados variados. Se, por exemplo, a defesa consegue convencer o tribunal de que a aparência e o jeito de um sistema são tão comuns que se tornaram padrão para o domínio público, então eles se tornam sujeitos à filtragem e não são protegidos pela lei.

O *software* é único no sentido em que é uma mercadoria à qual as leis de direito autoral e de patentes têm sido aplicadas. De fato, isto tem feito com que os tribunais às vezes fiquem relutantes em interpretar a proteção por direito autoral amplamente, temendo a sobreposição com a lei de patentes.

Como acontece com o direito autoral, as patentes também encontram problemas fundamentais quando usadas para proteger os direitos de propriedade de *software*. Um obstáculo é o princípio bem estabelecido de que ninguém pode se apropriar de fenômenos naturais, como as leis da física, as fórmulas matemáticas e os pensamentos — uma coleção que, sustentada nos tribunais, inclui os algoritmos. Existe, contudo, um número crescente de casos nos quais os desenvolvedores de *software* têm obtido patentes, como, por exemplo, o algoritmo de criptografia conhecido como RSA, amplamente utilizado nos sistemas de criptografia de chave pública.

Outro obstáculo ao uso de patentes para proteger os direitos de propriedade de *software* é que a obtenção da patente é um processo caro e demorado, que geralmente leva vários anos. Durante este tempo, um produto de *software* pode ficar obsoleto e, até que a patente seja concedida, o candidato não terá direito algum de impedir que terceiros se apropriem do produto.

Os direitos autorais e as leis de patentes são projetados para proteger os direitos de criadores e inventores, de modo a estimulá-los a publicar suas realizações. Em contraste, as leis de segredos industriais constituem um meio de restrição à disseminação de idéias. Projetadas para manter uma conduta ética entre empresas concorrentes, essas leis protegem contra a abertura imprópria ou a apropriação ilegal das realizações internas de uma empresa. A questão da abertura imprópria freqüentemente é formalizada como acordos de não-abertura, pelos quais uma empresa exige dos que têm acesso aos segredos da companhia que se comprometam a não revelar seus conhecimentos para terceiros. Os tribunais geralmente têm apoiado tais acordos.

Para proteger-se contra as responsabilidades associadas ao seu produto, os fabricantes de *software*, em geral acompanham seus produtos de termos de negação de responsabilidade, em que são declarados os limites de sua responsabilidade. São freqüentes declarações como “Em nenhum caso a Companhia X será responsável por qualquer dano que ocorra em decorrência do uso deste *software*”. Entretanto, os tribunais raramente levam em consideração esses termos se a acusação provar negligência por parte do acusado. Assim, casos de responsabilidade tendem a concentrar-se em estabelecer se o acusado apresentou ou não um nível de cuidado compatível com o produto em questão. Um nível de cuidado que seria julgado aceitável no caso de desenvolvimento de um sistema de processamento de texto pode ser considerado negligente no caso de desenvolvimento de um *software* para o controle de um reator nuclear. Logo, a melhor defesa contra a reivindicação de responsabilidade do *software* é a aplicação dos princípios da engenharia de *software* durante o processo de desenvolvimento.



QUESTÕES/EXERCÍCIOS

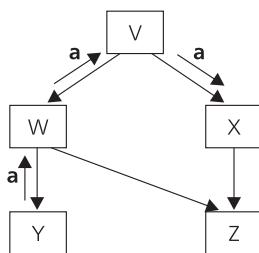
1. Que teste pode ser aplicado para decidir se um programa é substancialmente semelhante a outro?
2. Em que sentido as leis de direitos autorais, patentes e segredos industriais são projetadas para beneficiar a sociedade?
3. Até que ponto os termos de negação de responsabilidade não são reconhecidos pelos tribunais?

Problemas de revisão de capítulo

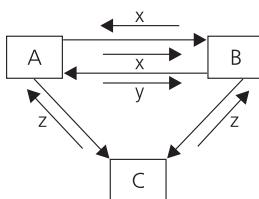
(Os problemas marcados com asterisco se referem às seções opcionais.)

1. Dê um exemplo de como os esforços para o desenvolvimento de *software* podem ser mais tarde vantajosos para a sua manutenção.
2. O que é prototipação evolutiva?
3. Faça um resumo do papel da utilização de ferramentas de Engenharia de *Software* Assistida por Computador (CASE) na mudança do processo de desenvolvimento de *software*.
4. Explique de que maneira a escassez de métricas para a avaliação quantitativa de certas propriedades do *software* afeta a disciplina de engenharia de *software*.
5. Suponha que você tenha uma métrica para a medição da complexidade de um sistema de *software*. Ela seria cumulativa? Isto é, a complexidade do sistema completo seria a soma das complexidades de suas partes? Justifique a sua resposta.
6. Suponha que você tenha uma métrica para a medição da complexidade de um sistema de *software*. Ela seria cumulativa, no sentido de que a complexidade do sistema completo seria a mesma se ele tivesse sido originalmente desenvolvido com o recurso X e mais tarde acrescentado o recurso Y, ou se tivesse sido originalmente desenvolvido com o recurso Y e mais tarde fosse acrescentado o recurso X? Justifique a sua resposta.
7. Em que a engenharia de *software* difere das outras áreas mais tradicionais da engenharia, como a elétrica e a mecânica?
8. a. Identifique uma desvantagem do modelo tradicional da cachoeira para o desenvolvimento de *software*.
- b. Identifique uma vantagem do modelo tradicional da cachoeira para o desenvolvimento de *software*.
9. De que maneira a definição de códigos de ética e de conduta profissional ajuda no desenvolvimento de *software* de alta qualidade?
10. Descreva como o uso de constantes em vez de literais pode simplificar a manutenção de um *software*.
11. Qual a diferença entre acoplamento e coesão? Qual deles deve ser minimizado e qual maximizado? Por quê?
12. Selecione um objeto do cotidiano e analise os seus componentes em termos da coesão funcional e lógica.
13. Quais das seguintes afirmações é um argumento a favor do acoplamento e qual é a favor da coesão?
 - a. Para um estudante aprender uma disciplina, esta deve ser apresentada em módulos bem organizados, com metas específicas.
 - b. Um estudante não comprehende de fato uma disciplina até que o escopo global da mesma e sua relação com as demais sejam assimilados.
14. No texto, mencionamos o acoplamento de controle, mas não o analisamos. Compare o acoplamento entre dois módulos de programa obtidos pelo uso de uma instrução *goto* simples com o obtido por meio de uma chamada de procedimento.
- *15. Identifique uma complicação em relação ao acoplamento de dados que pode ocorrer em contextos de processamento paralelo.

- 16.** Responda às perguntas seguintes, relativas ao diagrama estrutural abaixo:



- a. Para qual módulo o módulo Y devolve o controle?
 - b. Para qual módulo o módulo Z devolve o controle?
 - c. Os módulos W e X são interligados por acoplamento de controle?
 - d. Os módulos W e X são interligados por acoplamento de dados?
 - e. Qual dado é compartilhado pelos módulos W e Y?
 - f. De que forma os módulos Y e X se relacionam?
- 17.** Em relação ao diagrama estrutural do Problema 16, que terminações são necessárias para testar o módulo V? Que características as terminações devem apresentar?
- 18.** Responda às perguntas seguintes, em relação ao diagrama estrutural abaixo:



- a. Qual a diferença entre as maneiras como os módulos A e B usam os dados x e y?
 - b. Se um dos módulos estiver encarregado de obter o dado z de um usuário em um terminal distante, provavelmente qual módulo será?
- 19.** Faça um diagrama de classes simples que represente o relacionamento entre editores de revistas, as revistas e os assinantes.

- 20.** Estenda o diagrama de classes feito no problema anterior para ser tornar um diagrama de colaboração.
- 21.** Qual é a diferença entre um diagrama de classes e um de colaboração?
- 22.** O que é UML?
- 23.** É importante entender que um diagrama de classes representa os relacionamentos entre as classes, que são os gabaritos a partir dos quais os objetos são construídos. Para esclarecer esse ponto, suponha que três clientes estejam fazendo compras pela Internet na loja virtual descrita na Seção 6.3. Usando retângulos para representar objetos, faça um diagrama que mostre os objetos e seus relacionamentos que estariam presentes no sistema proposto na Figura 6.4.
- 24.** Usando um diagrama estrutural, represente a estrutura modular de um sistema *on-line* de registros de estoque/cliente para uma loja virtual. Quais módulos do seu sistema deverão ser alterados se houver modificações nas leis de impostos sobre vendas? E se houver mudança no comprimento do código de endereçamento do sistema postal?
- 25.** Projete uma solução orientada a objeto para o problema anterior e a represente por um diagrama de classes.
- 26.** Que problemas aparecem durante a fase de modificação de um programa grande que foi projetado com todos os seus elementos de dados globais?
- 27.** Identifique alguns padrões de projeto em outras áreas que não a arquitetura e a engenharia de *software*.
- 28.** Resuma o papel dos padrões de projeto na engenharia de *software*.
- 29.** Até que ponto as estruturas de controle em uma típica linguagem de programação de alto nível (*if-then-else*, *while* etc.) são padrões de projeto em pequena escala?
- 30.** Faça um diagrama de fluxo de dados que descreva o processo de inscrição em uma universidade.
- 31.** Compare a informação representada pelos diagramas de fluxo de dados com a representada pelos diagramas estruturais.

- 32.** Qual a diferença entre uma relação de um para muitos e uma de muitos para muitos?
- 33.** Dê um exemplo de uma relação de um para muitos que não tenha sido mencionada neste capítulo. Apresente um exemplo de uma relação de muitos para muitos que não tenha sido mencionada neste capítulo.
- 34.** Faça um diagrama de entidades e relacionamentos em que sejam representados os relacionamentos entre cozinheiros, garçonetes, clientes e caixas em um restaurante.
- 35.** Faça um diagrama de entidades e relacionamentos que represente as relações entre revistas, seus editores e seus assinantes.
- 36.** Nos seguintes casos, identifique se as atividades indicadas se relacionam com diagramas estruturais, de fluxo de dados, de entidades e relacionamentos ou com um dicionário de dados.
- Identificar os dados pertinentes ao sistema a ser desenvolvido.
 - Identificar a relação entre os vários elementos de dados que figuram no sistema.
 - Identificar as características de cada item de dados do sistema.
 - Identificar quais dados são compartilhados pelas várias partes do sistema.
- 37.** Qual é a diferença entre um diagrama de classes e um de entidades e relacionamentos?
- 38.** Resuma a diferença entre as estratégias de projeto cima-baixo e baixo-cima.
- 39.** Quais das sentenças abaixo ilustram o princípio de Pareto?
- Uma pessoa desagradável pode estragar a festa de todos.
 - Cada estação de rádio se concentra em um formato particular, como música *hard rock*, música clássica ou entrevistas.
- 40.** Os engenheiros de *software* esperam que os grandes sistemas sejam homogêneos ou heterogêneos quanto aos erros? Explique a sua resposta.
- 41.** O teste do caminho base é uma metodologia pela qual cada *instrução* do programa é executada pelo menos uma vez. Como isso difere da garantia de que cada *caminho* seja executado pelo menos uma vez?
- 42.** Qual a diferença entre o teste em caixa de vidro e o teste em caixa preta?
- 43.** Dê algumas analogias aos testes em caixa de vidro e em caixa preta que ocorrem em áreas que não a engenharia de *software*.
- 44.** De que maneira o desenvolvimento de código aberto difere do teste beta? (Considere o teste em caixa de vidro *versus* teste em caixa preta.)
- 45.** O desenvolvimento de código aberto é uma metodologia cima-baixo ou baixo-cima? Justifique a sua resposta.
- 46.** Suponha que 100 erros tenham sido intencionalmente introduzidos em um sistema de *software* de grande porte, antes de passar pelo teste final. Além disso, suponha que um total de 200 erros tenha sido descoberto e corrigido durante tal teste, dos quais 50 pertenciam ao grupo dos intencionalmente colocados no sistema. Se os 50 erros restantes forem então corrigidos, quantos erros não-identificados você estima que ainda se encontrariam no sistema? Por quê?
- 47.** De que forma as leis tradicionais de direitos autorais não conseguem salvaguardar os investimentos dos desenvolvedores de *software*?
- 48.** De que forma as leis tradicionais de patentes não conseguem salvaguardar os investimentos dos desenvolvedores de *software*?

Questões sociais

As seguintes questões procuram auxiliar o leitor no entendimento de alguns assuntos éticos, sociais e legais no campo da computação. O objetivo não é meramente que o leitor responda as perguntas. Ele deverá também justificá-las e verificar se as justificativas apresentadas preservam sua consistência de uma questão para outra.

1. a. A analista Mary foi encarregada de projetar um sistema para armazenar registros médicos em uma máquina conectada a uma grande rede. Suas exigências quanto à segurança não foram plenamente atendidas por razões financeiras, e foi-lhe ordenado que prosseguisse com o projeto, usando um sistema de segurança por ela considerado inadequado. Qual deve ser a sua conduta? Por quê?
b. Suponha que a analista Mary tenha desenvolvido o sistema como lhe fora ordenado, e que esteja ciente de que tais registros médicos estão sendo observados por pessoas não-autorizadas. Qual deve ser a sua conduta? Até que ponto ela é responsável por essa falha de segurança?
c. Suponha que, em vez de obedecer às ordens, Mary se recuse a prosseguir com o sistema e torne público tal fato, causando um enorme prejuízo financeiro à companhia e a perda de emprego de muitos funcionários inocentes. Esta conduta de Mary estaria correta? Entretanto, e se tal sistema fosse apenas uma parte de um projeto, e a analista Mary não soubesse que estivessem sendo feitos esforços reais em outro setor da companhia para desenvolver um sistema de segurança confiável, o qual seria aplicado ao sistema no qual ela estava trabalhando? Como isto mudaria sua opinião acerca da conduta de Mary? (Convém não esquecer que a visão dela sobre a situação não mudou.)
2. Quando sistemas de *software* de grande porte são desenvolvidos por muitas pessoas, como as responsabilidades podem ser distribuídas? Há uma hierarquia de responsabilidade? Há graus de responsabilidade?
3. Vimos que sistemas de *software* complexos de grande porte em geral são desenvolvidos por muitas pessoas, das quais apenas poucas visualizam o quadro completo do projeto. É aconselhável que um funcionário participe de um projeto sem total conhecimento de suas funções?
4. Até que ponto alguém é responsável pela forma final como suas realizações são utilizadas por outras pessoas?
5. No relacionamento entre um profissional de computação e um cliente, é da responsabilidade do profissional implementar as solicitações do cliente ou, em vez disso, orientar os seus desejos? E se o profissional intuir que os desejos de seu cliente podem ter consequências pouco éticas? Por exemplo, se o cliente deseja efetuar simplificações em favor da eficiência, mas o profissional pode prever, como decorrência, uma fonte de dados errôneos em potencial ou o uso indevido do sistema se tais alternativas forem adotadas. Se o cliente insistir, o profissional estará livre de responsabilidades?
6. Em muitas sociedades, a livre concorrência é vista como benéfica para o povo, pois estimula a criação de melhores produtos com menores preços. O que acontece, contudo, se a tecnologia começa a avançar tão rapidamente que novas invenções se tornam obsoletas antes que o inventor possa ter lucro com a invenção?
7. A revolução computacional está contribuindo para os problemas mundiais de energia ou ajudando a resolvê-los?
8. A sociedade continuará buscando novas aplicações para a tecnologia? Quais, se nada pode ser feito para reverter a dependência da sociedade à tecnologia? Qual seria o resultado para uma sociedade que continua a encontrar novas aplicações para a tecnologia e nunca reverte essa tendência?

Leituras adicionais

- Alexander, C., S. Ishikawa, and M. Silverstein. *A Pattern Language*. New York: Oxford Univ. Press, 1977.
- Beck, K. *Extreme Programming Explained*. Boston, MA: Addison-Wesley, 2000.
- Brooks, F. P. *The Mythical Man-Month*, Anniversary ed. Boston, MA: Addison Wesley Longman, 1995.
- Fenton, N. E. and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. Boston, MA: PWS, 1997.
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
- Pfleeger, S. L. *Software Engineering: Theory and Practice* 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 2001.
- Pooley, R. and P. Stevens. *Using UML: Software Engineering with Objects and Components*, revised ed. Reading, MA: Addison-Wesley, 2000.
- Pressman, R. S. *Software Engineering: A Practitioner's Approach*, 5th ed. New York: McGraw-Hill, 2001.
- Schach, S. R. *Classical and Object-Oriented Software Engineering*, 5th ed. New York: McGraw-Hill, 2002.
- Sommerville, I. *Software Engineering*, 6th ed. Reading, MA: Addison-Wesley, 2001.

Organização de dados

Vimos que a informação armazenada em um computador é representada em forma codificada e organizada de maneira compatível com a tecnologia apropriada, como células de memória individualmente endereçáveis, ou como setores em um disco. Essa organização raramente é a mesma que a idealizada pelo usuário dos dados. Por exemplo, os dados que representam as vendas semanais de uma equipe de vendedores de uma empresa devem ser apresentados na forma de tabelas, com uma coluna para cada dia da semana e uma linha para cada elemento da equipe de vendas. Ou então, os nomes e os cargos dos funcionários de uma empresa podem ser apresentados na forma de um quadro organizacional. Além disso, uma empresa pode desejar que seus cadastros estejam ordenados por número para uma aplicação e por custo para outra.

Na Parte 3, estudaremos como uma máquina pode ser programada para apresentar os dados nessas formas conceituais e mais úteis e de que maneira este objetivo de criar imagens ilusórias dos dados afeta a forma como os dados são de fato guardados na máquina. No Capítulo 7, analisaremos dados guardados na memória principal, no Capítulo 8, consideraremos o armazenamento em massa e, no Capítulo 9, introduziremos o tópico de sistemas de banco de dados.

Estruturas de dados

A memória principal de um computador é organizada na forma de células individuais, com endereços consecutivos. Entretanto, pode ser conveniente associar a tais células outros arranjos de dados. Por exemplo, registros de vendas semanais são mais facilmente visualizados em tabelas nas quais as vendas de diferentes produtos em diferentes dias são organizadas em linhas e colunas. Neste capítulo, analisaremos como são simuladas tais organizações abstratas de dados. O objetivo é permitir ao usuário dos dados raciocinar em termos de organizações abstratas, em vez de se preocupar com a organização real existente na memória principal do computador.

7.1 Básico de estruturas de dados

Abstração novamente
Estruturas estáticas *versus* estruturas dinâmicas
Ponteiros

7.2 Matrizes

7.3 Listas

Listas contíguas
Listas ligadas
Suporte à lista conceitual

7.4 Pilhas

Retrorastreamento (*backtracking*)
Implementação de pilhas

7.5 Filas

7.6 Árvores

Implementação de árvores
Um pacote para árvores binárias

* 7.7 Tipos de dados personalizados

Tipos definidos pelo usuário
Classes

* 7.8 Ponteiros em linguagem de máquina

*Os asteriscos indicam sugestões de seções consideradas opcionais.

7.1 Básico de estruturas de dados

Começamos isolando três pontos fundamentais no estudo das estruturas de dados: a abstração, a distinção entre estruturas estáticas e dinâmicas, e o conceito de ponteiro.

Abstração novamente

Já encontramos, é claro, o conceito de abstração muitas vezes. Contudo, devemos enfatizá-lo novamente, uma vez que a criação de ferramentas abstratas é a essência do tema das estruturas de dados. De fato, o tópico de estruturas de dados explora as maneiras como os usuários podem ficar isolados dos detalhes do armazenamento corrente (células de memória e endereços) e ainda assim ter acesso à informação como se ela estivesse armazenada de forma mais conveniente.

O termo *usuário*, neste contexto, não necessariamente se refere a uma pessoa; em vez disso, o significado da palavra depende da perspectiva no momento. Se estamos pensando em uma pessoa que usa um PC para manter um cadastro da Liga de Boliche, então o usuário é a pessoa. Nesse caso, o *software* de aplicação que está sendo usado é o responsável por apresentar os dados em uma forma abstrata conveniente à pessoa — talvez como uma coleção de tabelas. Se estamos pensando em um servidor na Internet, então o usuário pode ser uma outra máquina que desempenhe o papel de cliente. Nesse caso, o servidor é responsável por apresentar os dados em uma forma abstrata conveniente ao cliente. Se estamos pensando em termos da estrutura modular de um programa, então o usuário é qualquer módulo que necessite acesso aos dados. Nesse caso, o módulo que contém os dados é responsável por apresentar os dados em uma forma abstrata conveniente aos outros módulos.

Em cada um desses cenários, a linha comum é que o usuário, independentemente de sua constituição, tem o privilégio de acessar os dados como uma ferramenta abstrata. O modo como essas ferramentas são construídas é o tema central das estruturas de dados.

Estruturas estáticas versus estruturas dinâmicas

Uma importante distinção quando se constroem imagens abstratas de dados é se a estrutura que está sendo simulada é estática ou dinâmica — isto é, se a forma ou o tamanho das estruturas muda com o tempo. Por exemplo, se a ferramenta abstrata em construção é uma lista de nomes, é importante considerar se a lista manterá o tamanho fixo durante a sua existência ou se encolherá ou expandir-se-á à medida que os nomes forem eliminados ou acrescentados.

Como regra geral, as estruturas estáticas são mais fáceis de manejar do que as dinâmicas. Se uma estrutura é estática, precisamos apenas prover os meios de acesso aos vários itens de dados na estrutura e talvez os meios para alterar os valores em localizações específicas. Entretanto, se a estrutura for dinâmica, também deveremos lidar com os problemas de acrescentar e eliminar elementos de dados, bem como de encontrar o espaço de memória necessário a uma estrutura de dados crescente. No caso de uma estrutura mal projetada, um crescimento excessivo pode implicar a cópia da estrutura inteira para outra área de memória quando existir espaço disponível — um processo que pode consumir muito tempo.

Ponteiros

Lembre-se de que as várias células na memória principal de uma máquina são identificadas por endereços numéricos. Por serem valores numéricos, esses endereços podem, por sua vez, ser armazenados em células de memória. Um **ponteiro** é uma célula de memória (ou talvez um bloco de células de memória) que contém o endereço de outra célula de memória. No caso das estruturas de dados, os ponteiros são usados para registrar a localização onde os itens de dados estão armazenados, isto é, tendo guardado um item de dados em uma célula de memória, podemos armazenar o endereço desses dados em um ponteiro e mais tarde usá-lo como um meio de recuperar os dados. Sem dúvida, o valor do ponteiro nos dirá onde encontrar os dados. Em um certo sentido, ele aponta para os dados, daí o seu nome.

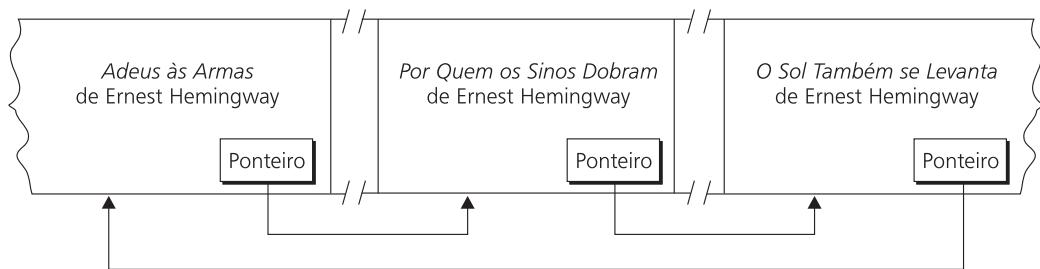


Figura 7.1 Acervo de biblioteca organizado por título, mas ligado por autor.

Já encontramos o conceito de ponteiro no contexto do contador de instruções dentro da UCP, que é usado para guardar o endereço da próxima instrução a ser executada. De fato, um outro nome para o contador de programa é **contador de instruções**. As URLs, usadas para ligar documentos de hipertexto, também podem ser consideradas exemplos de ponteiros, exceto em que elas apontam para localizações na Internet, e não na memória principal.

Muitas linguagens de programação atuais incluem os ponteiros como tipos de dados primitivos. Logo, elas permitem a declaração, alocação e manipulação de ponteiros, bem como essas operações para números inteiros e cadeias de caracteres. Usando tais linguagens, um programador pode projetar redes elaboradas de itens de dados na memória da máquina, onde cada bloco de células de memória contém ponteiros para outros blocos. Assim, caminhos podem ser percorridos de um bloco para outro seguindo os ponteiros.

Como exemplo, suponha que tenhamos uma lista de romances organizada alfabeticamente por títulos e guardada na memória de um computador. Embora conveniente em muitas aplicações, esse arranjo torna difícil encontrar todos os romances de um mesmo autor, uma vez que eles se encontram espalhados na lista. Para resolver esse problema, podemos reservar uma célula de memória adicional do tipo ponteiro em cada bloco de células que representa um romance. Então, em cada ponteiro, colocamos o endereço de outro bloco que representa um livro do mesmo autor, de forma que os romances de cada autor ficam ligados em um laço (Figura 7.1). Uma vez encontrado um romance de um autor, encontraremos todos os outros seguindo os ponteiros de um livro para outro.



QUESTÕES/EXERCÍCIOS

1. Descreva uma aplicação em que você espere envolver uma estrutura de dados estática. Então, descreva uma em que uma estrutura de dados dinâmica esteja envolvida.
2. Descreva contextos fora da Ciência da Computação nos quais o conceito de ponteiros ocorra.

7.2 Matrizes

No Capítulo 5, vimos que muitas linguagens de programação de alto nível permitem que o programador expresse os algoritmos como se os dados manipulados estivessem armazenados em um arranjo retangular chamado arranjo homogêneo ou matriz, onde o termo *homogêneo* significa que todos os elementos do arranjo são do mesmo tipo. Nesta seção, investigaremos como essas matrizes são de fato implementadas e como um tradutor converte as referências à matriz que ele encontra no programa-fonte em terminologia de células de memória e endereços.

Suponhamos que um algoritmo para manipular um conjunto de 24 leituras de temperatura, tomadas a cada hora, seja expresso em uma linguagem de alto nível. O programador provavelmente achará conveniente organizar tais leituras em uma matriz unidimensional chamada `Leitura`, onde a primeira leitura pode ser referenciada como `Leituras[1]`, a segunda, como `Leituras[2]`, e assim por diante. (Se você for usuário das linguagens C, C++, C# ou Java, essas referências terão as formas `Leituras[0]` e `Leituras[1]`, mas para a nossa discussão, é conveniente pensarmos que o primeiro elemento tenha índice um. A conversão é trivial — veja a Questão/Exercício 4 no fim desta seção.)

A conversão da organização conceitual de matriz unidimensional para o arranjo real interno da máquina pode ser direta, uma vez que os dados podem ser armazenados em uma seqüência de 24 posições de memória, com endereços consecutivos na mesma matriz idealizada pelo programador. Conhecendo o endereço da primeira posição desta seqüência, um tradutor converte dados como `Leituras[4]` para a terminologia própria de memória. Ele apenas subtrai um do índice do elemento desejado e soma o resultado com o endereço da célula de memória que contém a primeira leitura de temperatura. Se a primeira leitura estiver no endereço x , então a quarta leitura estará no endereço $x + (4 - 1)$ como mostrado na Figura 7.2.

Agora suponha que um programador queira escrever um programa para registrar as vendas feitas pela equipe de vendas de uma empresa durante uma semana. Podemos imaginar tais dados organizados em uma tabela, com os nomes do pessoal de vendas listados na coluna mais à esquerda e os dias da semana na primeira linha da matriz. Conseqüentemente, o programador pode querer escrever o programa como se os dados estivessem dispostos em uma matriz bidimensional, onde os valores dispostos ao longo de cada linha indicam as vendas feitas por um funcionário, enquanto os dispostos em uma coluna representam todas as vendas feitas durante o dia correspondente.

A memória de uma máquina não é organizada nesta forma retangular, mas na forma de um conjunto de células individuais, com endereços consecutivos. Assim, a estrutura retangular exigida pela matriz deve ser simulada. Para tanto, conscientizemo-nos primeiramente de que a matriz é estática, no sentido em que o seu tamanho é invariável, apesar das atualizações dos seus dados. Portanto, podemos calcular a área necessária para seu armazenamento completo e reservar um bloco de células contíguas de memória que tenha esse tamanho. Em seguida, armazenamos, linha a linha, os dados nas posições consecutivas. Assim, começando na primeira célula do bloco reservado, armazenamos os valores da primeira linha da tabela para posições consecutivas de memória, depois armazenamos a próxima linha e a seguinte, e assim por diante (Figura 7.3). Nesse esquema de armazenamento, segue-se a **seqüência de linhas**^{*}, e não a **seqüência de colunas**^{**}, em que a seqüência de armazenamento da matriz seria coluna a coluna.

Com os dados armazenados dessa maneira, consideremos como poderíamos encontrar a célula de memória que contém o valor da terceira linha e quarta coluna da matriz. Para tanto, imaginemo-nos postados na primeira posição do bloco reservado para a

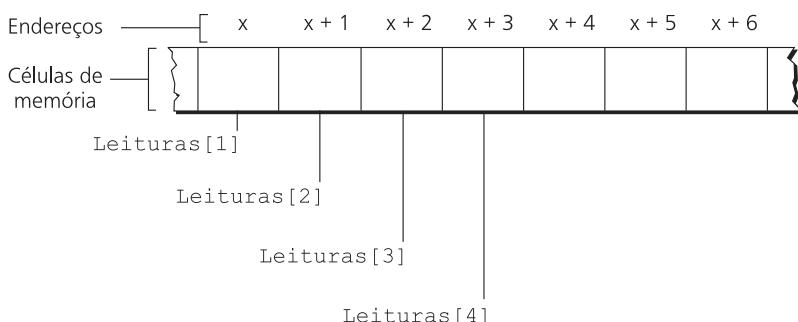


Figura 7.2 A matriz de `Leituras` armazenada em memória, com início no endereço x .

*N. de T. Em inglês, *row major order*.

**N. de T. Em inglês, *column major order*.

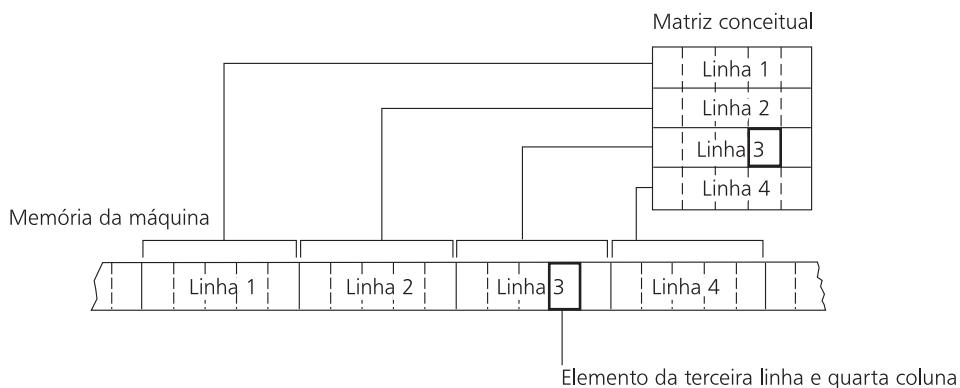


Figura 7.3 Uma matriz bidimensional com quatro linhas e cinco colunas, armazenada segundo sua seqüência das linhas.

matriz, na memória da máquina. A partir desta posição, encontramos o dado na primeira linha da matriz seguido pelo segundo, o qual, por sua vez, é seguido pelo terceiro e assim por diante. Para obter o dado da terceira linha, devemos nos deslocar para além da primeira e da segunda linhas. Como cada linha contém cinco elementos (um para cada dia, de segunda a sexta-feira), devemos nos deslocar para a frente de um total de dez elementos para atingir o primeiro elemento da terceira linha. Como estamos no início da terceira linha, devemos nos deslocar para adiante — mais três elementos — para alcançar o elemento da quarta coluna da matriz. Em resumo, para obter o elemento da terceira linha e quarta coluna, devemos nos deslocar 13 elementos, a partir do início do bloco.

O cálculo acima é um caso particular de um processo geral, utilizado para a obtenção de elementos de uma matriz bidimensional, quando armazenada pela sua seqüência de linhas. Em particular, se C representar o número de colunas da matriz (que é o número de elementos em cada linha), então o endereço do elemento na linha I e coluna J será

$$x + (c \times (i - 1)) + (j - 1)$$

onde x é o endereço da célula que contém o elemento na primeira linha e na primeira coluna da matriz. Isto é, precisamos saltar $i - 1$ linhas, em que cada uma contém c elementos, para atingir a linha I e depois mais $j - 1$ outros elementos, para encontrar o elemento J desta linha. No nosso exemplo anterior, $c = 5$, $i = 3$ e $j = 4$. Então, se a matriz estiver armazenada a partir do endereço x , o elemento da terceira linha e quarta coluna estará no endereço $x + (5 \times (3 - 1)) + (4 - 1) = x + 13$. (A expressão $(c \times (i - 1)) + (j - 1)$ às vezes é chamada de **polinômio de endereçamento**^{*}.)

Com esta informação, podem ser escritas rotinas de software para converter solicitações feitas em termos de linhas e colunas para posições relativas ao bloco de memória que contém a matriz. Por exemplo, um tradutor usa esta técnica para converter uma referência como *Vendas [2, 4]* em um endereço efetivo de memória. Assim, o programador pode desfrutar do privilégio de trabalhar com dados dispostos em tabelas (a estrutura conceitual), embora, na verdade, eles estejam armazenados em uma única linha (a estrutura real) interna da máquina.

*N. de T. Este polinômio constitui uma fórmula de linearização para matrizes multidimensionais.



QUESTÕES/EXERCÍCIOS

1. Mostre como a matriz abaixo aparece na memória quando armazenada na sua seqüência de linhas.

5	3	7
4	2	8
1	9	6

2. Escreva uma fórmula para encontrar o elemento da linha I e coluna J de uma matriz bidimensional se ela for armazenada na seqüência de colunas, e não na de linhas.
3. Se uma matriz bidimensional de 8 linhas e 11 colunas for armazenada na seqüência de linhas e iniciar no endereço de memória 25, qual será o endereço do elemento da terceira linha e da sexta coluna se cada elemento ocupar duas células de memória?
4. Nas linguagens de programação C, C++ e Java, os índices das matrizes começam em 0. Assim, o elemento da primeira linha e da quarta coluna de uma matriz `Array` é referenciado como `Array[0][3]`. Neste caso, qual polinômio de endereçamento é usado pelo tradutor para converter referências da forma `Array[i][j]` em endereços de memória?

Implementação de listas contíguas

As primitivas para construção e manipulação de matrizes fornecidas pela maioria das linguagens de programação de alto nível são ferramentas convenientes para construir e manipular listas contíguas. Se os elementos da lista forem todos de um mesmo tipo de dado primitivo, a lista nada mais será do que uma matriz unidimensional. Um exemplo ligeiramente mais complicado é uma lista de dez nomes, onde cada um não ultrapassa oito caracteres, como discutido no texto. Nesse caso, um programador poderia construir a lista contígua como uma matriz bidimensional de caracteres com dez linhas e oito colunas, que produziria a estrutura mostrada na Figura 7.4 (pressupondo que a matriz seja armazenada segundo sua seqüência de linhas).

Muitas linguagens de programação de alto nível incorporam recursos que estimulam essa implementação de listas. Por exemplo, se a matriz bidimensional de caracteres proposta acima for chamada `ListaMembro`, então a expressão `ListaMembro[3,5]` normalmente irá se referir ao único caractere contido na terceira linha da quinta coluna, mas algumas linguagens usam a expressão `ListaMembro[3]` para se referir à terceira linha inteira, que seria o terceiro elemento da lista.

7.3 Listas

Uma **lista** é uma coleção de elementos que aparecem em ordem seqüencial. Exemplos incluem as listas de presença em reuniões, as listas de “coisas a fazer” e os dicionários. Exemplos menos óbvios incluem sentenças, que podem ser vistas como seqüências de palavras, e palavras, que podem ser consideradas seqüências de letras. Ao contrário das matrizes, que são estruturas estáticas, as listas aparecem tanto na forma estática como na dinâmica. Nesta seção, trataremos dos tópicos que aparecem quando se implementa uma lista dinâmica, em oposição a uma lista estática.

Listas contíguas

Consideremos as técnicas para armazenar uma lista de nomes na memória principal de uma máquina. Uma estratégia é guardar a lista completa em um único bloco de células de memória, com endereços sucessivos. Supondo que cada nome não tenha mais de oito letras, podemos dividir o bloco de células em um conjunto de sub-blocos, cada um com oito células. Em cada sub-bloco, podemos armazenar um nome que guarde o seu código ASCII, usando uma célula para cada letra. Se um nome sozinho não preencher todas as células alocadas para ele, simplesmente preencheremos as posições restantes com

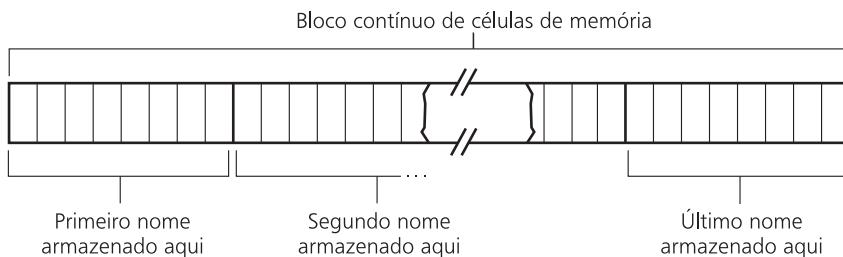


Figura 7.4 Nomes armazenados em memória como uma lista contínua.

o código ASCII correspondente ao espaço em branco. Usando este esquema, será necessário um bloco de 80 posições sucessivas de memória para armazenar uma lista de dez nomes.

Tal sistema de armazenamento é resumido na Figura 7.4. O ponto significativo é que a lista inteira é armazenada em um único grande bloco de memória, onde os elementos sucessivos seguem os outros em células contíguas. Essa organização se chama **lista contígua**.

Uma lista contígua é uma estrutura de armazenamento conveniente à implementação de listas estáticas, mas tem desvantagens quando usada para implementar listas dinâmicas. Suponha, por exemplo, que se queira eliminar um nome em uma lista implementada como lista contígua. Se este nome estiver no início da lista, cuja sequência deve ser mantida (provavelmente em ordem alfabética), deveremos deslocar todos os nomes que aparecem depois dele na lista para o começo da memória, a fim de preencher o espaço vazio deixado pela eliminação deste elemento. Um problema mais sério ocorrerá se desejarmos acrescentar nomes, uma vez que poderemos ter de mover a lista inteira para obter um bloco disponível de células contíguas grande o suficiente para conter a lista expandida.

Listas ligadas

Os problemas encontrados no tratamento das listas dinâmicas poderão ser simplificados, se permitirmos que os nomes individuais da lista sejam armazenados em diferentes áreas de memória, em vez de juntos em um único bloco contíguo. Para esclarecer, voltemos ao nosso exemplo de armazenamento de uma lista de nomes, cada um com oito caracteres no máximo. Guardaríamos cada nome em um bloco de nove células de memória. As oito primeiras são usadas para guardar o nome propriamente dito, e a última, como um ponteiro para o próximo nome na lista. Desta forma, a lista pode estar distribuída como um grupo de diversos pequenos blocos de nove células, interligados por ponteiros. Devido a tal sistema de encadeamento utilizado, essa organização é conhecida como **lista ligada**^{*}.

Para manter a informação de onde o primeiro elemento de uma lista ligada está localizado, reservamos uma posição de memória, na qual guardamos o endereço do primeiro elemento. Esta posição, o **ponteiro para o início da lista**^{**}, indica a posição física que contém o seu primeiro elemento.

Para marcar o fim da lista ligada, usamos um **ponteiro vazio**^{***}, padrão especial de bits que aparece na célula do ponteiro do último elemento para indicar que não existe qualquer outro elemento na lista. Por exemplo, se concordarmos que nunca armazenaremos um elemento da lista no endereço 0, o valor zero nunca aparecerá como um valor legítimo de ponteiro, e, portanto, poderemos utilizá-lo como o ponteiro vazio.

*N. de T. Em inglês, *linked list*.

**N. de T. Em inglês, *head pointer*.

***N. de T. Em inglês, *NIL pointer*.

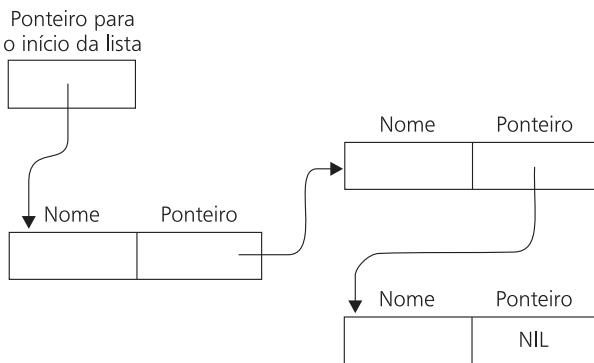


Figura 7.5 A estrutura de uma lista ligada.

nar e inserir elementos na lista. Nossa objetivo é verificar como o uso de ponteiros pode reduzir a quantidade dos movimentos de nomes, necessários quando armazenados em uma lista formada por um bloco único e contíguo. Primeiramente, observemos que um nome pode ser removido alterando-se um único ponteiro. Isto é feito modificando o ponteiro, que antes apontava para o nome removido, de forma que passe a apontar para o nome que o segue na lista (Figura 7.6). A partir daí, quando a lista for percorrida, o nome removido será ignorado, por não pertencer mais à cadeia.

Inserir um novo nome é ligeiramente mais trabalhoso. Primeiramente, encontramos um novo bloco de nove células de memória, armazenamos o novo nome nas primeiras oito posições e preenchemos a nona com o endereço do nome que deve figurar após esse novo nome. Finalmente, alteramos o ponteiro associado ao nome que deve preceder o novo elemento, de forma que ele passe a apontar para o novo nome (Figura 7.7). Depois disso, o novo elemento estará na posição correta todas as vezes que a lista for percorrida.

Suporte à lista conceitual

Quando escreve um programa, o programador freqüentemente deve decidir se a implementação das listas será com estrutura contígua ou ligada. Entretanto, uma vez tomada a decisão e estabelecida a lista, ele deve poder se abstrair dessas questões e se concentrar nos outros detalhes. Resumindo, o programa deve ser construído de forma que a lista possa ser referenciada em outras partes do mesmo como ferramenta abstrata.

Por exemplo, considere a tarefa de desenvolver um pacote de software encarregado de construir e manter o registro de matrículas em um sistema de cadastro universitário. O programador encarregado de desenvolver este pacote poderá resolver o problema definindo, para cada classe, uma estrutura própria, constituída de uma lista alfabeticamente ordenada dos estudantes dessa classe. Feito isso, os esforços do programador deverão ser direcionados às características mais gerais do problema, e não aos detalhes de como os dados

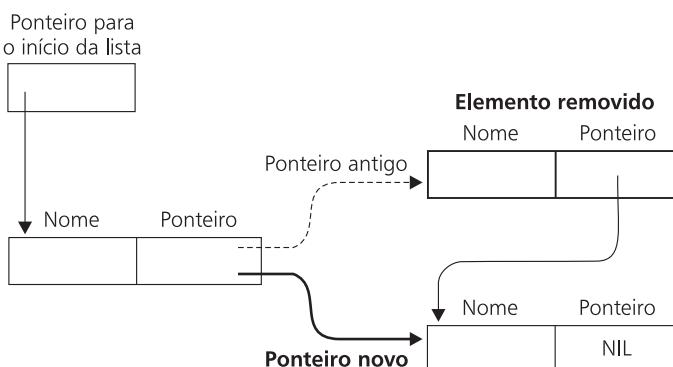


Figura 7.6 Eliminação de um elemento de uma lista ligada.

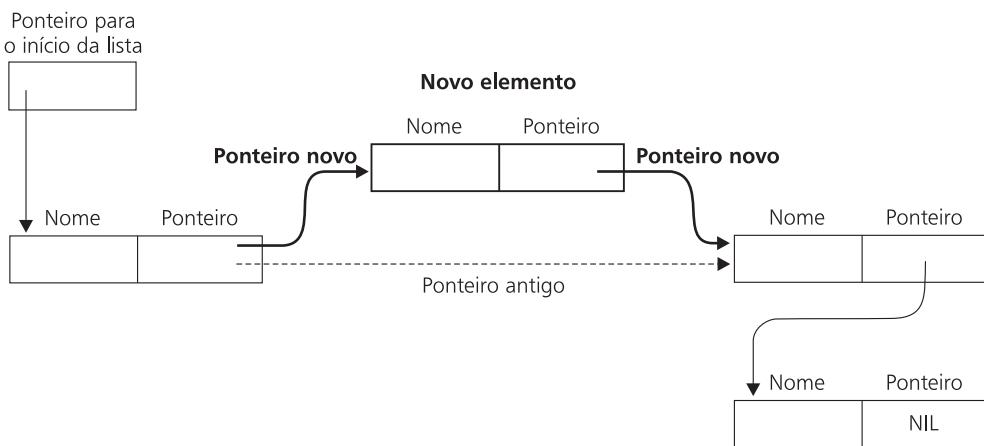


Figura 7.7 Inserção de um elemento em uma lista ligada.

podem ser movidos, em uma lista contígua, ou quais trocas de ponteiros são necessárias em uma lista ligada.

Para isso, o programador escreve uma coleção de procedimentos para a execução de atividades como inserir um novo elemento, eliminar um elemento existente, localizar um elemento ou imprimir a lista, e então usa esses procedimentos no restante do pacote de software para manipular as listas no sistema de registro. Por exemplo, para matricular o aluno J. W. Brown na disciplina Física 208, o programador usará uma instrução como

Insere (“Brown, J.W.”, “Física 208”)

deixando a cargo do procedimento `Insere` os detalhes da inserção. Dessa maneira, o programador pode desenvolver as outras partes do programa sem se preocupar com as tecnicidades de como a inserção é realizada.

Como um exemplo dessa abordagem, um procedimento chamado `ImprimeLista`, para imprimir uma lista ligada de nomes, está apresentado na Figura 7.8. O primeiro elemento da lista é apontado pelo ponteiro para o início da lista correspondente, e cada elemento da lista é formado de duas partes: um nome e um ponteiro para o próximo elemento. Uma vez desenvolvido esse procedimento, poderá ser utilizado como uma ferramenta abstrata para imprimir a lista, sem que seja preciso saber como esta será de fato realizada. Por exemplo, para obter uma lista impressa da classe de Economia 301, o programador escreverá apenas:

`ImprimeLista (“Economia 301”)`

```

procedure ImprimeLista (Lista)
PonteiroCorrente ← Ponteiro para o início da Lista
while (PonteiroCorrente não for NIL) do
    (Imprimir o nome do elemento apontado pelo PonteiroCorrente;
    Obter o valor contido no ponteiro do elemento de Lista, apontado pelo
    PonteiroCorrente, e reatribuir a PonteiroCorrente esse valor)

```

Figura 7.8 Um procedimento para imprimir uma lista ligada.



QUESTÕES/EXERCÍCIOS

1. Se você conhece o endereço do início do primeiro elemento de uma lista contígua, como obtém o endereço do seu quinto elemento? E no caso de uma lista ligada?
2. Que condição indica que uma lista ligada está vazia?
3. Modifique o procedimento da Figura 7.8 de modo que finalize a impressão quando um determinado nome for impresso.
4. Projete um procedimento para localizar e remover um elemento especificado em uma lista ligada.

7.4 Pilhas

Uma das características de uma lista que torna a estrutura ligada preferível à contígua é a necessidade de inserir e remover elementos. Essas operações, como sabemos, têm potencial para forçar a execução de muitos deslocamentos de nomes sempre que for necessário preencher ou abrir espaços vazios em uma lista contígua. Se restringirmos talas operações às extremidades da estrutura, concluiremos que o uso de uma estrutura contígua se torna mais conveniente. Um exemplo deste fenômeno é uma **pilha**^{*}, que é uma lista na qual todas as inserções e remoções são executadas por uma única extremidade da estrutura. A consequência dessa restrição é que o último elemento a entrar é o primeiro a ser removido — observação que leva as pilhas a serem conhecidas como estruturas **last-in, first-out — LIFO**^{**}.

A extremidade em que tais operações se realizam se chama topo da pilha. A outra às vezes é chamada de base da pilha. Para mostrar que o acesso a uma pilha é restrito apenas ao elemento mais ao topo, usaremos uma terminologia especial para designar as operações de inserção e remoção. O processo de inserir um objeto na pilha é chamado operação de **empilhamento**, e o de remover, operação de **desempilhamento**. Assim, em relação a pilhas, falamos em empilhar e desempilhar elementos.

Retrorastreamento (*backtracking*)

Uma aplicação clássica das pilhas ocorre quando uma unidade de programa solicita a execução de um procedimento. Para atender a essa solicitação, a máquina transfere sua atenção para o procedimento, e, quando sua execução estiver terminada, o controle deverá retornar à posição original na unidade de programa que fez a solicitação.

A situação real se mostra ainda mais complexa pelo fato de o procedimento poder solicitar a execução de outro procedimento, o qual, por sua vez, pode solicitar um outro e assim por diante (Figura 7.9). Como consequência, as posições a serem memorizadas vão sendo empilhadas. Depois, à medida que cada procedimento é concluído, a execução retorna ao ponto devido, no módulo de programa que ativou tal procedimento. Portanto, torna-se necessário um esquema para armazenar as posições de retorno e depois recuperá-las na ordem adequada.

Uma pilha é uma estrutura ideal para esse sistema. Quando um procedimento é solicitado, um ponteiro para a posição de retorno correspondente é empilhado no topo de uma pilha, e quando este procedimento estiver completo, o elemento do topo da pilha será removido, e seu conteúdo será um ponteiro com a indicação do endereço de retorno desejado.

*N. de T. Em inglês, *stack*.

**N. de T. Em português, estrutura em que o último a entrar é o primeiro a sair.

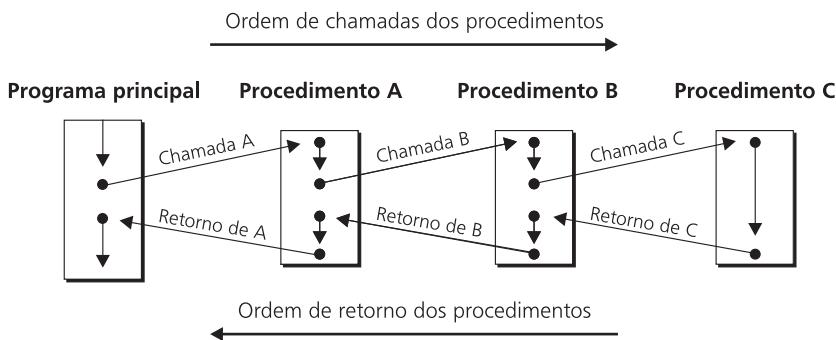


Figura 7.9 Procedimentos aninhados terminam na ordem inversa àquela em que foram ativados.

Este exemplo representa uma aplicação de pilha de modo genérico, porque demonstra a relação entre pilhas e o processo de retrorastreamento. Por **retrorastreamento** queremos denotar a maneira de sair de um ponto do sistema retroagindo as atividades que nos levaram até lá. Um exemplo clássico é a idéia de deixar rastros no passeio à floresta com o objetivo de seguir retroativamente os rastros para retornar ao ponto de partida. Tais rastros são sistemas LIFO e, assim, o conceito de pilha é inerente a qualquer processo que necessite reverter uma situação usando o mesmo caminho na ordem oposta em que ele foi percorrido.

Como outro exemplo, suponha que desejemos imprimir os nomes de um lista ligada (como analisado na Seção 7.2) na ordem inversa — isto é, o último nome deve ser o primeiro a ser impresso. Nossa problema é que precisamos de um meio para armazenar cada nome lido, até que todos os demais tenham sido recuperados e impressos. Nossa solução consiste em empilhar os nomes encontrados à medida que percorremos esta lista, do início ao final. A seguir, imprimimos os nomes, na ordem em que serão desempilhados (Figura 7.10). O procedimento para este processo está apresentado na Figura 7.11.

Implementação de pilhas

Para implementar uma estrutura de pilha na memória do computador, é comum reservar um bloco de células contíguas de memória suficientemente grande para acomodar a pilha à medida que ela cresce ou diminui. (Em geral, determinar o tamanho deste bloco é um problema crítico de projeto. Se o espaço reservado for muito pequeno, a pilha acabará por exceder o espaço de armazenamento a ela reservado. Se for muito grande, ocorrerá desperdício de memória.) Uma extremidade do bloco é designada como a base da pilha. É onde ficará armazenado o primeiro elemento empilhado, e cada um dos demais será colocado ao lado de seu antecessor à medida que a pilha crescer em direção à outra extremidade do bloco reservado.

Assim, à medida que elementos são empilhados e desempilhados, o topo da pilha vai se movendo para frente e para trás dentro do bloco de células de memória. Para manter atualizada essa posição, o endereço do elemento do topo é armazenado em uma célula adicional conhecida como **ponteiro da pilha**, isto é, o ponteiro da pilha aponta para o topo.

O sistema completo, descrito na Figura 7.12, funciona da seguinte forma: para empilhar um novo elemento, primeiramente fazemos o ponteiro da pilha apontar para a área livre que se encontra acima do topo, e aí inserimos o novo elemento. Para desempilhar um elemento, lemos o dado apontado pelo ponteiro da pilha e o ajustamos, apontando-o para o elemento imediatamente abaixo.

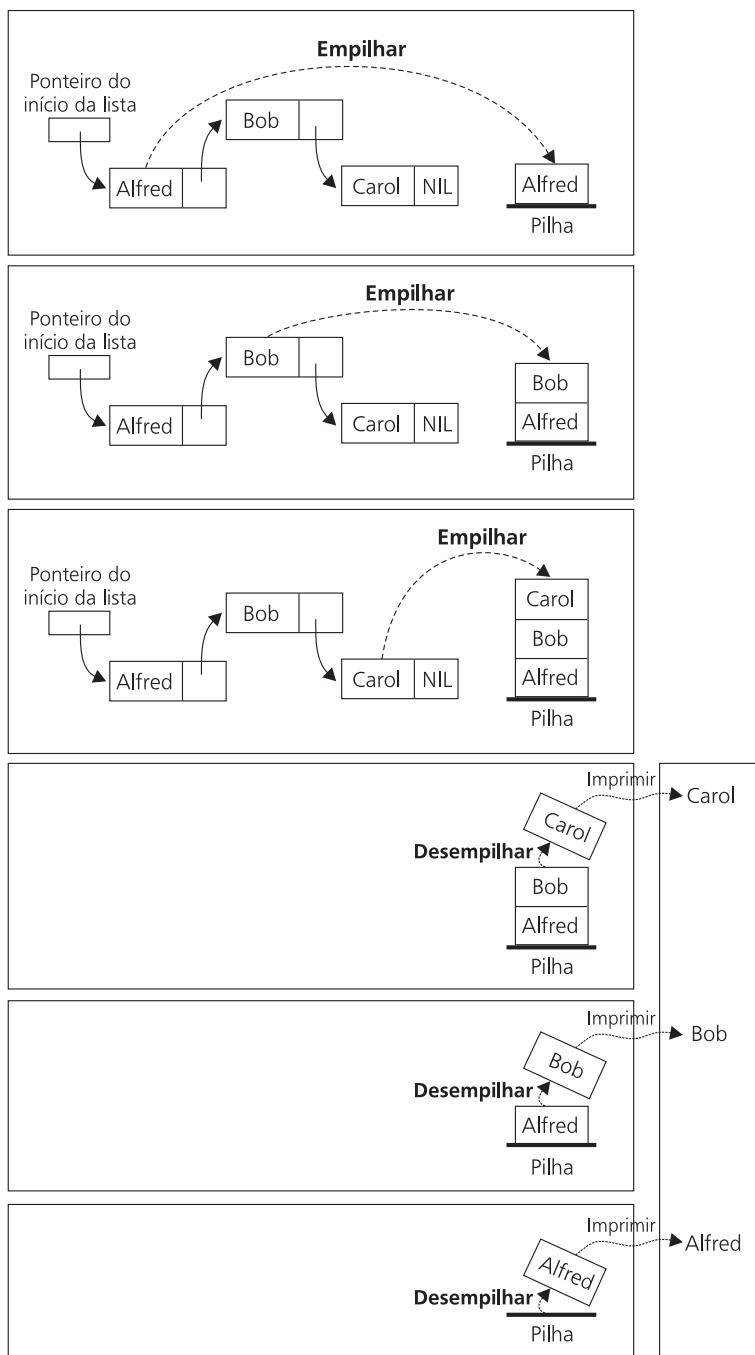


Figura 7.10 Uso de uma pilha para imprimir, em ordem invertida, uma lista ligada.

Como observamos no caso de listas, um programador certamente acharia vantajoso escrever procedimentos para realizar essas operações de empilhamento e desempilhamento de tal forma que a pilha pudesse ser usada como ferramenta abstrata. Note que esses procedimentos devem tratar os casos especiais, como a tentativa de retirar um elemento de uma pilha vazia e de acrescentar um elemento a uma pilha lotada. De fato, um sistema completo de pilha provavelmente conterá procedimentos para empilhar, desempilhar e testar se a pilha está vazia ou lotada.

Uma pilha assim organizada apresenta poucas diferenças entre as estruturas conceitual e real dos dados na memória. Entretanto, suponhamos que não seja possível prever o tamanho máximo da pilha, portanto, não poderemos reservar um bloco fixo de memória com certeza de que a pilha caberá inteira nele. Neste caso, uma solução possível é implementar a pilha como uma estrutura ligada, semelhante à discutida na Seção 7.2. Isso elimina a necessidade de restringi-la a um bloco de tamanho fixo, uma vez que os elementos podem ser armazenados em qualquer pequena área disponível na memória. Em tal situação, a estrutura conceitual da pilha será bem diferente do arranjo real dos dados na memória.

```

procedure ImprimeInvertido (Lista)
PonteiroCorrente ← Ponteiro para o início de Lista
while (PonteiroCorrente não for NIL) do
    (Empilhar o nome do elemento apontado pelo PonteiroCorrente;
     Obter o valor contido no ponteiro do elemento de Lista, apontado
     pelo PonteiroCorrente, e reatribuir a PonteiroCorrente esse valor)
while (a pilha não estiver vazia) do
    (Desempilhar um nome da pilha e imprimi-lo)

```

Figura 7.11 Um procedimento (usando uma pilha auxiliar) para imprimir uma lista ligada em ordem invertida.

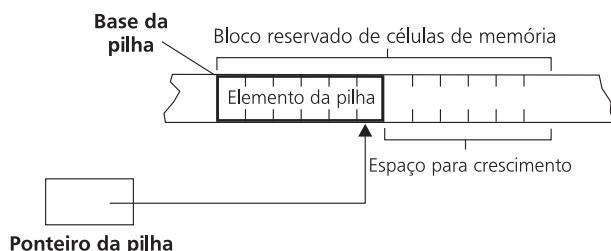


Figura 7.12 Uma pilha na memória.



QUESTÕES/EXERCÍCIOS

1. Cite outras ocorrências de pilhas na vida cotidiana.
2. Suponha que um programa principal chame o procedimento A, que, por sua vez, chame o procedimento B. Quando B termina, o procedimento A chama o procedimento C. Siga esta seqüência, observando a dinâmica da pilha de endereços de retorno.
3. Com base na técnica apresentada nesta seção para implementar uma pilha em um bloco contíguo de células, qual a condição indicativa de que a pilha está vazia?
4. Projete um procedimento para retirar um elemento de uma pilha, sendo esta implementada usando um ponteiro para o topo da pilha. Seu procedimento deverá imprimir uma mensagem de erro se a pilha estiver vazia.
5. Descreva como uma pilha pode ser implementada em uma linguagem de alto nível, em termos de uma matriz unidimensional.

7.5 Filas

Ao contrário da pilha, na qual as inserções e remoções são executadas pela mesma extremidade, uma fila determina que todas as inserções sejam realizadas por uma extremidade, enquanto todas as remoções o são pela outra. Já analisamos anteriormente esta estrutura para as filas de espera do Capítulo 3, ocasião em que verificamos ser um esquema de armazenamento do tipo *first-in, first-out* (FIFO)^{*}. De fato, o conceito de fila é inerente a qualquer sistema no qual objetos sejam retirados na mesma ordem em que chegaram. As extremidades da fila são nomeadas de acordo com a analogia das filas de espera da vida real. A extremidade pela qual são removidos os elementos é denominada **início da fila**,^{**} e aquela pela qual novos elementos são inseridos, **fim da fila**.^{***}

Podemos implementar uma fila na memória de um computador com um bloco de células contíguas de memória, semelhante ao que utilizamos para armazenar a pilha. Executamos operações nas duas extremidades da estrutura, reservando para isto duas posições de memória, de modo a ter dois ponteiros em lugar de um só, como acontecia no caso da pilha. Designamos um deles para indicar o **início**, e outro, para indicar o **final da fila**. Começamos com uma fila vazia, apontando tais ponteiros para uma mesma posição (Figura 7.13). Todas as vezes que um elemento for inserido, iremos depositá-lo na posição apontada pelo ponteiro para o final da fila, e então ajustaremos o ponteiro para a próxima posição livre do bloco. Desta maneira, o ponteiro para o final da fila estará sempre apontando para o primeiro espaço vazio, após o final da fila. Para remover um elemento, extraímos o objeto que ocupa a posição apontada pelo ponteiro para o início da fila, o qual deve ser então corrigido, de forma que passe a apontar para o elemento que se encontrava logo após aquele que foi removido.

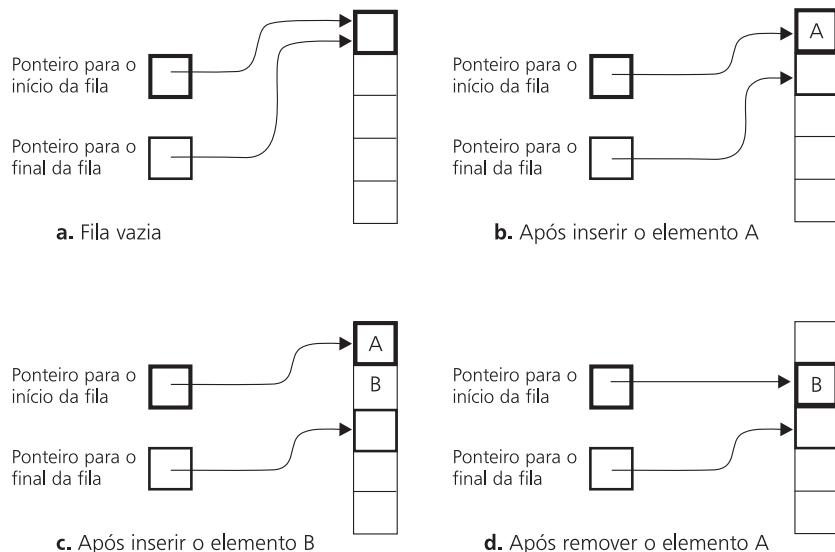


Figura 7.13 Uma fila, implementada com ponteiros para o início e final da mesma.

*N. de T. Em português, estrutura em que o primeiro a entrar é o primeiro a sair.

**N. de T. Em inglês, *head, front of the queue*.

***N. de T. Em inglês, *tail, rear of the queue*.

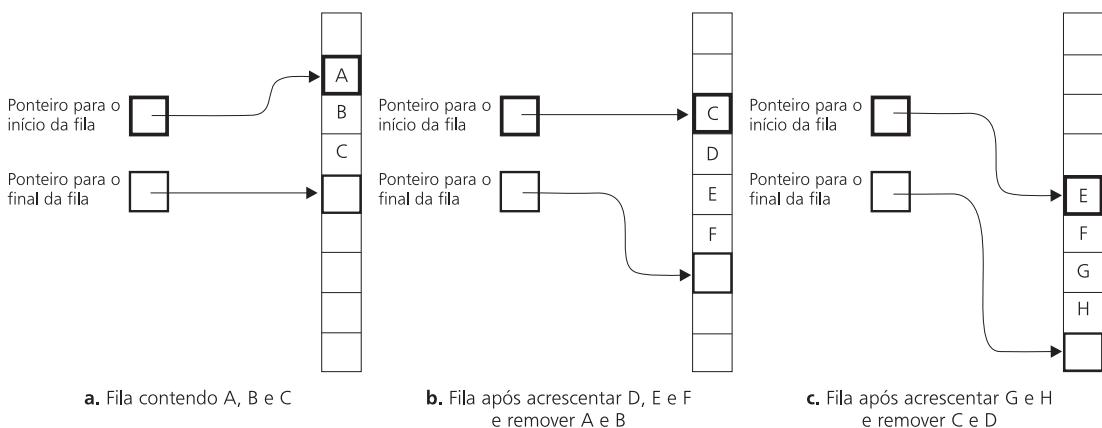


Figura 7.14 Uma fila que se move lentamente pela memória.

Resta ainda um problema neste esquema de armazenamento. Se deixada sem verificação, a fila se moverá lentamente pela memória como uma geleira, destruindo qualquer outro dado que estiver em seu caminho (Figura 7.14). Este movimento é resultado de uma política egocêntrica de inserir cada novo elemento simplesmente colocando-o ao lado do anterior e reposicionando adequadamente o ponteiro que indica o final da fila. Se acrescentarmos muitos elementos, o final da fila se estenderá, em última instância, até ocupar toda a memória da máquina.

Esta avidez por memória não é resultado do tamanho da fila, mas um efeito colateral de sua implementação. (Uma fila pequena, porém ativa, pode solicitar mais recursos de memória de uma máquina do que uma fila grande, mas inativa.) Uma solução para este problema de espaço de memória é deslocar para frente os elementos da fila, à medida que os elementos dianteiros forem removidos, da mesma maneira como as pessoas que esperam em fila para comprar entradas para o teatro dão um passo adiante a cada cliente atendido. Contudo, essa estratégia seria ineficiente no computador, uma vez que exigiria movimento em massa dos dados.

Uma solução comum para este dilema consiste em reservar um bloco de memória para a fila, começar a fila em uma das extremidades do bloco e deixá-la migrar até a outra extremidade. Quando o final da fila atingir o final do bloco, simplesmente inseriremos os elementos adicionais a partir da extremidade original do bloco, que, a esta altura, já deverá estar vazia. Por analogia, quando o último elemento do bloco finalmente se tornar o início da fila e for removido, ajustaremos o ponteiro para o início da

Um problema com os ponteiros

Da mesma forma que o uso de fluxogramas leva a projetos confusos de algoritmos (Capítulo 4), e o uso irrestrito de instruções *goto* resulta em programas mal projetados (Capítulo 5), o uso indisciplinado de ponteiros tem produzido estruturas de dados desnecessariamente complexas e sujeitas a erro. Para tentar organizar esse caos, muitas linguagens de programação restringem a flexibilidade dos ponteiros. Por exemplo, a Java não permite a existência de ponteiros em sua forma genérica. Em vez disso, permite apenas uma forma restrita, chamada referência. Uma distinção é que as referências não podem ser modificadas por operações aritméticas. Por exemplo, se um programador Java desejar avançar a referência Próximo para o próximo elemento de um lista contígua, ele usará uma instrução equivalente a

redirecionar Próximo ao próximo elemento da lista enquanto um programador C usaria uma instrução equivalente a

atribuir a Próximo o valor Próximo + 1.

Note que a instrução Java reflete melhor o objetivo subjacente. Além disso, para executá-la, é preciso que haja um outro elemento na lista, mas se Próximo já estiver apontando para o último elemento, a instrução C fará com que aponte para alguma coisa fora da lista – um erro comum tanto para o iniciante como para um programador C experiente.

fila, de modo que aponte para o início do bloco, onde outros elementos já deverão estar aguardando. Desta maneira, a fila circula em torno de si mesma, dentro do bloco, em vez de se expandir memória afora.

Essa técnica resulta na implementação denominada **fila circular**, pois o processo se baseia na utilização cíclica do bloco de posições de memória reservado para a fila (Figura 7.15). No que se refere à fila, a última célula do bloco reservado a ela é considerada adjacente à sua primeira célula.

Novamente, devemos observar a diferença entre a estrutura conceitual visualizada pelo usuário de uma fila e a estrutura cíclica real implementada na memória da máquina. Como nas estruturas anteriores, essas diferenças são superadas com o auxílio do software: juntamente com o conjunto de células de memória utilizado para o armazenamento de dados, a implementação de fila inclui um conjunto de procedimentos que inserem e removem elementos na fila, além de detectar se ela está vazia ou lotada. Então, um programador que trabalhe em outra unidade de programa pode solicitar a inserção ou remoção de elementos por meio desses procedimentos, sem se preocupar com os detalhes de como a fila está implementada na memória.

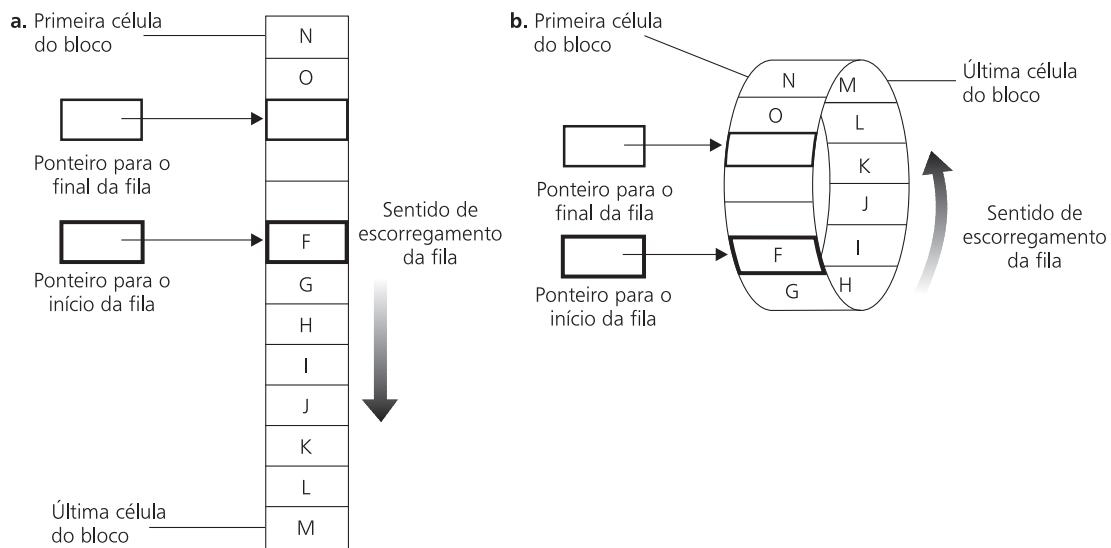


Figura 7.15 Uma fila circular (a) contém as letras de F a O como realmente estão armazenados na memória e (b) em sua forma conceitual, na qual a última célula no bloco é “adjacente” à primeira célula.



QUESTÕES/EXERCÍCIOS

- Usando papel e lápis, mantenha um registro da estrutura de fila circular descrita nesta seção durante o seguinte cenário (assuma a hipótese de que o bloco reservado para a fila tem capacidade para armazenar quatro elementos):

Inserir o elemento A.

- Inserir o elemento B.
 - Inserir o elemento C.
 - Remover um elemento.
 - Remover um elemento.
 - Inserir o elemento D.
 - Inserir o elemento E.
 - Remover um elemento.
 - Inserir o elemento F.
 - Remover um elemento.
2. Quando uma fila é implementada em forma circular, conforme descrito nesta seção, qual a relação entre os ponteiros para o início e para o final da fila? E quando a fila estiver lotada? Como descobrir se uma fila está lotada ou vazia?
3. Escreva um procedimento para inserir um elemento em uma fila circular.

7.6 Árvores

A última estrutura de dados que analisaremos é a **árvore**, que é a estrutura refletida pelo diagrama organizacional de uma empresa (Figura 7.16). Aqui, o presidente é representado no topo, e dele partem linhas descendentes, que se ramificam até os vice-presidentes, os quais são seguidos pelos gerentes regionais e assim por diante. A esta definição intuitiva de uma estrutura em árvore impomos uma restrição adicional, a de que (em termos de quadro organizacional) nenhum indivíduo da companhia se reporta a dois superiores diferentes, isto é, os diferentes ramos da empresa não se fundem em níveis inferiores.

Cada posição da árvore é chamada **nó** (Figura 7.17). O único nó do topo é chamado **nó-raiz** (uma vez que, com o desenho invertido, este nó representaria a base ou a raiz da árvore). Os nós na outra extremidade são chamados **nós terminais** (ou nós-folha). Se nos posicionarmos em qualquer nó de uma árvore, verificaremos que este nó, juntamente com os demais nós a ele ligados, possui, por sua vez, a estrutura de uma árvore. Chamamos estas estruturas menores de **subárvores**. Às vezes, nos referimos a estruturas de árvore como se cada nó gerasse outros que ficasse imediatamente abaixo dele. Mencionamos freqüentemente os ancestrais de um nó ou os seus descendentes. Referimo-nos a seus descenden-

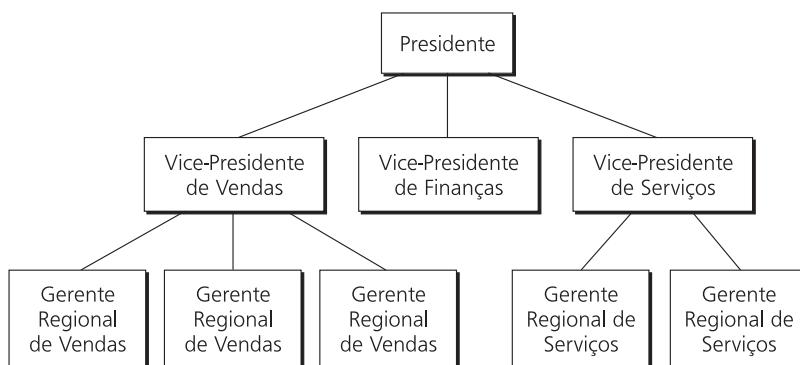


Figura 7.16 Um exemplo de diagrama organizacional.

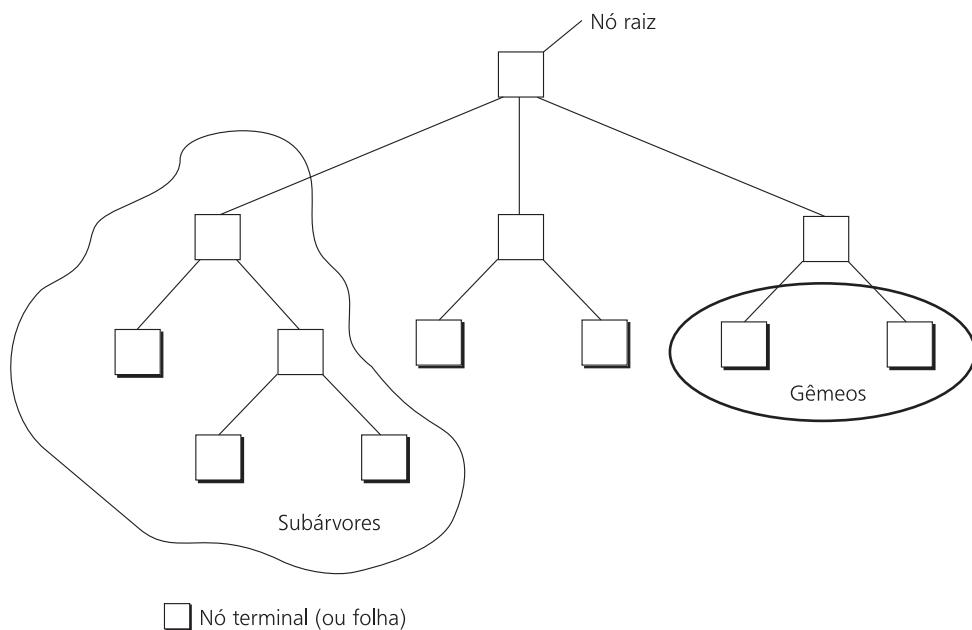


Figura 7.17 Terminologia de árvores.

tes imediatos como seus **filhos** e a seu ancestral imediato como seu **pai**^{*}. Além disso, referimo-nos a dois ou mais nós descendentes do mesmo pai como **gêmeos**^{**}. Finalmente, mencionamos com freqüência a **profundidade**^{***} de uma árvore, que é o número de nós compreendidos na mais longa trajetória da raiz até uma folha. Em outras palavras, a profundidade de uma árvore é o número de camadas horizontais nella compreendidas.

Encontraremos diversas vezes as estruturas de árvore nos capítulos subsequentes, por isso, não apresentaremos qualquer aplicação no momento. Mais adiante, nesta seção, e também no nosso estudo sobre organização indexada, no Capítulo 8, verificaremos que uma informação que deva ser obtida com grande rapidez, na consulta a bancos de dados, muitas vezes é organizada em árvore. No Capítulo 10, verificaremos como os jogos eletrônicos podem ser analisados com o auxílio de árvores.

Implementação de árvores

Com o propósito de discutir as técnicas de armazenamento de árvores, restringiremos a nossa atenção às **árvores binárias**, nas quais cada nó tem dois filhos no máximo. Tais árvores geralmente são armazenadas na memória usando uma estrutura semelhante à das listas ligadas. Em vez de cada elemento conter dois componentes (o dado e um ponteiro para o próximo elemento), cada elemento (ou nó) da árvore binária contém três: (1) o dado, (2) um ponteiro para o primeiro e (3) outro ponteiro para o segundo filho do nó. Embora não haja um lado esquerdo ou direito na memória de uma

^{*}N. de T. Em inglês, usa-se o termo mais genérico *parent* (significa genitor).

^{**}N. de T. Ou ainda, *irmãos*. Em inglês, *twins* ou *siblings*.

^{***}N. de T. Alguns autores preferem o termo *altura* em vez de *profundidade*. Em inglês, *depth*.

máquina, é útil visualizar o primeiro deles como sendo o **ponteiro para o filho esquerdo** e o outro como o **ponteiro para o filho direito**, em referência ao modo como a árvore seria desenhada no papel. Cada nó da árvore é representado por um pequeno bloco contíguo de células de memória, com o formato descrito na Figura 7.18.

Armazenar uma árvore na memória envolve encontrar blocos disponíveis de células de memória para armazenar os nós e suas ligações de acordo com a estrutura desejada da árvore. Assim, cada ponteiro apontará para o filho esquerdo ou direito do nó correspondente, ou então será associado ao valor NIL, caso não haja mais nós nesta direção da árvore. Isto significa que, em um nó terminal, os dois ponteiros estão associados a NIL. Finalmente, reservamos uma posição especial de memória para armazenar o endereço do nó-raiz, chamado **ponteiro para a raiz** da árvore. É ele que provê o acesso inicial à árvore.

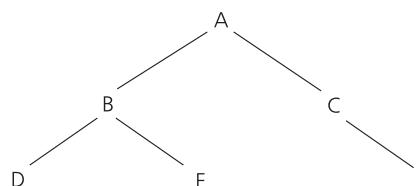
Um exemplo deste esquema de armazenamento ligado está ilustrado na Figura 7.19, em que uma estrutura conceitual de árvore binária é apresentada juntamente com uma representação de como tal árvore seria implementada de fato na memória de um computador. Note que a disposição dos nós na memória principal pode ser bem diferente do arranjo conceitual da árvore. Na verdade, os nós podem estar espalhados em uma grande área de memória. Contudo, sempre encontraremos o nó-raiz por meio do ponteiro para a raiz da árvore, podendo então traçar qualquer trajetória descendente sobre a árvore, seguindo os ponteiros apropriados de um nó para outro.

Uma alternativa ao sistema de armazenamento ligado, para árvores binárias, é a técnica de reservar um bloco de células contíguas de memória, armazenar o nó-raiz na primeira destas células (para



Figura 7.18 A estrutura de um nó em uma árvore binária.

Árvore conceitual



Organização real do armazenamento

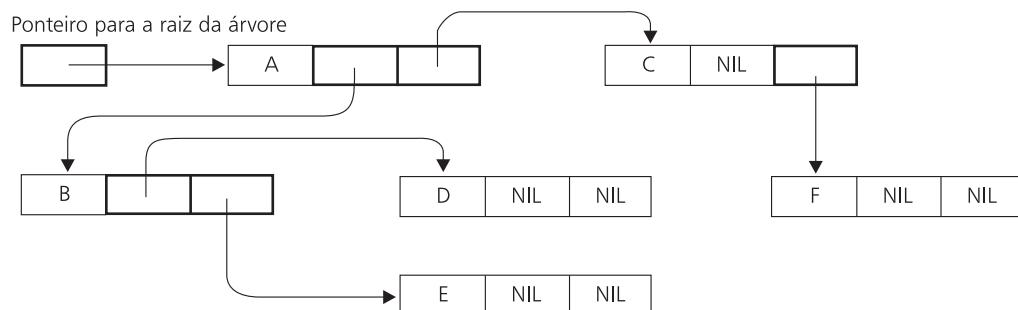


Figura 7.19 A organização conceitual e real de uma árvore binária com a utilização de um sistema de armazenamento ligado.

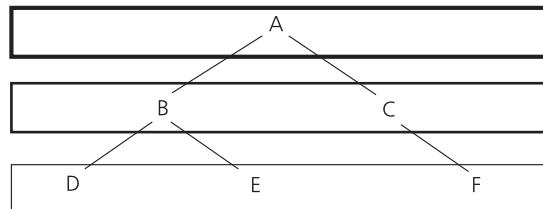
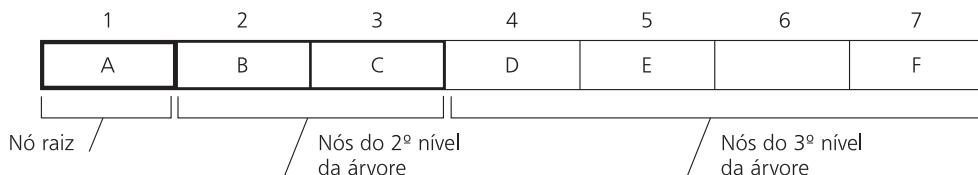
Árvore conceitual**Organização real do armazenamento**

Figura 7.20 Uma árvore armazenada sem o uso de ponteiros.

facilitar, pressupomos que cada nó da árvore exige somente uma posição de memória), armazenar o filho esquerdo da raiz na segunda posição, o filho direito da raiz na terceira posição e, de modo geral, armazenar os filhos à esquerda e à direita do nó correspondente à célula n , nas células $2n$ e $2n+1$, respectivamente. As células internas do bloco não utilizadas pela estrutura de árvore em construção são marcadas com um padrão de bits próprio, que indica a ausência de dados. A Figura 7.20 mostra como uma árvore

Coleta de lixo

À medida que as estruturas dinâmicas crescem e encolhem, o espaço de armazenamento é usado e reusado. O processo de reivindicar espaço inutilizado para uso futuro é conhecido como **coleta de lixo**. Ele é necessário em diversos contextos. O gerente de memória em um sistema operacional deve realizar a coleta de lixo quando aloca e recupera o espaço de memória.

O gerente de arquivos faz a coleta de lixo à medida que arquivos vão sendo criados e eliminados no sistema de armazenamento em massa da máquina.

Além disso, qualquer processo em execução sob controle de um despachante pode necessitar fazer a coleta de lixo dentro de seu próprio espaço de memória.

A coleta de lixo envolve alguns problemas sutis. No caso de estruturas ligadas, cada vez que um ponteiro para um item de dados é alterado, o coletores de lixo deve decidir se o espaço de armazenamento para o qual o ponteiro originalmente apontava deve ser reivindicado.

O problema se torna especialmente complexo em estruturas interligadas que envolvem muitos caminhos por ponteiros. Rotinas de coleta de lixo imperfeitas podem levar à perda de dados ou a um uso ineficiente do espaço de armazenamento. Mais especificamente, se a coleta de lixo falhar na reivindicação do espaço de armazenamento, este irá lentamente minguar, fenômeno conhecido como **vazamento de memória**.

seria armazenada utilizando-se essa técnica. Note-se que a técnica consiste, essencialmente, em armazenar os nós em níveis sucessivamente mais baixos da árvore, como segmentos, um após o outro. Dessa forma, o primeiro elemento do bloco é o nó-raiz, seguido pelos filhos da raiz, os quais, por sua vez, vêm seguidos pelos netos da raiz e assim por diante.

Ao contrário da estrutura ligada anteriormente descrita, esse sistema alternativo de armazenamento apresenta um método conveniente para a localização do ancestral ou dos irmãos de qualquer nó. (Naturalmente, isto pode ser feito na estrutura ligada, mediante o uso de ponteiros adicionais.) De fato, a posição do ancestral de um nó pode ser encontrada dividindo por 2 a posição do nó no bloco e descartando o resto da divisão (o ancestral do nó da posição 7 é o nó da posição 3), e o irmão de um nó pode ser encontrado somando 1 à posição do nó quando esta for par ou subtraíndo 1 quando for ímpar (o irmão do nó da posição 4 é o da posição 5, enquanto o irmão do nó da posição 3 é o da posição 2). Além disso, este esquema de armazenamento fará uso eficiente de espaço, caso as árvores binárias estejam relativamente balanceadas (no sentido de as subárvore tenderem a apresentar a mesma profundidade) e cheias (no sentido de não possuírem ramificações longas e finas). Para árvores que não atendam a essas especificações, entretanto, esse esquema pode se mostrar bastante ineficiente, como ilustra a Figura 7.21.

Um pacote para árvores binárias

Como no caso das outras estruturas que estudamos, é vantajoso isolar as tecnicidades da implementação de uma árvore das outras partes de um sistema de *software*. Em consequência, um programador normalmente identifica as atividades que serão realizadas na árvore, escreve os procedimentos para implementá-las e então usa-os para ter acesso à árvore de outras partes do programa. Assim, esses procedimentos, juntamente com a área de armazenamento, constituem um pacote que é usado como ferramenta abstrata.

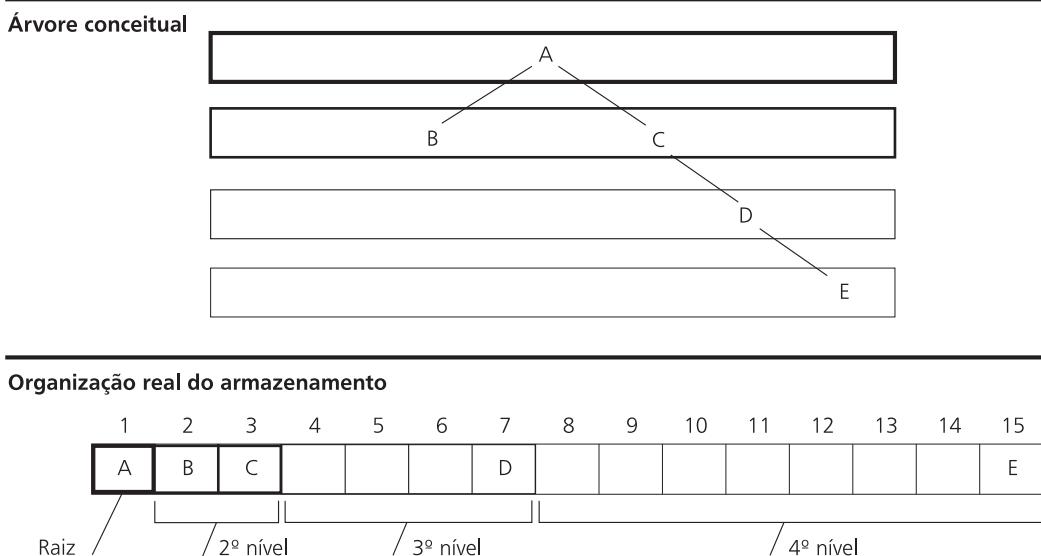


Figura 7.21 Uma árvore esparsa e balanceada, apresentada em sua forma conceitual e na forma como poderia ser armazenada sem ponteiros.

Para demonstrar este pacote, retomemos o problema de armazenar em ordem alfabética uma lista de nomes. Suponhamos que as operações a serem executadas nesta lista sejam:

localizar a presença de um elemento,
imprimir a lista em ordem alfabética, e
inserir um novo elemento

Nossa meta é desenvolver um sistema de armazenamento acoplado a um conjunto de procedimentos para a execução de tais operações.

Comecemos considerando as opções existentes, relativas ao procedimento de busca na lista. Se a lista for armazenada de acordo com o modelo de lista ligada da Seção 7.3, seremos obrigados a percorrê-la de forma seqüencial, processo este que seria muito ineficiente caso tal lista viesse a se tornar longa, conforme foi discutido no Capítulo 4. Logo, a opção seguinte para o nosso procedimento de busca seria uma implementação baseada no algoritmo de busca binária (Capítulo 4). Para aplicar este algoritmo, nosso esquema de armazenamento deve localizar o elemento central de partes sucesivas da lista, cada vez menores. Tal operação é possível com o uso de uma lista contígua, uma vez que podemos, da mesma forma como calculamos as posições dos elementos em um vetor, calcular o endereço do elemento central. Contudo, utilizar uma lista contígua apresenta seus problemas nos casos de inserção, conforme esclarece a Seção 7.3.

Nosso problema pode ser resolvido implementando a lista na forma de uma árvore binária, em vez de utilizar um dos sistemas de lista tradicionais. Associamos o elemento central da lista ao nó-raiz, o elemento central da primeira metade da lista ao filho esquerdo da raiz, e o elemento central da segunda metade ao filho direito. Os elementos centrais de cada quarta parte da lista se tornam os filhos dos filhos do nó-raiz, e assim sucessivamente. Por exemplo, segundo este processo, a árvore da Figura 7.22 pode representar a seguinte lista de letras: A, B, C, D, E, F, G, H, I, J, K, L e M. (Consideraremos como elemento central o maior dos dois elementos medianos quando a parte da lista em questão contém um número par de elementos.)

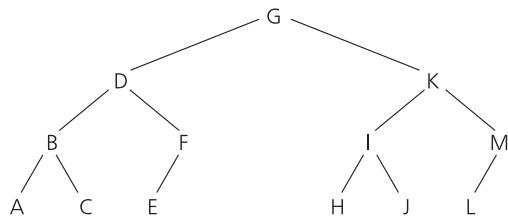


Figura 7.22 As letras de A até M dispostas em uma árvore ordenada.

```

procedure Busca(Arvore,ValorDesejado)
  if (ponteiro do nó-raiz = NIL)
    then
      (declarar que a busca fracassou)
    else
      (executar o bloco de instruções abaixo,
       associado com o caso apropriado)
      caso 1: ValorDesejado = valor do nó-raiz
        (Declarar que a busca foi bem sucedida)
      caso 2: ValorDesejado < valor do nó-raiz
        (Aplicar o procedimento Busca para
         verificar se o ValorDesejado se encontra
         na subárvore identificada pelo ponteiro do
         filho da esquerda da raiz e reportar o resultado)
      caso 3: ValorDesejado > valor do nó-raiz
        (Aplicar o procedimento Busca para
         verificar se o ValorDesejado se encontra
         na subárvore identificada pelo ponteiro do
         filho da direita da raiz e reportar o resultado)
  ) end if
  
```

Figura 7.23 A busca binária aplicada a uma lista implementada como árvore binária ligada.

Para pesquisar uma lista armazenada desta maneira, compararmos o valor desejado com o nó-raiz. Se os dois forem iguais, a nossa busca teve sucesso. Se não, deslocamo-nos para o filho esquerdo ou direito da raiz, dependendo de o elemento desejado ser menor ou maior do que a raiz, respectivamente. Lá encontraremos o elemento central dessa parte da lista, necessário ao prosseguimento da busca. Este processo de comparação e deslocamento em direção a um dos filhos continua até que o elemento desejado seja encontrado (o que significa que nossa busca teve êxito) ou se atinja o final da árvore, sem encontrá-lo (o que significa que a busca fracassou).

A Figura 7.23 apresenta uma forma como este processo de busca pode ser expresso, para o caso de uma estrutura ligada. Note que a Figura 7.23 é sim-

plesmente um refinamento do procedimento mostrado na Figura 4.14, que é a nossa versão original da busca binária. A distinção é superficial. Enquanto a nossa imagem original da busca binária era a de considerar sucessivamente menores segmentos da lista, nossa imagem agora é a de considerar sucessivamente menores subárvores (Figura 7.24).

Como alteramos a ordem sequencial natural da nossa lista em favor de uma busca mais eficiente, poder-se-á supor que o processo de impressão da lista em ordem alfabética tenha sido dificultado. Contudo, isso não ocorre. Para imprimir a lista em ordem alfabética, precisamos apenas imprimir a subárvore esquerda, em ordem alfabética, o nó-raiz e a subárvore direita, também em ordem alfabética (Figura 7.25). Afinal de contas, a subárvore esquerda contém os elementos menores do que o nó-raiz, enquanto a direita contém os maiores do que a raiz. Um esboço da nossa rotina de impressão seria:

```
if (árvore não está vazia)
then (imprimir a subárvore esquerda, em ordem alfabética;
      imprimir o nó-raiz;
      imprimir a subárvore direita, em ordem alfabética)
```

Pode-se questionar se este esboço nos aproxima do nosso objetivo de desenvolver um procedimento completo de impressão, pois envolve as tarefas de impressão das subárvores esquerda e direita, em ordem alfabética, ambas idênticas à nossa tarefa original. Entretanto, imprimir uma subárvore é uma tarefa menor do que fazer o mesmo com a árvore inteira. Isto é, resolver o problema de imprimir uma árvore envolve a tarefa menor de imprimir subárvores, o que sugere uma abordagem recursiva ao nosso problema de impressão.

Seguindo este raciocínio, podemos completar o nosso esboço, em pseudocódigo, do procedimento para imprimir nossa árvore em ordem alfabética, conforme mostra a Figura 7.26. Criamos a rotina `ImprimeArvore`, a qual solicita os serviços de `ImprimeArvore` para imprimir as suas subárvores à esquerda e à direita. A condição terminal do processo recursivo (quando é encontrada uma subárvore vazia) é garantida, pois a cada ativação a rotina opera sobre uma árvore menor que aquela que causou a sua ativação.

A inserção de um novo elemento na árvore também é mais fácil do que parece à primeira vista. Pode-se imaginar que determinadas inserções exijam cortes na

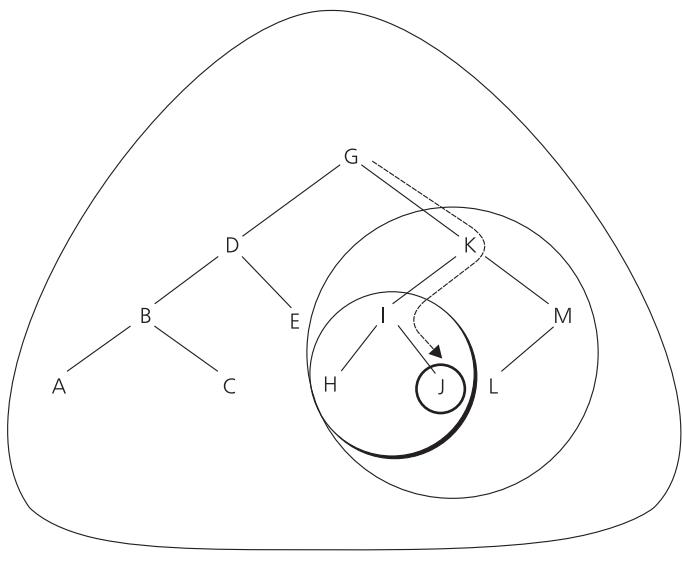


Figura 7.24 As árvores cada vez menores consideradas pelo procedimento da Figura 7.23 à procura da letra J.

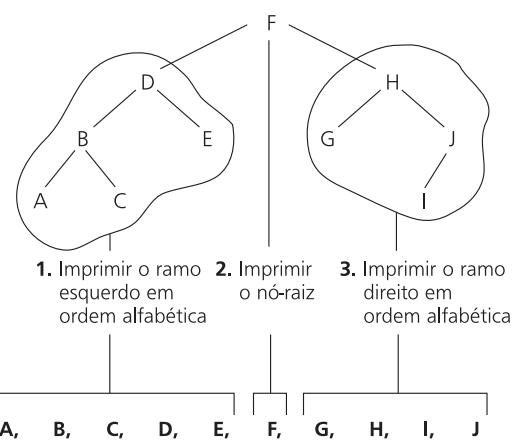


Figura 7.25 Impressão, em ordem alfabética, de uma árvore de busca.

```

procedure ImprimeArvore (Arvore)
if (Arvore não estiver vazia)
  then (Executar o procedimento ImprimeArvore à
        subárvore que aparece como ramo esquerdo
        de Arvore;
        Imprimir o nó-raiz de Arvore;
        Aplicar o procedimento ImprimeArvore à
        subárvore que aparece como o ramo direito
        de Arvore)

```

Figura 7.26 Um procedimento para imprimir os dados de uma árvore binária em ordem alfabética.

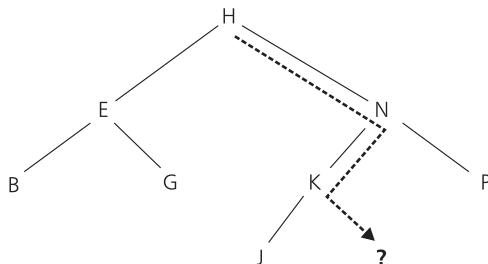
inserido (chamado NovoValor) e acrescenta na posição correta o novo nó-folha que contém o NovoValor. Note que se o elemento a ser inserido for encontrado durante a busca, nenhuma inserção será feita.

Verificamos, então, que um pacote de software constituído de uma árvore ligada, associada aos nossos procedimentos de busca, impressão e inserção, compõe um sistema completo que pode ser usado como ferramenta abstrata pela aplicação hipotética.

árvore, a fim de permitir a entrada de um novo elemento, mas na verdade o novo nó sempre pode ser colocado como folha no final da árvore, apesar de seu valor. Para encontrar a posição correta para essa nova folha, percorremos a mesma trajetória utilizada para efetuar a busca do valor a ser inserido. Uma vez que tal valor não se encontra na árvore, nossa busca levará a um ponteiro vazio. Aí estará localizada a posição correta para o novo nó (Figura 7.27). De fato, teremos encontrado a posição aonde uma operação de busca do novo elemento nos levaria.

Um procedimento que expressa tal processo sobre uma estrutura de árvore ligada está mostrado na Figura 7.28. Ele percorre a árvore em busca do valor a ser

a. Buscar o novo elemento até que seja constatada a sua ausência



b. Esta é a posição na qual o novo elemento deve ser colocado

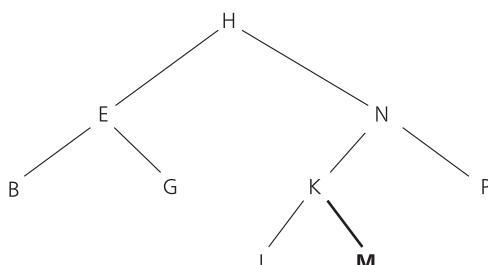


Figura 7.27 Inserção do elemento M na lista B, E, G, H, J, K, N, P armazenada como árvore.

```

procedure Insere(Arvore,NovoValor)
  if (Ponteiro do nó-raiz = NIL)
    then (ajustar o ponteiro do nó-raiz para apontar
          para uma nova folha que contém NovoValor)
    else (executar o bloco de instruções abaixo,
          associado com o caso apropriado)
      caso 1: NovoValor = valor do nó-raiz
              (Nada a fazer)
      caso 2: NovoValor < valor do nó-raiz
              (if (ponteiro para o filho da esquerda do nó-raiz = NIL)
                  then (ajustar esse ponteiro para apontar para
                        uma nova folha que contém o NovoValor)
                  else (aplicar o procedimento Insere para
                        instalar o NovoValor na subárvore
                        identificada pelo ponteiro para o
                        filho da esquerda))
      caso 3: NovoValor > valor do nó-raiz
              (if (ponteiro para o filho da direita do nó-raiz = NIL)
                  then (ajustar esse ponteiro para apontar para
                        uma nova folha que contém o NovoValor)
                  else (aplicar o procedimento Insere para instalar
                        o NovoValor na subárvore identificada pelo ponteiro
                        para o filho da direita))
  ) end if

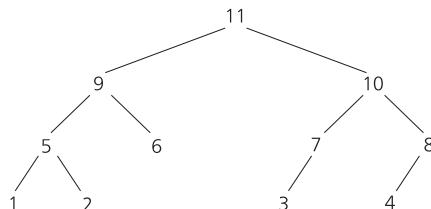
```

Figura 7.28 Um procedimento para inserir um elemento em uma lista armazenada como árvore binária.

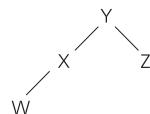


QUESTÕES/EXERCÍCIOS

- Identifique a raiz e os nós de folha da árvore seguinte. Identifique as subárvore abaixo do nó 9. Identifique os grupos de filhos dentro da árvore.



- Qual condição indica que uma árvore ligada, implementada na memória de uma máquina, está vazia?
- Faça um diagrama que represente como a árvore aparece na memória, quando armazenada com o uso de ponteiros para os filhos à esquerda e à direita, conforme foi descrito nesta seção. A seguir, faça um outro diagrama que mostre como tal árvore seria em armazenamento contíguo, utilizando o esquema de armazenamento alternativo, também apresentado nesta seção.



4. Faça uma estrutura em árvore binária para armazenar a lista R, S, T, U, V, W, X, Y e Z para futuras buscas.
5. Indique o caminho percorrido pelo algoritmo de busca binária da Figura 7.23 quando aplicado à árvore da Figura 7.22 para a busca do elemento J. E para o elemento P?
6. Faça um diagrama que represente o estado das ativações do algoritmo recursivo de impressão de árvores da Figura 7.26 na ocasião em que é impresso o nó K da árvore ordenada da Figura 7.22.
7. Descreva como uma estrutura de árvore na qual cada nó pode ter até 26 folhas poderia ser usada para codificar o processo de soletrar palavras na língua inglesa.

7.7 Tipos de dados personalizados

No Capítulo 5, introduzimos o conceito de tipos de dados e discutimos os tipos básicos, como inteiro, real, caractere e booleano, que a maioria das linguagens de programação oferece como primitivos. Nesta seção, analisaremos os meios pelos quais um programador pode definir tipos de dados próprios, que mais correspondam às necessidades de suas aplicações.

Tipos definidos pelo usuário

A tarefa de expressar um algoritmo em geral é mais conveniente se existem outros tipos de dados, além dos primitivos oferecidos pelas linguagens de programação. Por isso, linguagens de programação mais modernas permitem aos programadores definir tipos de dados adicionais, utilizando os tipos primitivos e as estruturas como blocos construtivos. Estes tipos de dados “feitos em casa” são conhecidos como **tipos definidos pelo usuário**.

Para ilustrar, suponha que desenvolvemos um programa com numerosas variáveis, cada qual com a mesma estrutura heterogênea constituída de nome, idade e nível de qualificação. Poder-se-ia redeclarar a composição da estrutura todas as vezes que fosse necessário fazer referência a tal estrutura. Por exemplo, para definir a variável Funcionario com esta estrutura heterogênea, um programador C escreveria:

```
struct
{char Nome[8];
 int Idade;
 float NivelDeQualificacao;
} Funcionario;
```

conforme foi apresentado na Figura 5.6 do Capítulo 5.

Um problema desta abordagem é que o programa pode ficar volumoso e difícil de ler, caso tal estrutura seja muito freqüente. Além disso, sem um exame profundo, é difícil verificar que todas as estruturas são idênticas. Um método mais adequado consiste em declarar a estrutura heterogênea como um novo tipo (definido pelo usuário) e então usá-lo como se fosse primitivo.

Encontramos um exemplo típico dessa idéia na linguagem de programação C, onde novos tipos podem ser declarados com a instrução `typedef` (abreviatura de *type definition*). Ela consiste na palavra reservada `typedef` seguida da descrição estrutural do novo tipo, seguida do nome que será usado nas referências ao novo tipo. Assim, a instrução

```
typedef struct
{char Nome[8];
 int Idade;
 float NivelDeQualificacao;
} TipoFuncionario;
```

define um novo tipo, `TipoFuncionario`, que consiste na estrutura heterogênea com um nome, uma idade e um `NivelDeQualificacao`. Esse novo tipo pode ser usado para declarar variáveis da mesma forma como um tipo primitivo. Por exemplo, a variável `Funcionario` pode ser declarada com a instrução

```
TipoFuncionario Funcionario;
```

As vantagens de tal tipo definido pelo usuário são mais sentidas quando são declaradas diversas variáveis. Da mesma forma que um programador de linguagem C pode declarar as variáveis `Manga`, `Cintura` e `Pescoco`, como do tipo primitivo real, com a instrução

```
float Manga, Cintura, Pescoco;
```

a instrução

```
TipoFuncionario Gerente, Vendedor1, Vendedor2;
```

declara as três variáveis `Gerente`, `Vendedor1` e `Vendedor2` como do `TipoFuncionario`.

É importante distinguir entre um tipo definido pelo usuário e um item de dados desse tipo. O último se refere a uma **instância** do tipo em questão. Um tipo definido pelo usuário é essencialmente um modelo usado para construir instâncias de um determinado tipo. Ele descreve as características que todas as instâncias deste tipo devem assumir, mas não constitui, ele próprio, uma ocorrência real deste tipo (como uma forma para cortar biscoitos é um modelo a partir do qual os biscoitos podem ser feitos, mas não é biscoito). No exemplo anterior, o tipo definido pelo usuário `TipoFuncionario` foi usado para construir três instâncias daquele tipo, conhecidas como `Gerente`, `Vendedor1` e `Vendedor2`.

Classes

Embora o conceito de tipo definido pelo usuário seja vantajoso, ele deixa a desejar na criação de novos tipos de dados, no sentido pleno da palavra. Convém lembrar que um tipo de dados consiste em duas partes: um sistema predeterminado de armazenamento (como o sistema de complemento de dois, para o caso do tipo inteiro, e o sistema de vírgula flutuante, para o caso do tipo real), e um conjunto de operações predefinidas (como adição e subtração). Entretanto, os tipos definidos pelo usuário tradicionalmente permitem aos programadores definir apenas novos sistemas de armazenamento. Eles não fornecem meios para definir as operações a serem executadas com os dados nessas estruturas.

Uma classe, como introduzida na Seção 5.5, é uma maneira mais completa de estender os tipos disponíveis em uma linguagem de programação. Como acontece com o tipo definido pelo usuário, a classe é apenas um modelo, distinto das instâncias do tipo. Contudo, ela incorpora tanto um sistema de armazenamento de dados, via instâncias de variáveis, como uma coleção de procedimentos que definem as operações que podem ser realizadas nos dados do sistema.

As Figuras 7.29 e 7.30 mostram como uma classe conhecida como `PilhaDeInteiros` pode ser definida nas linguagens C++, C# e Java. Consideraremos os detalhes desses exemplos brevemente; por ora precisamos simplesmente notar que cada uma define a classe `PilhaDeInteiros` para conter duas variáveis de instância (um vetor de inteiros chamado `ElementosDaPilha` e um número inteiro usado para identificar o topo da pilha no vetor, chamado `PonteiroDaPilha` — veja o exercício 5 da Seção 7.4), dois procedimentos (chamados `Empilhar` e `Desempilhar`) e um construtor que estabelece a capacidade máxima da pilha a cada vez que uma instância da classe é criada.

Usando essa classe como modelo, um objeto chamado `PilhaUm`, que representa uma pilha com no máximo 50 números inteiros, pode ser criado em um programa Java ou C# com a instrução

```
PilhaDeInteiros PilhaUm = new PilhaDeInteiros(50);
```

e em um programa C++, com a instrução

```
PilhaDeInteiros PilhaUm(50);
```

Mais tarde, no programa, o valor 106 pode ser empilhado em `PilhaUm` usando-se a instrução

```

class PilhaDeInteiros
{
private:
    int *ElementosDaPilha;
    int PonteiroDaPilha;
    int PilhaMaxima;

public:
    PilhaDeInteiros(int Máxima)
    {ElementosDaPilha = new int[Máxima];
     PonteiroDaPilha = 0;
     PilhaMaxima = Máxima;
    }

    void empilha(int NovoElemento)
    {if (PonteiroDaPilha < PilhaMaxima)
        ElementosDaPilha[PonteiroDaPilha++] = NovoElemento;
    }

    int desempilha()
    {if (PonteiroDaPilha > 0) return ElementosDaPilha[--PonteiroDaPilha];
    };
};

```

Figura 7.29 Uma pilha de inteiros implementada em C++.

```

class PilhaDeInteiros
{
private int[] ElementosDaPilha;
private int PonteiroDaPilha;
private int PilhaMaxima;

public PilhaDeInteiros(int Máxima)
{ElementosDaPilha = new int[Máxima];
Ponteiro DaPilha = 0;
PilhaMaxima = Máxima;
}

public void empilha(int NovoElemento)
{if (PonteiroDaPilha < PilhaMaxima)
    ElementosDaPilha[PonteiroDaPilha++] = NovoElemento;
}

public int desempilha()
{if (PonteiroDaPilha > 0) return ElementosDaPilha[--PonteiroDaPilha];
else return 0;
}
}

```

Figura 7.30 Uma pilha de inteiros implementada em Java e C#.

```
PilhaUm.empilha(106);
```

ou o elemento do topo de `PilhaUm` pode ser recuperado e armazenado na variável `ValorAntigo` usando a instrução

```
ValorAntigo =
    PilhaUm.desempilha();
```

Retornando às Figuras 7.29 e 7.30, observe como o encapsulamento é usado para garantir a integridade das estruturas de dados dentro da classe. Em particular, `ElementosDaPilha` e `PonteiroDaPilha` são designados privados, enquanto os métodos `empilha` e `desempilha` são públicos. Assim, a estrutura inteira da pilha não pode ser referenciada diretamente. Todos os acessos à pilha têm de ser realizados via métodos públicos.

Para apreciar o significado dessa proteção, suponha que `ElementosDaPilha` e `PonteiroDaPilha` não sejam privados e que um programador deseje referenciar o terceiro elemento de uma pilha do tipo `PilhaDeInteiros`. O programador, que conhece a forma como a pilha foi fisicamente implementada, ficará tentado a violar a sua integridade, referenciando diretamente `ElementosDaPilha`, em vez de seguir o processo formal de desempilhar antes os dois primeiros elementos. A dificuldade reside no fato de que futuros encarregados da manutenção do programa podem efetuar nela modificações incompatíveis com a referência direta, que se encontra camouflada em alguma região do programa. Como ilustração, para aumentar o tamanho máximo da pilha, a estrutura interna do tipo `PilhaDeInteiros` poderia ser reimplementada como uma estrutura ligada, em lugar de um vetor, como era até este ponto. Isso seria totalmente incompatível com a referência direta mencionada, a qual pressupõe que a pilha esteja implementada como vetor.

Para encerrar, observe que a classe `PilhaDeInteiros`, como definida nas Figuras 7.29 e 7.30, representa a culminância do tema que vem permeando este capítulo inteiro — o de coletar as estruturas de dados e os procedimentos que as manipulam em uma única unidade de programa. De fato, o conceito

Conhecimento declarativo versus procedural

O conhecimento tende a tomar uma das duas formas – uma exibida por uma pessoa que sabe o próprio nome, a outra exibida por uma pessoa que sabe andar. A primeira é declarativa, responde às questões relacionadas com *quem*, *o quê* e *por quê*. Frequentemente associamos o conhecimento declarativo com o acúmulo de fatos. A última forma é procedural, responde às questões relacionadas a *como*. Muitas vezes, associamos o conhecimento procedural com a habilidade de realizar a ação apropriada. As duas formas de conhecimento são importantes. Particularmente, o acúmulo de conhecimento declarativo sozinho raramente implica o domínio de um assunto em estudo.

Dentro de um computador, pensamos no conhecimento declarativo sendo armazenado em estruturas de dados tradicionais, enquanto o conhecimento procedural é armazenado sob a forma de algoritmos codificados como programas. Nesse sentido, então, o movimento das estruturas de dados tradicionais em direção aos objetos representa um deslocamento de estruturas que contêm apenas conhecimento declarativo a outras mais inteligentes, que compreendem as duas formas de conhecimento.

A biblioteca de modelos padrão

As estruturas de dados discutidas neste capítulo se tornaram estruturas padrão de programação – na verdade, tão padronizadas que muitos ambientes de programação as tratam como primitivas. Um exemplo é encontrado no ambiente de programação do C++, que frequentemente é acrescido da Biblioteca de Modelos Padrão (Standard Template Library – STL). A STL é meramente uma coleção de classes predefinidas (parecidas com a classe `PilhaDeInteiros` da Figura 7.30) que descreve as estruturas de dados populares. Em consequência, ao incorporar a STL a um programa C++, o programador fica liberado da tarefa de descrever essas estruturas em detalhe. Ele precisa apenas declarar identificadores que sejam desses tipos, da mesma maneira como declaramos `PilhaUm` para ser do tipo `PilhaDeInteiros` na Seção 7.7.

de classe estende o tema, permitindo que os programadores empacotem as estruturas de dados e seus procedimentos relacionados na forma de tipos de dados personalizados a partir dos quais muitas instâncias podem ser produzidas. Na realidade, contudo, o conceito de classe é mais geral do que o de tipos de dados, uma vez que uma classe pode consistir apenas em procedimentos. Ou, no outro extremo, uma classe às vezes consiste apenas em estruturas de dados. Assim, o conceito de classe é uma poderosa ferramenta para o desenvolvimento de *software* com a qual modelos para unidades de programa de várias formas podem ser projetados e implementados. As classes e o paradigma da programação orientada a objeto se tornaram ferramentas importantes no arsenal atual do desenvolvimento de *software*.



QUESTÕES/EXERCÍCIOS

1. Qual é a diferença entre um tipo e uma instância do mesmo?
2. Em que aspectos os tipos definidos pelo usuário e as classes são similares? Em que aspectos são diferentes?
3. Por que alguns itens em uma classe são designados como privados?
4. Descreva duas estruturas subjacentes que podem ser utilizadas para implementar um objeto do tipo “fila-de-inteiros”.

7.8 Ponteiros em linguagem de máquina

Neste capítulo, introduzimos os ponteiros e mostramos como podem ser usados na construção de estruturas de dados. Nesta seção, consideraremos como os ponteiros são manipulados em linguagem de máquina.

Suponha que queiramos escrever um programa na linguagem de máquina descrita no Apêndice C para retirar um elemento da pilha descrita anteriormente na Figura 7.12 e colocá-lo em um registrador de propósito geral. Em outras palavras, queremos carregar um registrador com o conteúdo da célula de memória, a qual é o elemento no topo da pilha. Nossa linguagem de máquina fornece duas instruções para a carga de registradores — uma com o código de operação 2, a outra com o código 1. Lembre-se de que, no caso do código 2, o campo de operando contém o dado a ser carregado, e no caso do código 1, contém o endereço do dado.

Uma vez que não conhecemos o conteúdo, não podemos usar o código de operação 2 para realizar o nosso objetivo. Além disso, também não podemos usar o código 1, pois não sabemos o endereço. Afinal de contas, o endereço do topo da pilha irá variar ao longo da execução do programa. O que conhecemos é o endereço do ponteiro da pilha, isto é, sabemos a posição do endereço do dado que queremos carregar. O que precisamos então é de um terceiro código de operação para carga de registrador, no qual o operando contenha o endereço de um ponteiro para o dado a ser carregado.

O projetista da máquina do Apêndice C pode usar o código de operação D para realizar esse objetivo. A linguagem seria implementada de maneira que uma instrução na forma DRXY significasse carregar o registrador R com o conteúdo da célula de memória cujo endereço é encontrado no endereço XY (Figura 7.31). Assim, se o ponteiro da pilha estivesse na célula de memória com endereço F0, então a instrução D4F0 faria o dado no topo da pilha ser carregado no registrador 4.

Essa instrução, contudo, não completa a operação de desempilhar. Também devemos subtrair um do ponteiro da pilha, de modo que ele passe a apontar para o novo topo. Isso significa que, após a instrução de carga, nosso programa em linguagem de máquina teria de carregar o ponteiro da pilha em um registrador, subtrair um e guardar o resultado de volta na memória.

Usando um dos registradores como ponteiro da pilha, em vez de uma célula de memória, podemos reduzir essa movimentação para trás e para a frente do ponteiro da pilha entre registrador e memória.

Entretanto, isso significa que teríamos de reprojetar a instrução de carga de maneira que ela esperasse o ponteiro em um registrador, e não na memória principal. Assim, em vez da abordagem inicial, o projetista da máquina pode definir uma instrução com código de operação D na forma DROS, que significa carregar o registrador R com o conteúdo da célula de memória apontada pelo registrador S (Figura 7.32). Então, a operação desempilhar completa seria realizada com essa instrução e uma outra para subtrair um do valor armazenado no registrador S.

Note que uma instrução similar é necessária para implementar a operação de empilhamento. O projetista da máquina pode, portanto, estender mais a linguagem do Apêndice C, introduzindo o código de operação E de maneira que uma instrução na forma EROS signifique armazenar o conteúdo do registrador R na célula de memória apontada pelo registrador S. Mais uma vez, para completar a operação de empilhamento, essa instrução seria seguida por outra, que acrescentaria um ao valor do registrador S.

Com essas instruções, a linguagem de máquina descrita no Apêndice C teria três técnicas de endereçamento. Na primeira, o operando da instrução contém o dado envolvido, como demonstrado pelo código 2; na segunda, contém o endereço do dado envolvido, como demonstrado pelos códigos 1 e 3 e na terceira, contém a localização do endereço do dado envolvido, como demonstrado pelos códigos D e E. Essas técnicas são conhecidas como **endereçamento imediato**, **endereçamento direto** e **endereçamento indireto**, respectivamente. Todas são comuns nas linguagens de máquina atuais.

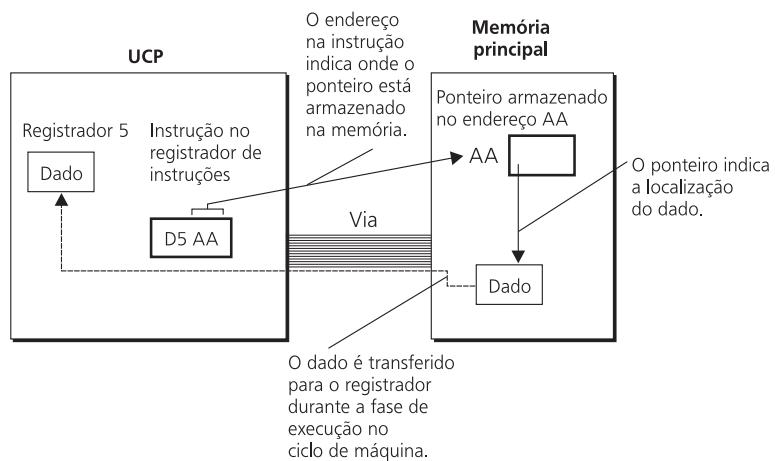


Figura 7.31 Nossa primeira tentativa de expandir a linguagem de máquina do Apêndice C para que ela tire vantagem dos ponteiros.

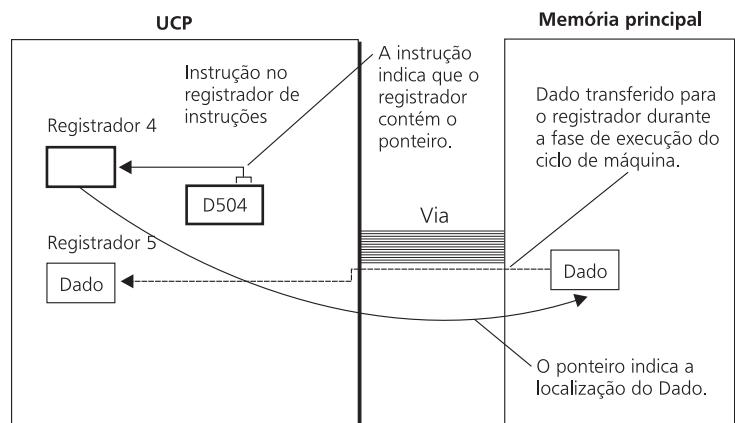


Figura 7.32 Carga de um registrador a partir de uma célula de memória localizada por meio de um ponteiro armazenado em outro registrador.



QUESTÕES/EXERCÍCIOS

1. Suponha que a linguagem de máquina do Apêndice C tenha sido expandida, como descrito no fim desta seção. Além disso, suponha que o registrador 8 contenha o padrão DB; a célula de memória no endereço DB, o padrão CA; e a célula de memória no endereço CA, o padrão A5. Que padrão de bits estará no registrador 5 imediatamente após a execução de cada uma das seguintes instruções?
 - a. 25A5
 - b. 15CA
 - c. D508
2. Usando as extensões descritas no fim desta seção, escreva uma rotina completa em linguagem de máquina para realizar uma operação de desempilhamento. Pressuponha que a pilha esteja implementada como mostrado na Figura 7.12, o ponteiro da pilha esteja no registrador F e o topo da mesma deva ser movido para o registrador 5.
3. Usando as extensões descritas no fim desta seção, escreva uma rotina completa em linguagem de máquina para copiar o conteúdo das cinco células de memória contíguas que começam no endereço A0 para as cinco que começam no endereço B0. Pressuponha que o programa inicie no endereço 00.
4. No texto, introduzimos uma instrução de máquina na forma DROS. Suponha que tenhamos expandido essa forma para DRXS, que significa “Carregar o registrador R com o dado apontado pelo valor no registrador S somado com o valor X”. Assim, o ponteiro para o dado é obtido tomando-se o valor do registrador S e então incrementando-o em X. O valor no registrador S não é alterado. (Se o registrador F contém 04, então a instrução DE2F carrega o registrador E com o conteúdo da célula de memória no endereço 06. O valor do registrador E continua 04.) Que vantagens esta instrução tem? Que tal uma instrução da forma DRTS — que significa “Carregar o registrador R com o dado apontado pelo valor do registrador S somado com o valor do registrador T”?

Problemas de revisão de capítulo

(Os problemas marcados com asterisco se referem às seções opcionais.)

1. Desenhe figuras que mostrem como a seguinte matriz aparece na memória de um computador, quando armazenada na sua seqüência de linhas e quando armazenada na de colunas.

A	B	C	D
E	F	G	H
I	J	K	L

2. Suponha uma matriz de 6 linhas por 8 colunas, armazenada em seqüência de linhas, a partir do endereço 20 (decimal). Se cada elemento da matriz ocupar apenas uma posição de memória, qual será o endereço correspondente ao elemento da sua terceira linha e quarta coluna? Qual será esse endereço se cada elemento ocupar duas posições de memória?
3. Resolva o Problema 2 supondo que o armazenamento seja feito na seqüência de colunas, em lugar da seqüência de linhas.
4. Que complicações ocorrem quando se tenta implementar uma lista dinâmica usando um arranjo homogêneo tradicional?
5. Descreva um método para armazenar um arranjo homogêneo tridimensional. Que fórmula de endereçamento seria usada para localizar o elemento do plano I, linha J e coluna K?
6. Suponha que uma lista de letras A, B, C, E, F e G seja armazenada em um bloco de posições contíguas de memória. Quais ações são exigidas para inserir a letra D na lista, mantendo a ordem alfabética?

- 7.** A seguinte tabela representa o conteúdo de algumas células da memória principal de um computador, ao lado do endereço de cada célula representada. Note que algumas das células contém letras do alfabeto e que cada posição é seguida por uma célula vazia. Coloque endereços nestas células vazias, de forma que cada célula que contenha uma letra, juntamente com a célula seguinte, forme um elemento em uma lista ligada, na qual as letras apareçam em ordem alfabética. (Use zero como ponteiro vazio.) Qual endereço contém o ponteiro para o início da lista?

Endereço	Conteúdo
11	C
12	
13	G
14	
15	E
16	
17	B
18	
19	U
20	
21	F
22	

- 8.** A tabela abaixo representa parte de uma lista ligada na memória principal de um computador. Cada elemento da lista é constituído de duas células: a primeira contém uma letra do alfabeto, e a segunda, um ponteiro para o próximo elemento da lista. Altere os ponteiros de forma que a letra N não pertença mais à lista. Em seguida, substitua-a pela letra G e altere os ponteiros de forma que a nova letra apareça na posição certa da lista, em ordem alfabética.

Endereço	Conteúdo
30	J
31	38
32	B
33	30
34	X
35	46
36	N
37	40
38	K
39	36
40	P
41	34

- 9.** A tabela seguinte representa uma lista ligada utilizando o mesmo formato dos problemas anteriores. Se o ponteiro para o início da lista contém o valor 44, qual o nome que a lista representa? Mude os ponteiros, de forma que a lista contenha o nome Jean.

Endereço	Conteúdo
40	N
41	46
42	I
43	40
44	J
45	50
46	E
47	00
48	M
49	42
50	A
51	40

- 10.** Qual das seguintes rotinas insere corretamente NovoElemento logo após o elemento chamado ElementoAnterior em uma lista ligada? O que há de errado com a outra rotina?

Rotina 1

1. Copiar o valor contido no campo de ponteiro de ElementoAnterior para o campo de ponteiro de NovoElemento.
2. Mudar o valor contido no campo de ponteiro de ElementoAnterior para o endereço de NovoElemento.

Rotina 2

1. Mudar o valor contido no campo de ponteiro de ElementoAnterior para o endereço de NovoElemento.
2. Copiar o valor contido no campo de ponteiro de ElementoAnterior para o campo de ponteiro de NovoElemento.

- 11.** Projete um procedimento para concatenar duas listas ligadas (isto é, colocar uma à frente da outra para formar uma lista única).

- 12.** Projete um procedimento para combinar duas listas contíguas ordenadas em uma única. E se elas forem listas ligadas?

- 13.** Projete um procedimento para inverter a ordem de uma lista ligada.

- 14.** Na Figura 7.11, apresentamos um algoritmo para imprimir uma lista ligada em ordem inversa

usando uma pilha como estrutura auxiliar de armazenamento. Projete um procedimento recursivo para executar esta mesma tarefa, sem fazer uso explícito de uma pilha. De que forma uma pilha ainda está relacionada com essa solução recursiva?

- 15.** Às vezes, uma única lista ligada é apresentada em duas seqüências distintas, bastando que cada elemento tenha dois ponteiros, em vez de um. Preencha a seguinte tabela de forma que, seguindo o primeiro ponteiro depois de cada letra, encontremos o nome Carol, porém, seguindo o segundo ponteiro após cada letra, encontremos as letras em ordem alfabética. Que valores contêm os ponteiros para o início de cada uma das duas listas representadas?

Endereço	Conteúdo
60	O
61	
62	
63	C
64	
65	
66	A
67	
68	
69	L
70	
71	
72	R
73	
74	

- 16.** A seguinte tabela representa uma pilha armazenada em um bloco de células contíguas de memória, na forma descrita no texto. Se a base da pilha estiver no endereço 10 e o ponteiro da pilha contiver o valor 12, qual valor será recuperado por uma instrução de desempilhamento? Qual valor se encontrará então no ponteiro da pilha?

Endereço	Conteúdo
10	F
11	C
12	A
13	B
14	E

- 17.** Faça uma tabela que mostre qual seria o conteúdo final das células de memória se a instrução do problema 16 fosse para empilhar a letra D, e não desempilhar. Qual seria o valor no ponteiro da pilha depois da instrução de empilhamento?
- 18.** Projete um procedimento para remover o elemento da base de uma pilha mantendo o restante inalterado.
- 19.** Projete um procedimento para comparar o conteúdo de duas pilhas.
- 20.** Suponha duas pilhas que contenham dados. Se for permitido mover um elemento de cada vez de uma pilha para outra, que arranjos dos dados originais seriam possíveis? Que arranjos seriam possíveis com três pilhas?
- 21.** Suponha duas pilhas que contenham dados, sendo permitido mover um elemento de cada vez de uma pilha para outra. Projete um algoritmo para trocar dois elementos adjacentes em uma das pilhas.
- 22.** Suponha que desejemos criar uma pilha de nomes, de comprimento variável. Por que é vantajoso armazenar cada nome em uma área separada da memória, para depois construir a pilha com ponteiros para tais nomes, em vez de construir a pilha com os próprios nomes?
- 23.** Uma fila se move lentamente na memória em direção ao seu início ou ao seu final?
- 24.** Suponha que você queira implementar uma “fila” na qual os elementos possuam prioridades associadas. Assim, um novo elemento deve ser colocado na frente dos outros com prioridade menor. Descreva um sistema de armazenamento para implementar essa “fila” e justifique as suas decisões.
- 25.** Suponha que cada elemento de uma fila ocupe uma célula de memória, sendo que o ponteiro para o seu início contenha o valor 11 e o ponteiro para o seu final, o valor 17. Quais os valores destes ponteiros após a inserção de um elemento e a remoção de dois?
- 26.** a. Imagine uma fila implementada de modo circular, conforme o diagrama abaixo. Faça um diagrama que apresente a estrutura após

a inserção das letras G e R, da remoção de três letras e da inserção das letras D e P.



- b. Quais erros poderão ocorrer na parte (a) se as letras G, R, D e P forem inseridas antes que alguma outra seja removida?

27. Descreva como um vetor pode ser utilizado para implementar uma fila em uma linguagem de alto nível.

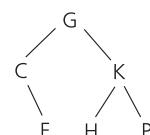
28. Suponha duas filas que contenham dados e que seja permitido mover apenas um elemento do início de uma fila para o fim da outra. Projete um algoritmo para trocar dois elementos adjacentes em uma das filas.

29. A tabela seguinte representa uma árvore armazenada na memória de um computador. Cada nó da árvore é constituído de três células. A primeira contém os dados (uma letra); a segunda, um ponteiro para o filho esquerdo do nó; e a terceira, um ponteiro para o filho direito do nó. O ponteiro vazio apresenta o valor 0. Sendo 55 o valor do ponteiro para a raiz da árvore, faça um desenho dessa árvore.

Endereço	Conteúdo
40	G
41	0
42	0
43	X
44	0
45	0
46	J
47	49
48	0
49	M
50	0
51	0
52	F
53	43
54	40
55	W
56	46
57	52

30. A tabela a seguir representa o conteúdo de um bloco de células da memória principal de um computador. Observe que algumas das células contêm letras do alfabeto, seguidas de duas células em branco. Preencha as posições em branco, de forma que esse bloco de memória represente a árvore esboçada abaixo da tabela. Utilize a primeira célula que aparece depois de uma letra como o ponteiro para o filho esquerdo do nó e a célula seguinte, como ponteiro para o filho direito. Use o valor 0 para ponteiros NIL. Que valor deve figurar no ponteiro para a raiz dessa árvore?

Endereço	Conteúdo
30	C
31	
32	
33	H
34	
35	
36	K
37	
38	
39	E
40	
41	
42	G
43	
44	
45	P
46	
47	



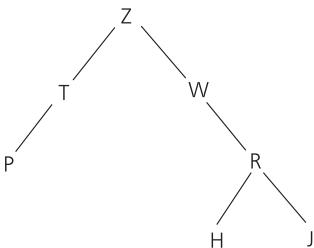
31. Projete um algoritmo não-recursivo para substituir o algoritmo recursivo representado na Figura 7.23.

32. Projete um algoritmo não-recursivo para substituir o algoritmo recursivo representado na Figura 7.26. Use uma pilha para controlar qualquer retrorastreamento que possa ser necessário.

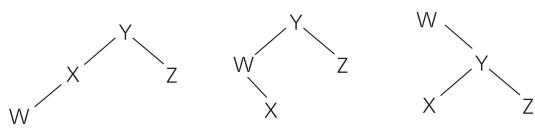
33. Aplique à árvore do Problema 29 o algoritmo recursivo de impressão de árvore da Figura 7.26.

Faça um diagrama que represente as ativações aninhadas do algoritmo (é a situação corrente de cada uma) no momento da impressão do nó X.

34. Mantendo inalterados o nó-raiz e a posição física dos dados, mude os ponteiros da árvore do Problema 29, de forma que o algoritmo de impressão de árvore da Figura 7.26 imprima em ordem alfabética os nós dessa árvore.
35. Faça um desenho que mostre como a árvore binária abaixo, armazenada sem ponteiros, está armazenada na memória com o uso de um bloco de células contíguas de memória, como descrito na Seção 7.6.



36. Suponha que as células contíguas que representam uma árvore binária, como descrito na Seção 7.6, contenham os valores A, B, C, D, E, F e G. Desenhe a árvore.
37. Dê um exemplo no qual se deseja implementar uma lista (a estrutura conceitual) como uma árvore (a estrutura subjacente). Dê um exemplo no qual se queira implementar uma árvore (a estrutura conceitual) como uma lista (a estrutura subjacente).
38. As estruturas de árvore ligada discutidas no texto contêm ponteiros que permitem percorrer a árvore no sentido dos pais para os filhos. Descreva um sistema de ponteiros que permita percorrer a árvore no sentido dos filhos para os pais. E se fosse pedido no sentido dos irmãos?
39. Descreva uma estrutura de dados adequada para representar a configuração de um tabuleiro durante um jogo de xadrez.
40. Identifique as árvores abaixo cujos nós seriam impressos em ordem alfabética pelo algoritmo da Figura 7.26.



41. Modifique o procedimento da Figura 7.26 para imprimir a "lista" em ordem invertida.
42. Descreva uma estrutura de árvore para armazenar a história genealógica de uma família. Quais operações são executadas na árvore? Se a árvore for implementada como uma estrutura ligada, que ponteiros estarão associados a cada nó? Projete procedimentos para executar as operações acima identificadas, supondo que a árvore seja implementada como uma estrutura ligada que utilize tais ponteiros. Usando o seu sistema de armazenamento, explique como se faz para encontrar todos os irmãos de uma pessoa.
43. Projete um procedimento para localizar e remover um determinado valor de uma árvore ordenada, na forma da Figura 7.22.
44. Na implementação tradicional de uma árvore, cada nó é construído com um ponteiro individual para cada filho possível. Se o nó possui menos filhos que ponteiros, alguns ponteiros são ajustados com o valor NIL. Entretanto, os nós nunca poderão ter mais filhos que ponteiros, e assim esse projeto limita o número máximo de filhos que um nó pode ter. Descreva como implementar uma árvore sem limitar o número de filhos que cada nó pode ter.
- *45. Qual a diferença entre um tipo definido pelo usuário e um tipo de dados primitivo?
- *46. Identifique as estruturas de dados e os procedimentos que você incluiria em uma classe para representar um caderno de endereços.
- *47. Identifique as estruturas de dados e os procedimentos que você incluiria em uma classe que representasse uma astronave em um jogo eletrônico.
- *48. Usando instruções das formas DROS e EROS, como descrito no fim da Seção 7.8, escreva uma rotina completa em linguagem de máquina para empilhar um elemento em uma pilha implementada, como mostrado na Figura 7.12. Pressuponha que o ponteiro da pilha esteja no registrador F e o elemento a ser empilhado, no registrador 5.

- *49. Suponha que cada elemento em uma lista ligada consista em uma célula de memória para o dado, seguida de um ponteiro para o próximo elemento. Além disso, suponha que um novo elemento, localizado no endereço A0, deva ser inserido entre os elementos localizados em B5 e C4. Usando a linguagem descrita no Apêndice C e os códigos de operação adicionais D e E como descritos no fim da Seção 7.8, escreva uma rotina em linguagem de máquina para realizar a inserção.
- *50. Que vantagens tem uma instrução da forma DR0S, como descrita na Seção 7.8, em relação a uma instrução da forma DRXY? Que vantagens tem a forma DRXS, como descrita na Questão/Exercício 4 da Seção 7.8, em relação à forma DR0S?

Questões sociais

As seguintes questões procuram auxiliar o leitor no entendimento de alguns assuntos éticos, sociais e legais no campo da computação. O objetivo não é meramente o de fornecer respostas a tais questões. O leitor também deve justificá-las e verificar se as justificativas apresentadas preservam sua consistência de uma questão para outra.

1. Suponha que um analista de *software* desenvolva uma organização de dados que proporcione uma manipulação extremamente eficiente dos dados para uma aplicação específica. Como os direitos dessa estrutura de dados podem ser protegidos? Ela é expressão de uma idéia (como um poema) e portanto protegida pela lei de direito autoral, ou as estruturas de dados se enquadram nas mesmas brechas legais dos algoritmos? E quanto à lei das patentes?
2. Até que ponto um dado incorreto é pior que a falta de dados?
3. Em muitos programas de aplicação, o tamanho máximo de uma pilha é determinado pela quantidade de memória disponível. Se acontecer de o espaço disponível ser consumido, o *software* será projetado para exibir uma mensagem como “estouro da pilha” e terminar a execução. Na maioria dos casos, esse erro nunca ocorre, e o usuário nem sabe de sua existência. Quem será responsável se ocorrer esse erro e alguma informação importante for perdida? Como o desenvolvedor do *software* poderia minimizar a sua obrigação?
4. Em uma estrutura de dados baseada no uso de ponteiros, a remoção de um dado normalmente consiste em alterar um ponteiro, em lugar de apagar células de memória. Assim, quando um elemento de uma lista ligada é removido, na verdade ele permanece na memória até que o espaço de memória que utilizava seja ocupado por outros dados. Quais pontos éticos e de segurança resultam dessa inércia na eliminação de dados que não são mais necessários?
5. É fácil transferir dados e programas de um computador para outro. Assim, é fácil transferir o conhecimento mantido por uma máquina a muitas máquinas. Entretanto, às vezes é muito demorada a transferência de conhecimento de uma pessoa para outra. Por exemplo, leva tempo para uma pessoa ensinar a outra uma nova língua. Que implicações esse contraste de velocidade de transferência de conhecimento poderia ter se a capacidade das máquinas começasse a desafiar a capacidade das pessoas?
6. O uso de ponteiros permite que dados relacionados sejam ligados na memória de um computador de uma maneira semelhante à forma como a informação é associada na mente humana. Como essas ligações na memória de um computador se assemelham com as ligações do cérebro? Como elas diferem? É ético tentar construir computadores que imitem cada vez mais a mente humana?
7. A popularização da tecnologia dos computadores produziu novos temas éticos ou simplesmente proporcionou um novo contexto no qual as teorias da ética são aplicáveis?

Leituras adicionais

- Carrano, F. M. and J. Prichard. *Data Abstraction and Problem Solving with Java: Walls and Mirrors*. Boston: Addison-Wesley, 2001.
- Jones, R. and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. New York: John Wiley & Sons, 1996.
- Kruse, R. L. and A. J. Ryba. *Data Structures and Program Design in C++*. Upper Saddle River, NJ: Prentice-Hall, 1999.
- Main, M. and W. Savitch. *Data Structures and Other Objects Using C++*. Boston: Addison-Wesley, 2001.
- Shaffer, C. A. *Practical Introduction to Data Structures and Algorithm Analysis*. 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 2001.
- Weiss, M. A. *Data Structures and Problem Solving Using Java*. 2nd ed. Boston: Addison-Wesley, 2002.

8

C A P Í T U L O

Estruturas de arquivo

No Capítulo 7, discutimos as formas de organização interna dos dados na memória principal da máquina. Neste capítulo, iremos nos concentrar nas técnicas de guardar dados no armazenamento em massa. Um tópico importante é que a maneira como a informação será acessada desempenha um papel importante na forma como ela deve ser armazenada. De forma semelhante ao caso das estruturas de dados, verificamos que o formato apresentado finalmente ao usuário pode não ser o mesmo do sistema de armazenamento. Assim como foi feito no Capítulo 7, aqui discutiremos e compararemos as organizações conceituais e as reais.

- 8.1 O papel do sistema operacional**
- 8.2 Arquivos seqüenciais**
 - Processamento em arquivo seqüencial
 - Arquivos texto
 - Aspectos de programação
- 8.3 Indexação**
 - Fundamentos de índices
 - Aspectos de programação
- 8.4 Hashing**
 - Um sistema específico de hashing
 - Problemas de distribuição
 - Aspectos de programação

8.1 O papel do sistema operacional

O software de aplicação geralmente é escrito em uma linguagem de alto nível que provê operações primitivas para acesso a arquivos por meio do sistema operacional. Essas primitivas permitem que o software de aplicação manipule um arquivo em unidades chamadas registros lógicos compatíveis com a aplicação. (Veja a subseção “Armazenamento e Recuperação de Arquivos” no fim da Seção 1.3.) Por exemplo, uma aplicação que lida com um arquivo de pessoal gostaria de manipular os dados em termos de registros lógicos, em que cada um consistisse em informação a respeito de uma única pessoa. Às vezes, a informação em um registro lógico é dividida em partes menores chamadas **campos**. Por exemplo, cada registro lógico em um arquivo de pessoal provavelmente consistiria em campos como nome, endereço, número de identificação etc.

Tabelas de alocação de arquivos

O texto indica que, para recuperar arquivos armazenados em disco magnético, um sistema operacional mantém uma lista dos setores atribuídos a cada arquivo. Na realidade, o espaço em disco geralmente é alocado em blocos com múltiplos setores, chamados aglomerados (*clusters*). Um aglomerado típico no PC consiste em 4 a 16 setores, e um disco rígido de alta capacidade possui milhares de aglomerados. Para ter um registro de quais aglomerados estão atribuídos a cada arquivo, um sistema operacional mantém uma tabela chamada tabela de alocação de arquivos (*file allocation table – FAT*) em cada disco. Essa tabela contém uma entrada para cada aglomerado do disco. Quando um arquivo é armazenado em um disco, o sistema operacional registra no diretório o número do primeiro aglomerado alocado ao arquivo. Então, no elemento da FAT que representa aquele aglomerado, o sistema operacional registra o número do próximo aglomerado alocado ao arquivo, e no elemento que representa este último aglomerado, o sistema registra o número do próximo. Assim, partindo do diretório e seguindo esse caminho através da FAT, o sistema operacional consegue recuperar o arquivo na ordem apropriada dos aglomerados.

As versões iniciais do sistema operacional Windows da Microsoft usavam FATs com entradas de 16 bits, o que significa que apenas 65.536 aglomerados diferentes podiam ser representados. Uma vez que cada um contém aproximadamente 2 KB, uma única FAT podia representar apenas 128 MB de espaço em disco – valor razoável quando os sistemas de disco rígido tinham capacidade de apenas 10 a 40 MB. Atualmente, são usadas entradas de 32 bits, o que significa que uma FAT pode ser usada para manter localizações em discos com capacidades medidas em terabyte – o equivalente a 2^{40} bytes.

Em contraste com essa composição lógica do arquivo, seu armazenamento impõe que ele seja dividido em blocos chamados registros físicos, que são compatíveis com o dispositivo de armazenamento em massa envolvido. Arquivos armazenados em disco, por exemplo, devem ser manipulados em unidades de tamanho de setor. A manipulação do arquivo em termos de registros físicos é feita pelo sistema operacional. Quando um programa de aplicação precisa ler uma parte do arquivo em unidades de registros lógicos, pede ao sistema operacional para realizar a leitura. Este responde lendo um conjunto de registros físicos necessários para atender à solicitação, colocando os dados obtidos em uma área de memória principal chamada retentor (*buffer*), e então disponibiliza esse retentor à aplicação (Figura 8.1). De modo similar, quando armazena informação, um programa de aplicação passa os dados para o sistema operacional. Este os guarda no retentor até que um registro físico completo tenha se acumulado, quando então ele é transferido para o armazenamento em massa.

Para cumprir essas obrigações de acesso a arquivos, o sistema operacional deve ter informações sobre o arquivo que está sendo manipulado. Por exemplo, ele deve saber em que dispositivo o arquivo está armazenado, o nome do arquivo, a localização do retentor em uso para transferir os dados para o software de aplicação e se o arquivo deve ser salvo após o término do programa. Essa informação é mantida em uma tabela chamada **descriptor do arquivo**, ou bloco de controle do arquivo, que é armazenada na memória principal. Um descriptor de arquivo é criado quando uma aplicação informa ao sistema operacional que está necessitando acesso ao arquivo e descartado quando a aplicação informa ao sistema operacional que o arquivo não é mais necessário. O processo de criar um descriptor de arquivo é conhecido como **abrir o arquivo** e o de descartar, como **fechar o arquivo**.

Antes que um programa de aplicação possa acessar um arquivo via sistema operacional, ele deve solicitar

ao sistema a abertura do arquivo. No paradigma imperativo, isso normalmente é feito por meio de uma instrução equivalente à seguinte, em pseudocódigo

```
Abir o arquivo
documento.txt
como ArqDoc para
leitura
```

que solicita a abertura do arquivo documento.txt. O resto da instrução indica que o arquivo já existe no armazenamento em massa (“para leitura” em vez de “para gravação”) e que será referenciado no programa pelo nome

ArqDoc. A razão para se usar um apelido (ArqDoc) no programa para se referir ao arquivo em vez do nome real é dupla. Primeiramente, o nome real usado pelo sistema operacional pode não ser compatível com as regras sintáticas ditadas pelo projeto da linguagem de alto nível. Por exemplo, o nome do arquivo documento.txt não seria compatível com uma linguagem na qual os identificadores não podem conter pontos. Em segundo lugar, o uso de um apelido permite que o programa seja convertido para um arquivo diferente simplesmente mudando-se o nome do arquivo na instrução de abertura, em vez de localizar e mudar cada ocorrência do nome ao longo do programa.

Em uma linguagem de programação orientada a objeto, os arquivos são tratados como objetos, e a abertura de um arquivo é feita no contexto de definir o objeto que fará o papel do arquivo. Assim, em um ambiente orientado a objeto, uma instrução para estabelecer um descritor de arquivo teria a forma

Criar o objeto ArqDoc como o arquivo de entrada documento.txt

que cria um objeto chamado ArqDoc, por meio do qual podemos acessar o documento.txt como arquivo de entrada. Mais tarde, no programa, os dados podem ser recuperados do arquivo pelo envio apropriado de mensagens ao objeto ArqDoc. Por exemplo, uma instrução como

Enviar a mensagem ObterCaractere a ArqDoc para recuperar Simbolo

pode ser usada para ler um único caractere do arquivo representado pelo objeto ArqDoc.

Quando um programa de aplicação termina de usar um arquivo, ele deve pedir ao sistema operacional para fechá-lo. Quando fecha um arquivo, o sistema operacional pode fazer mais do que simplesmente descartar o descritor. Por exemplo, se existe um registro físico parcialmente preenchido que contenha dados a serem armazenados no arquivo, o sistema deve transferir esse registro para o armazenamento em massa.

Para fechar um arquivo, um programa imperativo normalmente executa uma instrução equivalente a

Fechar o arquivo ArqDoc

O fechamento de um arquivo em um ambiente orientado a objeto é realizado pelo envio de uma mensagem ao objeto apropriado, instruindo-o a fechar o arquivo. Por exemplo,

Enviar a mensagem Fechar ao objeto ArqDoc

envia ao objeto chamado ArqDoc a mensagem de fechamento do arquivo.

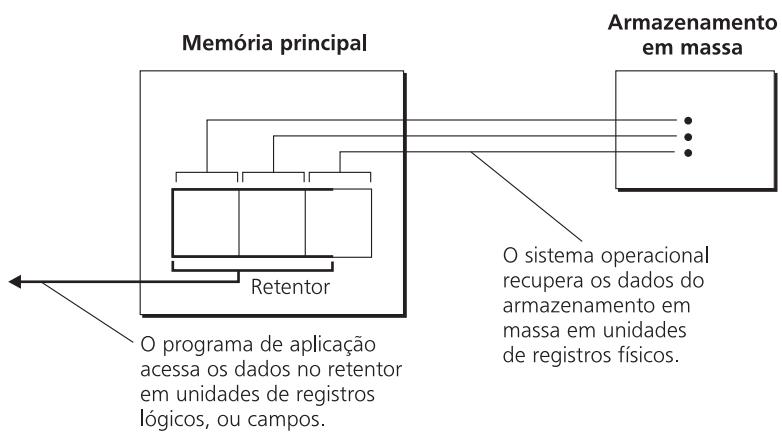


Figura 8.1 O papel de um sistema operacional ao acessar um arquivo.



QUESTÕES/EXERCÍCIOS

1. Resuma o sistema de retentor usado para transferir dados de um arquivo para um programa de aplicação.
2. Que é descritor de arquivo?
3. Que papel o gerente de arquivos de um sistema operacional desempenha na abertura de um arquivo?

8.2 Arquivos seqüenciais

Um **arquivo seqüencial** é acessado de uma maneira serial do início ao fim, como se a informação no arquivo estivesse disposta em uma longa lista. Exemplos incluem os arquivos de áudio e vídeo, os arquivos que contêm programas e os que contêm documentos de texto. De fato, a maioria dos arquivos criados por um usuário típico de PC é exemplo de arquivo seqüencial.

Processamento em arquivo seqüencial

Como contexto para apresentar as propriedades de um arquivo seqüencial, consideremos o exemplo clássico de um arquivo que contém registros de funcionários de uma pequena empresa. Tal arquivo consiste em registros lógicos (veja “Armazenamento e Recuperação de Arquivos” na Seção 1.3), em que cada um contém informações sobre um único funcionário. Cada registro é dividido em campos, como nome, endereço, número de identificação, número de seguro social etc.

Suponha que o nosso arquivo de funcionários seja utilizado para um processamento de folhas de pagamento, no qual, a cada período de pagamento, seja feito acesso a todo o arquivo. Assim que cada registro de funcionário é obtido, o pagamento correspondente é calculado, e o respectivo cheque, emitido. Dado que todos os registros serão processados, há pouca diferença em iniciar o processo por um deles ou por outro. Portanto, o método mais simples a adotar consiste em considerar que os registros estejam organizados na forma de uma lista e, então, recuperar e processar um de cada vez, do início ao final da lista. A atividade de processar esse arquivo seqüencial é exemplificada pela instrução

enquanto (o fim do arquivo não for atingido) **faça** (ler o próximo registro do arquivo e processá-lo)

Para suportar esse processamento seqüencial, os registros do arquivo devem ser armazenados em uma certa ordem, de modo a serem recuperados nesta mesma ordem. Se o sistema de armazenamento em massa em uso é uma fita ou um CD, isso é imediato. Uma vez que o próprio sistema de armazenamento é seqüencial, precisamos simplesmente gravar os registros no meio de armazenamento um após o outro. Então, processar o arquivo é meramente a tarefa de ler e processar os registros na ordem em que são encontrados. Este é exatamente o processo seguido quando se toca um CD de áudio, onde a

E-mail e MIME

A infra-estrutura padrão de *e-mail* na Internet é projetada para tratar arquivos ASCII tradicionais, sendo, portanto, incompatível com outros formatos de arquivo – incluindo imagens, sons e documentos produzidos por processadores de texto. Para transferir tais arquivos via correio eletrônico, é preciso codificá-los em um formato de arquivo texto, transferi-los via *e-mail* e então decodificá-los de volta ao seu formato original. Foi com esse propósito que o MIME (Multipurpose Internet Mail Extensions) foi desenvolvido. Resumindo, o MIME é uma coleção de padrões de codificação em que certos tipos de arquivo (por exemplo, imagens JPEG e GIF, vídeo MPEG e documentos do Word da Microsoft) podem ser convertidos para arquivo texto. Quando esse tipo de arquivo é submetido a um sistema de *e-mail* com MIME, ele é codificado e rotulado com o sistema de codificação utilizado, transferido a seu destino final e então decodificado de acordo com o rótulo do documento. Atualmente, os recursos do MIME são incorporados à maioria dos programas de aplicação de correio eletrônico.

música é armazenada como arquivo seqüencial nos setores ao longo de uma trilha contínua.

No caso de armazenamento em disco, contudo, o arquivo pode estar espalhado em diversos setores, que poderiam ser recuperados em diferentes ordens. Para preservar a ordem adequada, a maioria dos sistemas operacionais mantém uma lista dos setores nos quais o arquivo está armazenado (Figura 8.2). Essa lista é gravada como parte do sistema de diretório do disco. Por meio dela, o sistema operacional recupera os setores na seqüência correta e, portanto, o software de aplicação é escrito como se o arquivo estivesse armazenado seqüencialmente, ainda que na realidade esteja distribuído em várias partes do disco.

Inerente ao processamento de um arquivo seqüencial, está a necessidade de se detectar quando o fim do arquivo é atingido. Genericamente, nos referimos ao fim de um arquivo seqüencial como **EOF (end-of-file)**^{*}. Existem várias formas de se identificar o EOF. Uma delas consiste em armazenar um registro especial, denominado **sentinela**, como o último registro do arquivo. Naturalmente, para evitar confusão, os conteúdos deste registro devem ser valores que nunca aparecerão como dados na aplicação. Usando essa técnica, um programa em uma linguagem de terceira geração ditaria o processamento de um arquivo seqüencial com instruções como

```
ler o primeiro registro do arquivo;
enquanto (o registro lido não for a sentinela) faça
  (processar o registro e ler o próximo do
  arquivo)
```

Outra abordagem é deixar a tarefa de detectar o fim do arquivo ao sistema operacional. Por exemplo, se o arquivo está armazenado em disco e o sistema operacional está lendo os registros por meio de uma lista de setores, então ele sabe quando o fim foi atingido. Assim, pode reportar isso ao software de aplicação por meio de uma variável chamada EOF, à qual é atribuído o valor verdadeiro ou falso, dependendo de o fim do arquivo ter sido atingido. Usando essa estratégia, o software de aplicação poderia conter instruções como

```
while (not EOF) do
  (ler um registro do arquivo e processá-lo)
```

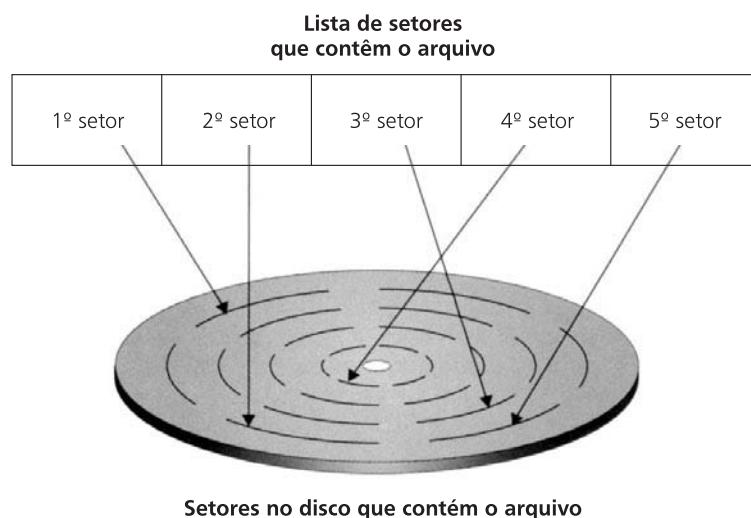


Figura 8.2 Manutenção da ordem de um arquivo por meio de uma tabela de alocação.

O consórcio World Wide Web

O Consórcio World Wide Web (W3C) foi formado em 1994 para promover a World Wide Web, desenvolvendo padrões de protocolo. O W3C foi a fonte do XML. Os projetos atualmente em andamento no W3C incluem o desenvolvimento de padrões para uma Web semântica, na qual a informação armazenada na Internet será (simplificadamente) ligada por meio de significado, em vez de palavras-chave. O W3C tem sede no CERN, o laboratório de física de partículas de alta energia, em Genebra, Suíça. Foi no CERN que a linguagem HTML original foi desenvolvida, bem como o protocolo HTTP para transferir documentos HTML pela Internet. Você pode aprender mais sobre o W3C em seu sítio na Web, cujo endereço é <http://www.w3c.org>.

*N. de T. Em inglês, *final de arquivo*.

procedimento Intercala (ArquivoEntradaA, ArquivoEntradaB, ArquivoSaída)

se (dois arquivos chegaram ao fim) **então** (Parar, com ArquivoSaída vazio)
 se (ArquivoEntradaA não estiver no fim) **então** (Declarar seu primeiro registro como registro corrente)
 se (ArquivoEntradaB não estiver no fim) **então** (Declarar seu primeiro registro como registro corrente)
enquanto (nenhum dos arquivos de entrada chegar ao fim) **faça**
 (Colocar o registro corrente com o “menor” valor do campo chave no ArquivoSaída;
 se (esse registro corrente for o último em seu arquivo)
então (Declarar que esse arquivo de entrada chegou ao fim)
senão (Declarar o próximo registro nesse arquivo de entrada como registro corrente)
)

Iniciando com o registro corrente do arquivo que não chegou ao fim,
 copiar os restantes para o ArquivoSaída.

Figura 8.3 Um procedimento para intercalar dois arquivos seqüenciais.

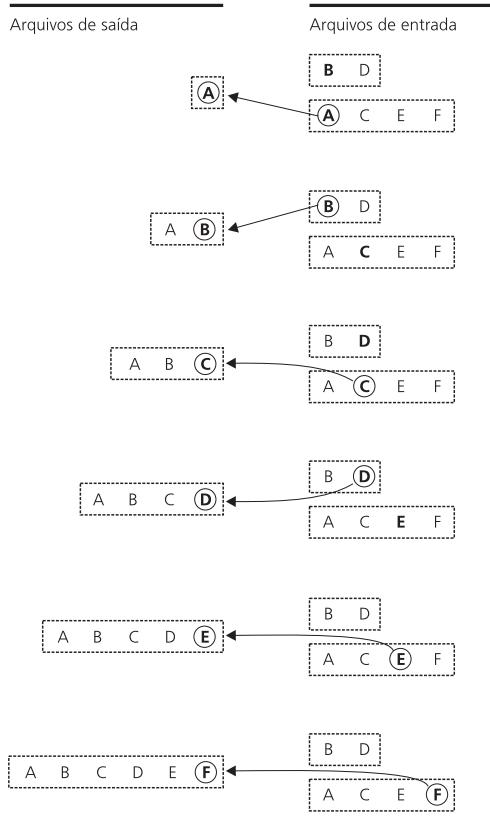


Figura 8.4 Aplicação do algoritmo de intercalação. (As letras são usadas para representar registros completos. Cada letra representa o valor do campo chave no registro.)

Os registros lógicos em um arquivo freqüentemente são identificados por um único campo. Por exemplo, em um arquivo de funcionários, o campo pode ser o que contém o número de Seguro Social ou talvez o de identificação do funcionário. Tal campo é chamado **campo chave**. Armazenando um arquivo seqüencial de acordo com o campo chave, pode-se reduzir drasticamente o tempo de processamento. Por exemplo, suponha que o processamento da folha de pagamentos necessite que cada registro de funcionário seja atualizado para refletir a informação de horas trabalhadas por ele. Se o arquivo que contém as horas trabalhadas está armazenado de acordo com o mesmo campo chave dos registros de funcionários, então esse processo de atualização pode ser feito acessando os arquivos seqüencialmente, usando as horas trabalhadas recuperadas de um arquivo para atualizar o registro correspondente no outro. Se os arquivos não estivessem arranjados por campo chave, o processo de atualização implicaria a leitura de cada registro de um arquivo e na busca repetida ao registro correspondente no outro.

Assim, a atualização de arquivos seqüenciais clássicos normalmente é feita em múltiplos passos. Primeiro, a nova informação (como a coleção de horas trabalhadas) é gravada em um arquivo seqüencial conhecido como arquivo de transações e este arquivo é ordenado para se compatibilizar com o arquivo a ser atualizado, que é chamado arquivo-mestre. Então, os registros do arquivo-mestre são atualizados lendo-se os registros dos dois arquivos seqüencialmente.

Outro exemplo clássico de processamento em arquivos seqüenciais é o de intercalar dois arquivos para formar um novo que contenha os registros de ambos. Os registros nos arquivos de entrada são dispostos em ordem crescente de acordo com um campo chave comum, e o arquivo de saída deve possuir essa mesma propriedade. O algoritmo clássico de intercalação está resumido na Figura 8.3. O tema subjacente é construir o arquivo de saída à medida que os arquivos de entrada são varridos seqüencialmente (Figura 8.4).

Arquivos texto

Quando um documento consiste em texto, é conveniente armazená-lo como um arquivo seqüencial no qual cada registro lógico contenha um único caractere codificado. Isso produz uma longa cadeia de *bits* que é facilmente dividida em segmentos compatíveis com os requisitos de registros físicos ditados pelo dispositivo de armazenamento em massa. Para recuperar o arquivo inteiro do sistema de armazenamento em massa, é necessário simplesmente ler os registros físicos na ordem adequada.

Esse arquivo é chamado **arquivo texto**. Os arquivos texto foram tradicionalmente codificados usando ASCII, resultando em formato de um caractere por byte. Atualmente, contudo, a popularidade do Unicode está levando a arquivos texto baseados em uma estrutura de um caractere em dois bytes. Assim, o termo genérico *arquivo texto* algumas vezes é substituído por terminologia mais precisa, como *arquivo ASCII* ou *arquivo Unicode*, para esclarecer o sistema de codificação subjacente.

É importante distinguir simples arquivos texto que são manipulados por programas utilitários chamados editores e os arquivos mais elaborados produzidos por processadores de texto. Ambos consistem em material textual. Contudo, um arquivo texto contém apenas uma codificação caractere a caractere do texto, enquanto um arquivo produzido por um processador de texto contém numerosos códigos internos que representam mudanças de fonte, informação de alinhamento etc. Além disso, os processadores de texto podem usar códigos internos, em vez de padrões, como ASCII ou Unicode, para representar o texto. Assim, a visualização e modificação de um arquivo produzido por um processador de texto exigem um programa mais elaborado do que um simples editor, necessário para processar um arquivo texto.

A simplicidade dos arquivos texto fez deles uma escolha popular em diversas situações. De fato, um arquivo texto freqüentemente é a estrutura subjacente usada na implementação de arquivos seqüenciais mais elaborados, como um arquivo de funcionários. É necessário simplesmente estabelecer um formato uniforme para representar a informação a respeito de cada funcionário como uma cadeia de texto, codificar a informação de acordo com esse formato e então gravar os registros de funcionários resultantes um após o outro, como uma única cadeia de texto. Por exemplo, pode-se construir um arquivo de funcionários simples convencionando-se que cada registro de funcionário terá 31 caracteres, consistirá em um campo de 25 caracteres que contenha o sobrenome do funcionário (preenchido com espaços em branco suficientes para completar o campo de 25 caracteres), seguido por um campo de 6 caracteres que represente o número de identificação do funcionário. O arquivo final seria uma longa cadeia de

Arquivos texto versus arquivos binários

Quando se imprimia um documento de texto usando os antigos dispositivos de teletipo, uma nova linha de texto exigia tanto a mudança de linha (movimento vertical) como um retorno de carro (movimento horizontal), cada qual codificado individualmente em ASCII. (A mudança de linha (*line feed*) é indicada pelo padrão 00001010, enquanto um retorno de carro (*carriage return*), por 00001101.) Para fins de eficiência, muitos programadores acharam conveniente indicar a quebra de linha em arquivos texto com apenas um desses códigos. Por exemplo, se todos concordam em marcar a quebra de linha com apenas um retorno de carro, em vez de retorno de carro e mudança de linha, então oito *bits* no espaço do arquivo são economizados para cada linha no texto. Tudo o que eles tinham de fazer era lembrar-se de inserir uma mudança de linha cada vez que um retorno de carro fosse encontrado quando imprimiam o arquivo.

Esses atalhos sobreviveram nos sistemas atuais. Em particular, o sistema operacional Unix pressupõe que as quebras de linha nos arquivos texto são indicadas com apenas uma mudança de linha, enquanto os sistemas desenvolvidos pela Apple Computer, Inc. usam apenas um retorno de carro, e os sistemas operacionais da Microsoft exigem retorno de carro e mudança de linha. O resultado é que quando esses arquivos são transferidos de um sistema para outro, conversões devem ser feitas. Essa é a origem da distinção entre “arquivos texto” e “arquivos binários” quando se transfere arquivos na Internet usando o protocolo File Transfer Protocol (FTP). Quando se usa o FTP, um “arquivo texto” é um arquivo que exige essa conversão, enquanto um “arquivo binário”, não. Em particular, arquivos produzidos por processadores de texto devem ser transferidos como “arquivos binários”, uma vez que, como vimos no texto, esses arquivos usam métodos próprios de codificação.

caracteres codificados na qual cada bloco de 31 caracteres representaria a informação a respeito de um único funcionário (Figura 8.5). A informação seria recuperada do arquivo em termos de registros lógicos que consistissem em blocos de 31 caracteres. Dentro de cada bloco, os campos individuais seriam identificados de acordo com o formato uniforme a partir do qual os blocos foram construídos.

A simplicidade dos arquivos texto tem levado até ao desenvolvimento de técnicas para codificar material não-textual em arquivos texto — um exemplo é uma partitura de música. À primeira vista, o padrão dos pentagramas, compassos e notas no qual a música tradicionalmente é representada não se conforma com o formato caractere a caractere ditado pelos arquivos texto. Contudo, podemos contornar esse problema desenvolvendo um sistema de notação alternativo. Mais precisamente, poderíamos convencionar representar o início do pentagrama por <pentagrama clave = "soprano">, o fim por </pentagrama>, a marcação de tempo com a forma <tempo> 2/4 </tempo>, o início e fim de um compasso por <comp> e </comp>, respectivamente, uma nota como dó colcheia por <notas> C col </notas> etc. Então, o texto

```
<pentagrama clave = "soprano"> <tom> C m </tom> <tempo> 2/4 </tempo>
<comp> <pausa> col </pausa> <notas> G col, G col, G col </notas> </comp>
<comp> <notas> E min </notas> </comp>
</pentagrama>
```

poderia ser usado para codificar a música mostrada na Figura 8.6. Usando essa notação, as partituras podem ser codificadas, modificadas, armazenadas e transferidas pela Internet como arquivos texto. Além disso, um *software* pode ser escrito para apresentar o conteúdo desses arquivos na forma de uma partitura tradicional ou mesmo para tocar a música em um sintetizador.

Note que o nosso sistema de codificação de partituras possui um certo estilo. Escolhemos limitar os termos chamados **marcas** (*tags*) que identificam componentes com os símbolos < e >. Decidimos indicar o início e fim de estruturas (como um pentagrama, um tom, uma sequência de notas, ou um compasso) por marcas de mesmo nome — a do fim precedida por uma barra (<comp> era finalizado com </comp>). E decidimos indicar os atributos especiais dentro de uma marca por expressões como clave = "soprano". Esse mesmo estilo poderia ser usado para desenvolver sistemas que representassem outros formatos, como expressões matemáticas, quadros médicos e páginas Web inteiras.

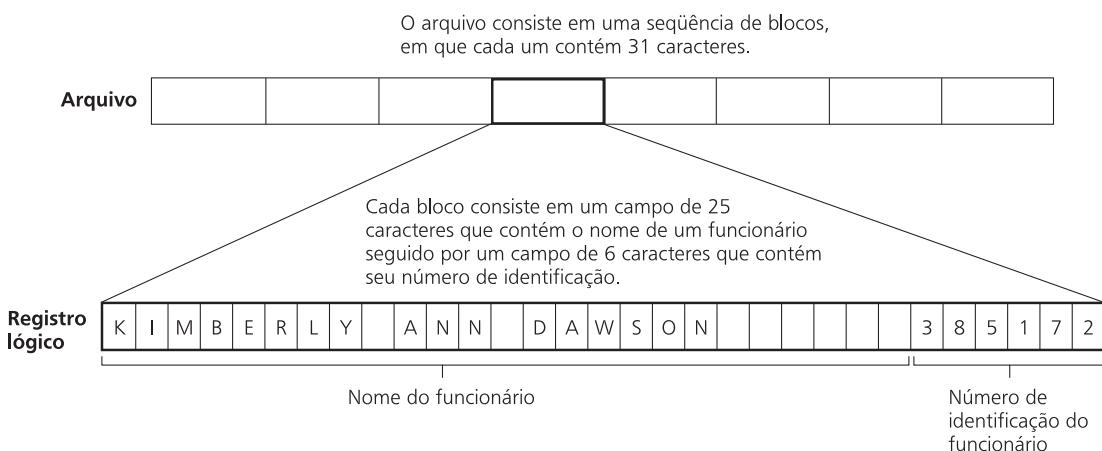


Figura 8.5 A estrutura de um simples arquivo de funcionários implementado como arquivo texto.

A eXtensible Markup Language (**XML**)^{*} é um estilo padronizado (similar ao nosso exemplo de música) para projetar sistemas de notação para representar dados como arquivos texto. (Na realidade, a XML é uma derivação simplificada de um antigo conjunto de padrões chamado Standard Generalized Markup Language, mais conhecido como SGML.) Seguindo o padrão XML, sistemas de notação chamados **linguagens de marcação** têm sido desenvolvidos para matemática (MathML), apresentações de multimídia (SMIL), música (4ML) e páginas Web (XHTML). (A XHTML é um refinamento da HTML que se conforma com o padrão XML. Por exemplo, a HTML pressupõe que o início de um novo parágrafo — indicado pela marca <p> — finaliza o parágrafo precedente, mas a XHTML insiste em que o parágrafo corrente seja explicitamente finalizado com a marca </p> antes do início do novo.)

A XML fornece um bom exemplo de como os padrões são projetados para ter uma larga faixa de aplicação. Em vez de projetar linguagens de marcação individuais e independentes para codificar vários tipos de documento, como partituras, texto e matemática, a abordagem implementada pela XML é desenvolver um padrão para as linguagens de marcação em geral. Então, usando esse padrão, as linguagens de marcação podem ser desenvolvidas para várias aplicações. As linguagens assim desenvolvidas possuem uma uniformidade que lhes permite ser combinadas para obter linguagens de marcação para aplicações complexas, como documentos de texto que contêm segmentos com partituras e expressões matemáticas.

Aspectos de programação

Fechamos esta seção, como faremos com as seções restantes desse capítulo, considerando como um programador que utilize uma linguagem de alto nível expressa os comandos para manipular o tipo de arquivo em consideração.

A maioria das linguagens de programação de alto nível fornece instruções para manipular arquivos seqüenciais cuja estrutura subjacente seja de arquivo texto. Se a linguagem for baseada no paradigma imperativo, essas instruções tomarão a forma de solicitação de execução de procedimentos pré-estabelecidos que realizam as operações desejadas. Por exemplo, elas podem ser equivalentes à instrução em pseudocódigo

Aplicar o procedimento LerArquivo para recuperar RegistroCorreio do arquivo ListaCorreio

que recupera o próximo registro lógico do arquivo ListaCorreio e o atribui à variável RegistroCorreio. Aqui, pressupomos que RegistroCorreio foi definido como um agregado heterogêneo cujos componentes representam os vários campos no registro, como Nome, Endereco, NumeroFuncionario etc. Em outros casos, os registros são transferidos campo a campo, de uma maneira similar à instrução em pseudocódigo

Aplicar o procedimento LerArquivo para recuperar Nome, Endereco,
NumeroFuncionario do arquivo ListaCorreio

Ainda em outros casos, os dados podem ser recuperados como caracteres individuais, como nas instruções na forma

Aplicar o procedimento ObterCaractere para recuperar Simbolo do arquivo ArqDocumento

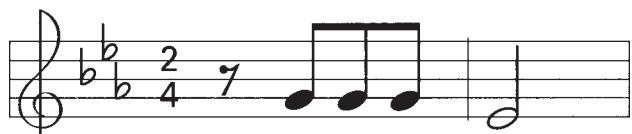


Figura 8.6 Os dois primeiros compassos da Quinta Sinfonia de Beethoven.

*N. de T. Linguagem de Marcação Extensível.

onde *Símbolo* é uma variável do tipo caractere, ou como linhas inteiras de texto, como na instrução

Aplicar o procedimento *LerLinha* para recuperar *LinhaTexto* do arquivo *ArqDocumento* que recupera uma linha inteira de texto delimitada pelo marcador de “fim-de-linha” do arquivo *ArqDocumento* e a atribui a um vetor de caracteres chamado *LinhaTexto*.

As linguagens de programação baseadas no paradigma orientado a objeto tratam os arquivos como objetos. Logo, as instruções para transferir de/para arquivos seqüenciais tomam a forma de mensagens aos objetos apropriados. Por exemplo, um programador escreveria instruções equivalentes a

Enviar a mensagem *LerArquivo* ao objeto *ListaCorreio* para recuperar *RegistroCorreio*

Em muitos casos, o processo de transferir dados de/para arquivos cuja estrutura subjacente é de arquivo texto envolve a conversão de dados em vez da simples transferência. Considere, por exemplo, a instrução

Aplicar o procedimento *Gravar* para colocar o valor de *Comprimento* no arquivo *ArqTexto* onde *Comprimento* é uma variável do tipo inteiro e *ArqTexto* é um arquivo texto baseado em ASCII. Então, o valor corrente de *Comprimento* deve ser convertido da notação de complemento de dois em caracteres codificados em ASCII, antes que ele seja gravado no arquivo. Mais precisamente, suponha que os números inteiros sejam representados na notação de complemento de dois e o valor corrente de *Comprimento* seja 134. Então, o padrão de bits associado a *Comprimento* será

0000000010000110

(representação de 134 em complemento de dois), mas o padrão de bits que deve ser gravado no arquivo é

001100010011001100110100

(o código ASCII para o símbolo 1, seguido do código ASCII para o símbolo 3, seguido do código ASCII para o símbolo 4) (Figura 8.7).

Considere agora o oposto, o processo necessário quando, mais tarde, recuperamos o valor de *Comprimento* gravado no arquivo, usando uma instrução como

Aplicar o procedimento *LerArquivo* para recuperar o valor de *Comprimento* gravado em *ArqTexto*

Isso exige que se recupere os símbolos de *ArqTexto* até que todos os dígitos que representam o valor tenham sido obtidos, e então se constrói a notação em complemento de dois correspondente ao valor antes que uma atribuição a *Comprimento* possa ser feita.

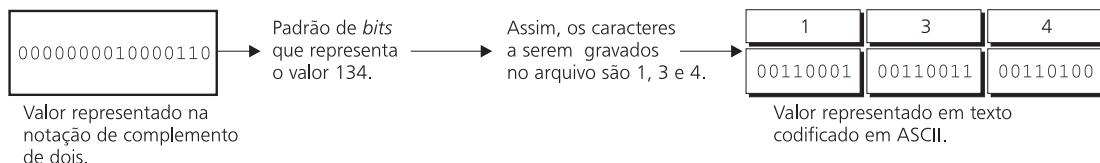


Figura 8.7 Conversão dos dados em complemento de dois para ASCII, de modo a serem armazenados em um arquivo texto.

Finalmente, devemos notar que a maioria das linguagens de programação de alto nível permite que um programador expresse as operações com os dispositivos periféricos como se eles fossem arquivos seqüenciais. Por exemplo, um teclado de computador normalmente é tratado como arquivo seqüencial de onde padrões de bits são recuperados em um formato caractere a caractere. De modo semelhante, o monitor e a impressora são tratados como arquivos seqüenciais, para onde padrões de bits são enviados. Assim, para recuperar dados digitados no teclado, um programador usaria uma instrução equivalente a

Aplicar o procedimento LerArquivo para recuperar Nome do arquivo Teclado

onde Nome é uma variável que se refere a um vetor de caracteres. Note, contudo, a conversão de dados que deve ser feita no caso de uma instrução como

Aplicar o procedimento LerArquivo para recuperar Idade do arquivo Teclado

onde Idade é uma variável do tipo inteiro. Se o valor 34 for digitado no teclado, este produzirá uma versão codificada do símbolo 3 seguida pela versão codificada do símbolo 4. Esse dado orientado a texto deve então ser convertido para a representação em complemento de dois de 34 antes que uma atribuição a Idade seja feita.

A Web semântica

Ao longo dos anos, a HTML se desenvolveu em um sistema de marcação de documentos de acordo com o modo como devem aparecer quando apresentados no monitor de um computador ou quando impressos. Por exemplo, os ingredientes em uma receita seriam marcados de maneira a aparecerem como uma lista, na qual cada um ficasse posicionado em uma linha separada. Contudo, espera-se que à medida que novas linguagens de marcação são desenvolvidas usando a XML, a ênfase seja posta na semântica. Suponha, por exemplo, que os ingredientes em uma receita sejam marcados como ingredientes (talvez com as marcas <ingrediente> e </ingrediente>) em vez de serem meros itens em uma lista. Então, um programa poderia ser escrito para identificar receitas que contenham ou não certos ingredientes. Isso seria um aperfeiçoamento substancial ao atual estado da arte, no qual apenas receitas que contêm ou não certas palavras podem ser isoladas. Mais precisamente, usando marcas semânticas, um programa poderia identificar todas as receitas de lasanha que não envolvam espinafre, enquanto um programa similar baseado somente no conteúdo de palavras saltaria uma receita que iniciasse com a declaração: “esta lasanha não contém espinafre”. Assim, uma melhora significativa na Web seria alcançada se os padrões na Internet fossem estabelecidos para marcar documentos de acordo com a semântica em vez da aparência. O resultado seria uma Web semântica mundial.



QUESTÕES/EXERCÍCIOS

1. Siga o algoritmo de intercalação mostrado na Figura 8.3, pressupondo que um arquivo de entrada contenha registros com valores do campo chave iguais a B ou E, enquanto o outro contém A, C, D e F.
2. O algoritmo de intercalação é o coração de um algoritmo popular de ordenação chamado ordenação por intercalação*. Você consegue descobrir esse algoritmo? (Dica: Qualquer arquivo não-vazio pode ser considerado uma coleção de arquivos de um elemento.)
3. Ser seqüencial é uma propriedade física ou conceitual de um arquivo?
4. Em que sentido um arquivo texto é um caso especial de arquivo seqüencial?

*N. de T. Em inglês, *merge sort*.

5. Que problemas podem ocorrer quando um documento de texto tem de retrorastrear uma longa distância em um documento que está sendo atualizado?
6. No último exemplo desta seção, suponha que as teclas 2 e 4 tenham sido pressionadas uma após a outra no teclado. Além disso, suponha que o teclado produza representações ASCII para os símbolos e que os valores inteiros sejam representados na notação de complemento de dois usando 16 bits por valor. Que padrões de bits seriam gerados pelo teclado? Que padrão de bits seria atribuído à variável Idade?
7. Às vezes, quando se edita um arquivo com um editor ou um processador de texto, a adição ou eliminação de um único caractere pode fazer com que o tamanho do arquivo mude em vários kilobytes. Por quê?

8.3 Indexação

Nas seções restantes deste capítulo, consideramos duas técnicas (indexação e *hashing*) que são usadas tanto no armazenamento em massa como na memória principal quando se armazena grandes coleções de dados. O objetivo de ambas é identificar rapidamente a localização em uma grande estrutura de armazenamento, onde os dados solicitados devem ser encontrados ou os dados a serem armazenados devem ser colocados. A primeira dessas técnicas consiste em usar um índice de uma maneira similar a quando se usa um índice em um livro para localizar tópicos de modo eficiente.

Fundamentos de índices

Um índice contém uma lista de valores que chamaremos **chaves** (uma vez que normalmente são valores do campo chave), e cada um destes identifica um bloco de informação que reside na estrutura de armazenamento relacionada. Cada chave do índice está associada a um elemento indicativo de onde o bloco de informação está armazenado. Assim, para encontrar um bloco de informação específico, primeiro encontra-se a chave identificadora no índice e então recupera-se o bloco de informação armazenado na localização associada com aquela chave.

Um índice tem sido há muito tempo uma ferramenta útil para conseguir acesso eficiente a arquivos guardados no armazenamento em massa, resultando em uma estrutura de arquivo conhecida como **arquivo indexado**. Um arquivo de índice em geral é armazenado separadamente no mesmo dispositivo de armazenamento em massa em que está guardado o arquivo indexado. Ele normalmente é trazido para a memória quando o arquivo é aberto, de modo que é fácil acessá-lo quando o acesso a registros no arquivo indexado é necessário (Figura 8.8).

Um exemplo clássico de arquivo indexado ocorre no problema de manter registros de funcionários. Aqui, um índice pode ser usado para evitar buscas demoradas quando se recuperam registros individualmente. Mais especificamente, se o arquivo de funcionários estiver indexado pelo número de identificação do funcionário, então um registro poderá ser

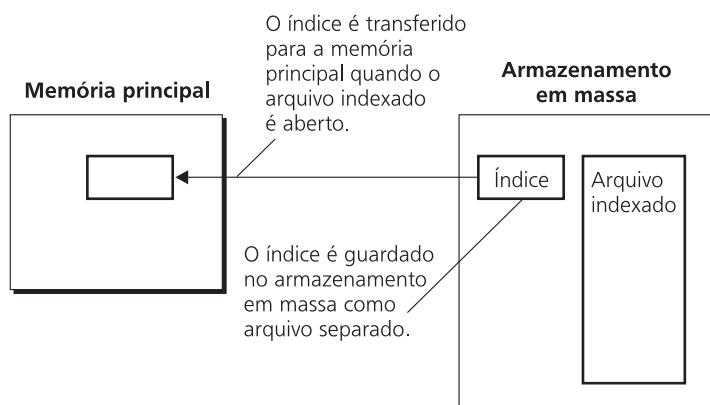


Figura 8.8 Abertura de um arquivo indexado.

recuperado rapidamente, desde que o número de identificação do funcionário seja conhecido. Outro exemplo é encontrado nos CDs de áudio, onde um índice é usado para permitir acesso relativamente rápido às gravações individuais.

Ao longo dos anos, numerosas variações no conceito básico de índice têm sido usadas. Uma delas é usar mais de um índice. Por exemplo, no caso dos registros de funcionários, pode ser necessário recuperar registros pelo número de identificação do funcionário ou pelo número do seguro social. Em tais casos, a solução é um sistema de múltiplos índices. Além de construir um índice com a lista dos números de identificação dos funcionários, construiremos um segundo índice baseado no campo chave do número do seguro social (Figura 8.9). Então, independentemente de iniciar pelo número do funcionário ou pelo do seguro social, o registro desejado pode ser rapidamente acessado consultando-se o índice apropriado. (Às vezes, este arquivo é chamado de **arquivo invertido**.)

Às vezes, é conveniente construir um índice que proporcione apenas uma localização aproximada, em vez da localização precisa da informação desejada. Isso é possível armazenando-se um arquivo seqüencial ordenado como segmentos com múltiplos registros. Cada segmento é, então, representado no índice por um único elemento, normalmente o último valor do campo chave do segmento. O resultado é um índice parcial, que contém somente alguns dos valores dos campos chave que aparecem no arquivo.

A estrutura de índice parcial está esboçada na Figura 8.10, na qual indicamos somente o elemento chave de cada registro e pressupomos que tais elementos são formados por um único símbolo alfabético. A recuperação de um registro deste arquivo consiste em localizar o primeiro elemento do índice que seja maior ou igual à chave e então buscar o segmento seqüencial correspondente ao registro desejado. Por exemplo, para encontrar o registro com chave E, no arquivo da Figura 8.10, seguiremos o ponteiro associado ao elemento de índice F e

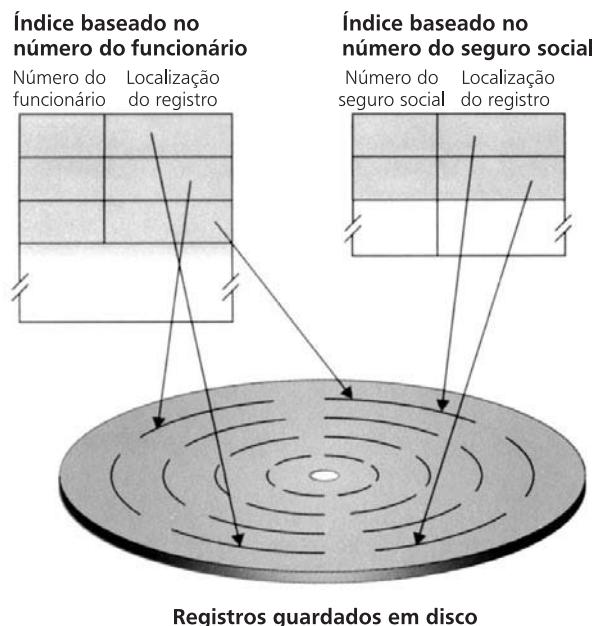


Figura 8.9 Um arquivo invertido.

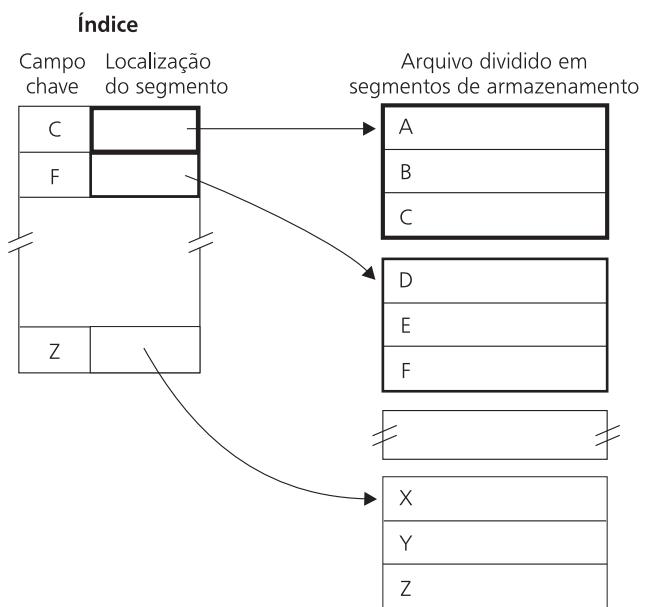


Figura 8.10 Um arquivo com índice parcial.

então localizaremos neste segmento o registro desejado.

A idéia de um índice parcial é aplicável a estruturas de dados que residem tanto na memória principal como no armazenamento em massa. Por exemplo, uma longa lista ligada poderia ser armazenada como uma lista menor de segmentos ligados. Então, um índice parcial seria usado para identificar o segmento que contém o elemento considerado na lista. Essa abordagem pode reduzir sensivelmente o tempo exigido para recuperar elementos da lista, uma vez que reduz o que seria uma pesquisa seqüencial aplicada à lista ligada completa a uma pesquisa aplicada a poucos segmentos.

Outra variação do conceito de índices é construir um índice de maneira hierárquica, na qual o índice inteiro apresenta uma estrutura em camadas ou em árvore. Um bom exemplo dessa estrutura de índices é o sistema de diretório hierárquico, usado pela maioria dos sistemas operacionais para a manutenção de arquivos. Nesse caso, os diretórios ou pastas fazem o papel de índices, e cada um contém ligações para os seus subíndices. Sob essa perspectiva, o sistema de arquivos inteiro é simplesmente um grande arquivo indexado.

Posicionamento de arquivos em disco

Suponha que queiramos armazenar um arquivo seqüencial, um arquivo texto, ou talvez uma imagem em um disco magnético. O ponto significativo é que os dados a serem armazenados ocupam mais de um setor, o que significa que cada vez que recuperarmos os dados, teremos de ler vários setores seqüencialmente. Podemos observar primeiramente que, se possível, devemos guardar o arquivo inteiro em uma mesma trilha (ou talvez no mesmo cilindro). De outra forma, a recuperação dos dados exigiria o reposicionamento do cabeçote de leitura/gravação durante o processo de recuperação. Outro ponto, contudo, não é tão óbvio – não devemos armazenar os dados em setores consecutivos ao longo da trilha, mas saltar alguns setores depois de cada setor usado.

A razão é que após a leitura de um setor, haverá um pequeno atraso antes que estejamos prontos para ler o próximo. Se tivermos armazenado a próxima parte do arquivo no próximo setor na trilha, esse setor já terá passado pelo cabeçote de leitura/gravação quando estivermos prontos para lê-lo. Assim, teríamos de esperar uma nova rotação do disco até que o setor venha outra vez.

Se, porém, houver setores entre o lido e o próximo desejado, teremos tempo para preparar a leitura antes que o setor desejado atinja o cabeçote de leitura/gravação. No caso de uma trilha com 16 setores, poderíamos gravar um arquivo seqüencial nos setores 5, 10, 15, 1, 6, 11, 16, 2, 7 etc. Dessa forma, usariam todos os setores da trilha e estariam aptos a recuperar os dados da trilha inteira em apenas cinco rotações do disco.

Aspectos de programação

Mais uma vez, encerramos considerando as características encontradas nas linguagens de programação de alto nível que permitem aos programadores manipular arquivos — nesse caso, arquivos indexados. Lembre-se de que um índice para um arquivo é meramente um arquivo secundário que contém uma lista dos valores de chave e as localizações dos registros correspondentes. Assim, para permitir que um programador implemente um arquivo indexado, a linguagem deve fornecer um meio de identificar e guardar a localização de cada registro quando ele é colocado no armazenamento em massa, bem como um meio de retornar a essa localização quando o registro é solicitado.

Um exemplo ideal dessa funcionalidade é encontrado no ambiente da linguagem de programação C. Ali, a função `fgetpos` pode ser usada para se obter a posição corrente em um arquivo. Por exemplo, a instrução

```
fgetpos (Pessoal, &Posicao);
```

atribui a posição corrente no arquivo `Pessoal` à variável `Posicao`. (A forma exata como essa posição é codificada varia de sistema para sistema.) A “posição corrente” é a posição na qual as próximas operações de leitura ou escrita no arquivo serão realizadas. Assim, ela pode ser usada em um índice para preservar a localização onde um registro é armazenado, isto é, se guardarmos a posição obtida com `fgetpos` antes de gravar cada registro, ficaremos com um índice que contém as localizações dos registros.

Enquanto `fgetpos` é usada para obter a posição corrente em um arquivo, a função `fsetpos` é usada para mover para uma nova posição. Mais especificamente, a instrução

```
fsetpos (Pessoal, &Posicao);
```

solicita que a posição corrente no arquivo `Pessoal` seja deslocada para a localização identificada pelo valor da variável `Posicao`. Se esta instrução for seguida por outra para ler do arquivo, obteremos os dados armazenados nessa localização. Assim, `fsetpos` fornece a ferramenta necessária para recuperar um registro específico por meio da informação obtida com `fgetpos`.

Resumindo, a função `fgetpos` permite-nos obter a localização na qual um item de dados foi originalmente armazenado em um arquivo, e se guardarmos essa localização em um índice, então a função `fsetpos` nos permitirá voltar àquela localização mais tarde para recuperar os dados.



QUESTÕES/EXERCÍCIOS

1. Que atividades um sistema operacional deve realizar quando solicitado a recuperar um registro em um arquivo indexado?
2. O que deverá ser acrescentado à sua resposta ao Exercício 1 se o sistema operacional também controlar um sistema de tempo compartilhado?
3. Qual é a diferença entre um índice de arquivo e um arquivo?
4. O que deve ser feito quando se fecha um arquivo indexado que não é necessário quando se fecha um arquivo seqüencial?

8.4 Hashing

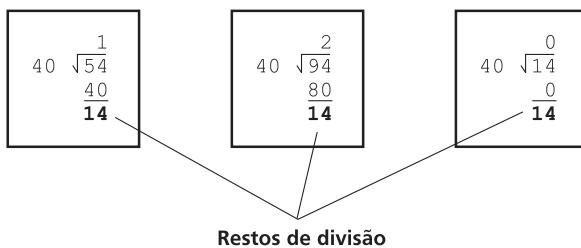
Embora a indexação proporcione acesso relativamente rápido a elementos em uma estrutura de armazenamento, isto é feito com o custo de se manter um índice. O **hashing**^{*} é uma técnica que proporciona acesso similar sem essa sobrecarga. Ela permite que um elemento seja localizado por meio do valor de sua chave, mas em vez de procurar a chave no índice, o *hashing* identifica a localização do elemento diretamente a partir da chave. O processo pode ser resumido como se segue: O espaço de armazenamento de dados é dividido em várias seções chamadas **baldes** (*buckets*). Os elementos de dados são dispersos nos baldes de acordo com um algoritmo (chamado **função hash**) que converte valores de chaves em números de balde. Cada elemento é armazenado no balde identificado por esse processo. Portanto, um item que tenha sido colocado na estrutura de armazenamento pode ser recuperado aplicando a função *hash* à sua chave de identificação para determinar o balde apropriado, recuperando o conteúdo desse balde e então procurando o desejado entre os elementos obtidos. Quando o *hashing* é aplicado a uma estrutura de armazenamento em massa, o resultado é chamado **arquivo hash**. Quando aplicado a uma estrutura de armazenamento na memória, geralmente é chamado **tabela hash**.

Um sistema específico de *hashing*

Apliquemos o conceito de *hashing* ao clássico arquivo de funcionários. Primeiro, vamos dividir a área de disco, destinada ao arquivo, em várias partes, os baldes. O número de baldes e o tamanho de cada um são decisões de projeto, a que iremos retornar mais adiante. Por ora, pressupomos que foram criados 40 baldes, que serão referenciados como balde número 0, balde número 1, até o número 39.

Vamos pressupor que o número de identificação de um funcionário será usado como chave para identificar o seu registro. Nossa próxima tarefa então é desenvolver uma função *hash* para converter essas chaves em números de balde. Ainda que os “números” de identificação dos funcionários possam

^{*}N. de T. Ao pé da letra, *to hash* significa picotar. Por falta de um termo técnico equivalente adequado em nossa língua, está mantido o termo em inglês.



Quando divididos por 40, os valores dos campos chave 14, 54 e 94 produzem um resto de 14, cada um. Assim, esses registros são armazenados no balde 14.

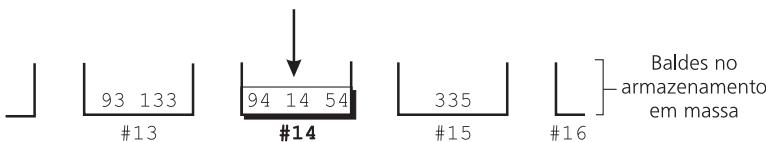


Figura 8.11 Os rudimentos de um sistema de *hashing*, no qual cada balde armazena os registros que lhe foram destinados pelo algoritmo de *hash*.

ter a forma 25X3Z ou J2X35 e, portanto, não sejam numéricos, eles são armazenados como padrões de *bits*, os quais podemos interpretar como números. Usando essa interpretação numérica, podemos dividir qualquer chave pelo número de baldes disponíveis e guardar o resto, que em nosso caso será um número inteiro na faixa entre 0 e 39. Assim, podemos usar o resto da divisão para identificar um dos 40 baldes (Figura 8.11).

Usando essa função, a construção do arquivo envolve a consideração de cada registro individualmente, a aplicação da nossa função *hash* divide-por-40, e então o armazenamento do registro nesse balde (Figura 8.12). Mais tarde, se precisarmos recuperar um registro, precisaremos simplesmente aplicar a nossa função *hash* à chave do registro para identificar o balde apropriado e então procurar nele o registro em questão.

A Arte do Hashing

Ao longo dos anos, muito se tem escrito sobre *hashing*. A meta é encontrar um algoritmo de *hash* eficiente e que distribua igualitariamente os elementos entre os baldes. Uma técnica, chamada *método meio-quadrado* (*mid-square*) é multiplicar o valor da chave por si próprio e selecionar os dígitos do meio de produto para representar o número do balde. Uma outra, chamada **método da extração**, é selecionar os dígitos que aparecem em certas posições da chave e construir o número do balde combinando esses dígitos selecionados por meio de algum processo predeterminado. Atualmente, essas variações do processo de divisão discutido no texto são bem populares. A principal preocupação é evitar o uso de chaves que contenham padrões, uma vez que padrões entre as chaves tendem a produzir padrões na seleção de baldes. Assim, é melhor basear um sistema *hashing* nos últimos dígitos de um número de telefone do que nos primeiros, já que o início de um número de telefone tende a representar uma área geográfica. De maneira similar, o uso de nomes como chaves é problemático, uma vez que os nomes tendem a se aglomerar em regiões (considere Smith, Mohammed e Deshpande).

Problemas de distribuição

O nosso sistema *hashing* tem alguns problemas; a essência de tais complicações está no fato de que, uma vez escolhida a função *hash*, não temos mais qualquer controle sobre a distribuição de registros nos baldes. Por exemplo, se todas as chaves produzirem o mesmo resto quando divididas por 40, elas serão colocadas no mesmo balde. O resultado é que a recuperação de um registro exigiria uma busca demorada em um único balde, o que proporcionaria pouca vantagem em relação à estrutura de arquivo seqüencial.

É, portanto, vantajoso selecionar uma função *hash* que distribua uniformemente os registros nos baldes. Podemos aperfeiçoar o nosso sistema proposto nesse aspecto mudando o número de baldes utilizados. Para justificar essa afirmação, recordemos que, se o dividendo e o divisor possuírem um fator comum, este fator estará presente no resto. Em particular, suponha que tenhamos construído o nosso sistema *hashing* usando 40 baldes e as chaves freqüentemente sejam múltiplos de 5. Como 40 também é um múltiplo de 5, o fator de 5 aparece como resto no nosso processo de divisão, e os elementos irão se aglomerar nos baldes associados aos restos 0, 5, 10, 15, 20, 25, 30 e 35. Situações semelhantes ocorrem nos casos de os valores das chaves serem múltiplos de 2, 4, 8, 10 e 20, pois todos são fatores de 40. Naturalmente, a

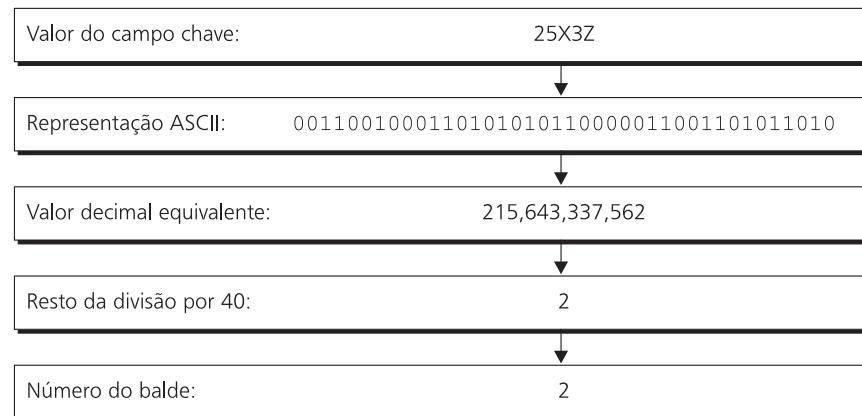


Figura 8.12
Hashing do valor de campo chave 25X3Z, sobre um dos 40 baldes.

observação deste fato sugere uma solução parcial: selecionar um número primo de baldes — minimizando assim a possibilidade de fatores comuns. Por exemplo, a chance de aglomerações no exemplo do arquivo de funcionários pode ser drasticamente reduzida usando-se 41 baldes, em vez de 40.

O fenômeno de duas chaves serem transformadas no mesmo valor é chamado **colisão**. As colisões são o primeiro passo para os aglomerados e, portanto, devem ser evitadas. Infelizmente, a teoria das probabilidades nos diz que as colisões são altamente prováveis, do mesmo quando o número de baldes é maior do que o de elementos armazenados. Para entender essa aparente contradição, vamos considerar o que acontece quando os elementos são inseridos em um sistema *hashing* de armazenamento com 41 baldes vazios.

Suponha que encontremos uma função *hash* que distribua arbitrariamente os registros entre os baldes, que nosso sistema de armazenamento esteja vazio e que iremos inserir os registros um por vez. Quando inserirmos o primeiro, este deverá ser guardado em um balde vazio. No entanto, quando inserirmos o próximo, só 40 dos 41 baldes estarão vazios, e assim a probabilidade de o segundo registro ser colocado em um balde vazio é de apenas $40/41$. Supondo que o segundo registro seja colocado em um balde vazio, o terceiro encontrará apenas 39 baldes vazios, e a probabilidade de ser colocado em um deles será de $39/41$. Prosseguindo neste processo, se os sete primeiros registros forem colocados em baldes vazios, o oitavo terá a probabilidade de $34/41$ de ser colocado em um dos baldes vazios restantes.

Esta análise nos permite calcular a probabilidade de os primeiros oito registros serem colocados em baldes vazios, pois é o produto das probabilidades de cada registro ser colocado em um balde vazio, supondo que os registros anteriores tenham sido colocados desta forma. Esta probabilidade é então:

$$\left(\frac{41}{41}\right) \left(\frac{40}{41}\right) \left(\frac{39}{41}\right) \left(\frac{38}{41}\right) \dots \left(\frac{34}{41}\right) = 0,482$$

O resultado obtido é menor que meio. Isto significa que é mais provável que pelo menos dois dos primeiros oito

Autenticação por meio do hashing

Embora tenhamos introduzido as técnicas de *hashing* no contexto da construção de estruturas eficientes de armazenamento, o *hashing* não se restringe a esse ambiente. Por exemplo, ele pode ser usado como um meio de autenticação de mensagens transmitidas pela Internet. O tema subjacente é transformar a mensagem (de uma maneira secreta) e criptografar o valor obtido. Esse valor é então transferido junto com a mensagem. Para autenticá-la, o receptor transforma a mensagem recebida (da mesma maneira secreta) e confirma que o valor produzido concorda com o original. Se eles não concordarem, pressupor-se-á que a mensagem é corrompida. Sob esse foco, a maioria das técnicas de detecção de erros pode ser considerada como aplicação de *hashing*. Por exemplo, o sistema de paridade simplesmente transforma cada padrão de bits em 0 ou 1 e envia esse resultado juntamente como o padrão de bits para ser usado no processo de autenticação.

registros sejam guardados no mesmo balde. Resumindo, uma colisão provavelmente ocorrerá no processo de inserção de apenas oito elementos em um sistema *hashing* com 41 baldes.

A alta probabilidade de colisões indica que o sistema *hashing* deve ser projetado para enfrentar o problema dos baldes encherem e possivelmente transbordarem. Uma abordagem seria permitir que os baldes se expandissem em tamanho. Por exemplo, quando um balde no armazenamento em massa começa a encher, uma área adicional de armazenamento poderia ser-lhe alocada. No caso de tabela *hash* na memória principal, os baldes são projetados como listas ligadas que podem crescer indefinidamente.

Nos casos em que o tamanho do balde é fixo, duas abordagens ao problema do transbordamento são comuns. Uma é permitir que um balde derrame em baldes adjacentes. Usando essa abordagem, se um novo elemento pertencer a um balde cheio, poderemos colocá-lo no primeiro balde disponível. Isso, é claro, complica o processo de recuperar dados quando descobre-se que o balde que deveria conter o elemento em questão está cheio, mas o elemento não está presente nele. O problema é ainda mais complicado quando começamos a eliminar elementos. Por exemplo, se eliminarmos elementos de um balde cheio que foi derramado nos seus vizinhos, deveremos procurar neles e trazer de volta os elementos para o balde originalmente atribuído?

Uma alternativa a deixar os baldes cheios entornarem nos vizinhos é deixá-los entornar em uma área excedente* que terá sido reservada para esse fim. A forma mais simples de se organizar a área excedente é permitir que ela seja uma mistura de elementos de todos os baldes. Nesse caso, a procura por um elemento que foi derramado de um balde cheio resulta em uma procura na área excedente inteira. A abordagem tradicional, contudo, é ligar os elementos que foram entornados de um único balde, de modo que a área excedente acabe contendo uma coleção de listas ligadas. Com tal sistema, um arquivo armazenado em cinco baldes pode ter a estrutura mostrada na Figura 8.13, na qual a área de armazenamento ocupada pelos registros é indicada por sombreamento. (Note que os baldes 1 e 4 entornaram na área excedente, enquanto o registro adicional no balde dois provocará nele a mesma ação.)

Um arquivo *hash* ou uma tabela *hash* relativamente vazios podem prover acesso eficiente aos dados, mas quando os elementos começam a derramar, o tempo de resposta pode degenerar significativamente. Por essa razão, as estruturas de armazenamento *hashing* são associadas ao conceito de **fator de carga**, que é a razão entre o número de elementos correntemente armazenados na estrutura e a capacidade total dos baldes. Enquanto essa razão estiver abaixo de 50%, o desempenho do sistema *hash*

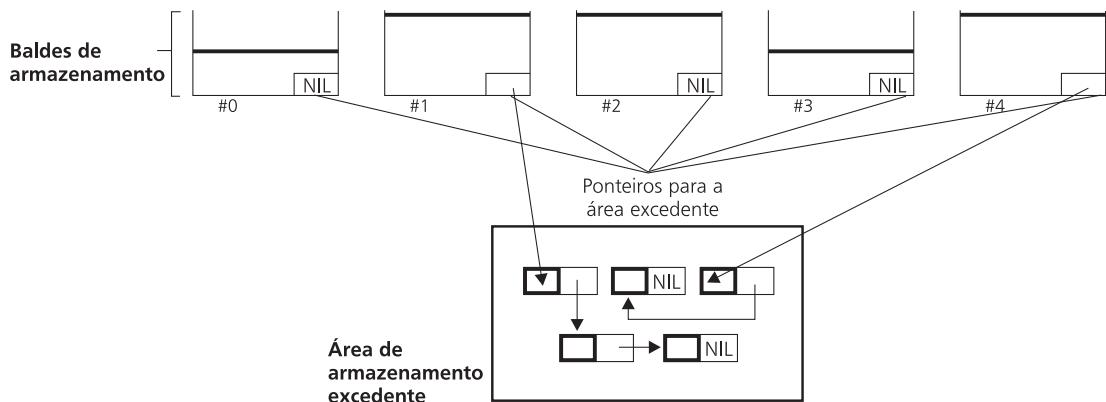


Figura 8.13 Manipulação da situação do balde cheio.

*N. de T. Em inglês, *overflow area*.

normalmente será bom. Contudo, se o fator de carga começar a subir para mais de 75%, o desempenho do sistema geralmente irá degenerar. De fato, um sistema *hash* costuma ser reconstruído com mais baldes quando o seu fator de carga atinge o valor de 75%.

Aspectos de programação

Mais uma vez, encerramos considerando as características encontradas nas linguagens de programação de alto nível para manipular as estruturas de armazenamento do tipo que foi discutido. No caso do *hashing*, encontramos várias opções em diferentes linguagens. Vamos começar considerando o ambiente de programação do Java.

A popularidade do *hashing* como meio de organizar grandes estruturas de armazenamento na memória principal é refletida pelo fato de o ambiente de programação Java fornecer uma classe (tipo) predefinida chamada *Hashtable*, a partir da qual tabelas *hash* podem ser construídas. Mais precisamente, a instrução

```
Tabela = new Hashtable(Capacidade, Fator);
```

cria um objeto do tipo *Hashtable* e o atribui à variável *Tabela*. A variável *Capacidade* é usada para indicar o número de baldes utilizados e a variável *Fator* estabelece um limite para o fator de carga. Cada balde na tabela é, de fato, uma lista ligada que consiste em elementos atribuídos àquele balde. Assim, uma tabela *hash* construída dessa maneira não sofrerá o problema de excedentes — os baldes simplesmente vão se tornando maiores. Por sua vez, o fator de carga é a razão entre o número de baldes não-vazios pelo número total de baldes. Quando o limite do fator de carga é ultrapassado, o Java automaticamente aumenta o número de baldes e reconstrói a tabela inteira.

Novos elementos podem ser armazenados na tabela por meio do método *put* (cujos argumentos identificam o valor da chave e os dados a serem guardados), e elementos podem ser recuperados por meio do método *get* (cujo argumento identifica o valor da chave desejada). Note que essa estrutura libera o programador Java dos aspectos de implementação e manutenção da tabela. De fato, a tabela *hash* pode ser usada como ferramenta abstrata.

Suponha, contudo, que queiramos criar um arquivo *hash* no armazenamento em massa, em vez de uma tabela *hash* na memória principal. Abordagens diferentes estão disponíveis, dependendo da linguagem de programação utilizada. Uma delas é criar numerosos arquivos seqüenciais, em que cada um faz o papel de um balde. Nesse caso, nosso arquivo *hash* consistiria em uma coleção de arquivos individuais, sob o ponto de vista do sistema operacional. Essa abordagem às vezes torna-se inadequada, uma vez que o sistema operacional limita o número de arquivos que a aplicação pode manter abertos. Contudo, contornamos esse problema projetando nosso programa para abrir e fechar os arquivos sempre que o acesso aos baldes for necessário.

Uma abordagem mais prática é reservar um espaço grande de armazenamento em massa criando um único arquivo vazio. Podemos então usar partes desse arquivo como baldes do nosso sistema. Várias linguagens de programação suportam essa abordagem. Por exemplo, a linguagem COBOL permite que um programador estabeleça um grande arquivo vazio que pode ser acessado de maneira similar a um vetor. Isto é, as localizações no arquivo podem ser identificadas por índices. Um programador poderia, portanto, criar esse arquivo e usar vários de seus segmentos como baldes em um arquivo *hash* (Figura 8.14). Assim, as localizações de 1 a 20 poderiam ser usadas como o primeiro balde, de 21 a 40, como o segundo etc. Dessa maneira, o conteúdo de um balde individual seria diretamente referenciado quando necessário.

Um sistema similar pode ser implementado em um programa C usando as funções *fgetpos* e *fsetpos*, introduzidas na seção anterior, para acessar diferentes partes do arquivo, quando solicitado.

Existe ainda outra abordagem à construção de um arquivo *hash*, que se beneficia das grandes memórias encontradas nos computadores atuais. Essa abordagem é ler o arquivo inteiro para a memória principal quando ele é aberto pela primeira vez e então acessá-lo como se fosse uma imensa tabela *hash*. Nesse caso, o arquivo de fato é armazenado como arquivo seqüencial, a partir do qual a tabela *hash* é inicializada quando o programa inicia a execução.

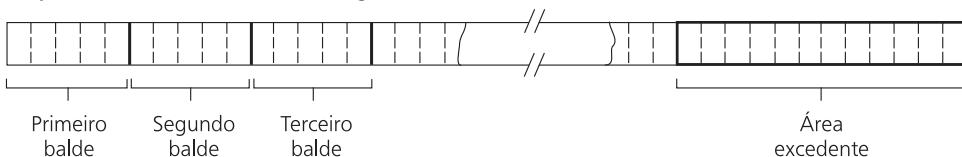
Arquivo inteiro visto como um longo vetor


Figura 8.14 Um grande arquivo particionado em baldes para ser acessado por *hashing*.

Essa é a abordagem da linguagem Java. O ambiente de programação Java provê uma variação da classe `Hashtable`, chamada `Properties`. Um objeto do tipo `Properties` é na realidade uma tabela *hash* inicializada a partir de um arquivo no armazenamento em massa por meio do método `load` e guardada no armazenamento em massa por meio do método `store`. Não se deve concluir, porém, que um objeto do tipo `Properties` seja gravado no armazenamento em massa como um arquivo *hash*. Em vez disso, a estrutura é na realidade a de um arquivo seqüencial de *bits*, a partir da qual a tabela *hash* apropriada pode ser construída na memória principal.


QUESTÕES/EXERCÍCIOS

1. Suponha que um sistema de registro de funcionários seja implementado como um arquivo *hash* com 1.000 baldes no armazenamento em massa, com o campo do número de seguro social como chave. Observe que uma possível função *hash* selecionaria os três primeiros dígitos do número do seguro social, pois sempre resultaria em um valor entre 0 e 999, inclusive. Por que esta não é uma boa escolha?
2. Explique como uma função *hash* mal escolhida pode resultar em um sistema de arquivos *hash* pouco melhor do que um arquivo seqüencial.
3. Suponha que um sistema de armazenamento *hash* seja construído usando a função de divisão conforme a descrição do texto, mas com seis baldes de armazenamento. Para cada um dos seguintes valores da chave, identifique o balde no qual está colocado o registro com tais valores. O que sai errado e por quê?

a. 24	b. 30	c. 3	d. 18	e. 15
f. 21	g. 9	h. 39	i. 27	j. 0
4. Quantas pessoas devem ser reunidas antes que haja chance de dois membros do grupo fazerem aniversário no mesmo dia do ano?
5. Na linguagem de programação Java, qual é a diferença entre as classes `Hashtable` e `Properties`?

Problemas de revisão de capítulo

(Os problemas marcados com asterisco se referem às seções opcionais.)

1. Suponha que um arquivo seqüencial contenha 50.000 registros e que sejam necessários 5 milissegundos para consultar um elemento.

Quanto tempo levará para se recuperar um registro no meio do arquivo?

2. Qual é a diferença entre um arquivo texto e um arquivo construído por um processador de texto?
3. Cite os passos a serem executados pelo algoritmo de intercalação da Figura 8.3 se um dos arquivos de entrada estiver inicialmente vazio.
4. Modifique o algoritmo da Figura 8.3 para manipular o caso em que os dois arquivos de entrada contêm um registro com o mesmo valor no campo chave. Pressuponha que esses registros sejam idênticos e que apenas um deva constar no arquivo de saída.
5. Projete um sistema no qual um arquivo armazenado em um disco possa ser processado como um arquivo seqüencial, com duas diferentes ordens de seqüência.
6. Usando o estilo XML informal apresentado no texto, projete uma linguagem de marcação para representar expressões algébricas simples como arquivos texto.
7. Usando o estilo XML informal apresentado no texto, projete um conjunto de marcas que um processador de texto usaria para marcar um texto. Por exemplo, como o processador de texto indicaria que o texto deve ser em negrito, itálico, sublinhado etc.?
8. Descreva como um arquivo seqüencial que contenha informações sobre os clientes de uma loja pode ser construído usando um arquivo texto como estrutura subjacente.
9. Projete uma técnica por meio da qual um arquivo seqüencial cujos registros lógicos não são consistentes quanto ao tamanho poderia ser implementado como arquivo texto. Por exemplo, suponha que você queira construir um arquivo seqüencial que contenha informações sobre romancistas no qual cada registro apresente informações sobre um romancista e uma lista dos trabalhos do autor.
10. Por que a escolha do número de identificação do funcionário para ser o campo chave é melhor do que a do sobrenome do mesmo?
11. Em que contexto se perdem as vantagens de um índice se, para mantê-lo pequeno, os segmentos utilizados em um esquema de índices parciais tiverem de ser extremamente grandes?
12. A seguinte tabela representa o conteúdo de um índice parcial.

Chave	Número do segmento
13C08	1
23G19	2
26X28	3
36Z05	4

Indique qual segmento deverá ser recuperado durante a busca do registro com cada chave:

- a. 24X17 b. 12N67
c. 32E75 d. 26X28

13. Baseado no índice do Problema 12, qual é o maior valor de chave do arquivo? E o menor?
14. Qual é a diferença entre um arquivo produzido por um editor e outro produzido por um processador de texto?
15. O que é a tabela de alocação de arquivos?
16. Que vantagens um arquivo indexado tem em relação a um arquivo *hash*? Que vantagens um arquivo *hash* tem em relação a um indexado?
17. Este capítulo apresentou um paralelo entre um índice tradicional de arquivo e um sistema de diretório de arquivos mantido por um sistema operacional. De que forma o sistema de diretórios de arquivos de um sistema operacional difere de um índice tradicional?
18. Qual estrutura de arquivos é mais recomendável para um arquivo que contenha a descrição do acervo de uma biblioteca, supondo que os livros possam ser referenciados pelo nome do autor, pelo título do livro e pelo assunto? Justifique a sua recomendação.
19. Se o único modo de extrair informação de um arquivo *hash* for pela transformação da chave de cada registro, qual informação será necessária para obter uma lista completa de todos os registros?
20. Se um arquivo *hash* for dividido em 10 baldes, qual será a probabilidade de pelo menos dois de três registros arbitrários cairem no mesmo balde? (Suponha que a função *hash* não reconheça prioridade entre baldes.) Quantos registros devem ser armazenados no arquivo até que a probabilidade de colisões ocorrerem seja maior do que a de não ocorrerem?
21. Resolva o problema anterior supondo que o arquivo esteja dividido em 100 baldes, em vez de 10.

- 22.** Se utilizarmos a técnica da divisão, discutida neste capítulo como sendo a função *hash*, e a área de armazenamento de arquivos for dividida em 23 baldes, em qual deles encontramos o registro cujo valor de campo chave, interpretado como um valor binário, é o número inteiro 124?
- 23.** Compare a implementação de um arquivo *hash* com a de uma matriz homogênea bidimensional. De que forma os papéis desempenhados pela função *hash* e pelo polinômio de endereçamento se assemelham?
- 24.** Projete um arquivo *hash* de palavras que possa ser utilizado para conferir ortografia. Qual função *hash* você escolheria? Esta sua escolha depende da linguagem a que pertencem as palavras? Por que tal arquivo não deve ser armazenado como um arquivo seqüencial?
- 25.** Calcule o tamanho do arquivo do problema anterior, supondo que contenha 50.000 palavras. Tal arquivo se ajustaria a uma memória principal de 8MB? Se a resposta for sim, qual seria a vantagem de mover todo o arquivo para a memória principal, para só então utilizá-lo?
- 26.** Projete um arquivo indexado (índice simples, parcial ou hierárquico) de palavras que possa ser utilizado em um corretor ortográfico. Por que tal arquivo não deve ser armazenado como arquivo seqüencial? O seu sistema indexado seria prático em termos de utilização do espaço de armazenamento? Ele seria prático em termos do tempo de recuperação?
- 27.** Se a função *hash* de divisão for utilizada conforme apresentado no texto, por que há maior probabilidade de aglomerações quando o espaço de armazenamento do arquivo for dividido em 60 baldes em vez de 61?
- 28.** Por que, em um arquivo *hash*, é vantajoso manter a lista dos registros excedentes dos baldes ordenada de acordo com os valores da chave?
- 29.** Se dividirmos a área de armazenamento de um arquivo *hash* em 41 baldes, de modo que cada um possa armazenar exatamente um registro, espera-se que haja, pelo menos, uma seção com transbordamento após o armazenamento de apenas oito registros. Por outro lado, se usarmos a mesma área de armazenamento como uma seção que possa armazenar 41 registros, sempre poderemos armazenar 41 registros antes da ocorrência de transbordamento. O que nos impede de implementar arquivos *hash* que utilizem esta última configuração?
- 30.** Suponha que o valor do campo chave de um registro seja XY. Usando a técnica de divisão para o *hashing* discutida no texto, converta este valor no número da seção que conteria o registro em um arquivo *hash* formado por 41 baldes. (Suponha que os caracteres sejam armazenados em código ASCII, um byte por caractere.)
- 31.** Suponha que seja criado um arquivo *hash* com informações sobre os habitantes de uma comunidade local americana. Se o campo chave deste arquivo for constituído de números de telefone de sete dígitos, por que não seria uma boa idéia aplicar o algoritmo de *hash* sobre os três primeiros dígitos do campo chave?
- 32.** Um arquivo *hash* que usa a função de divisão discutida no texto deve ser construído com 50, 51, 52 ou 53 baldes. Qual é a melhor escolha? Por quê?
- 33.** Suponha que um arquivo *hash* tenha sido construído usando a técnica de divisão discutida neste capítulo como função *hash*. Além disso, suponha que um dos registros no balde 3 possua um valor de campo chave que, quando interpretado como valor binário, se torne o número inteiro 26. Em quantos baldes a área de armazenamento em massa foi dividida para este arquivo?
- 34.** Projete uma tabela *hash* cujos elementos possam ser de diferentes tipos, ainda que todas as chaves consistam em cadeias de caracteres. Por exemplo, o elemento com chave “xyz” pode ser um registro de funcionário, mas o elemento com chave “abc” pode conter informações a respeito de um item no estoque.
- 35.** Quais das estruturas de arquivo discutidas nesse capítulo (seqüencial, indexada e *hashing*) é análoga à de um catálogo telefônico quando se procura o número do telefone de uma pessoa? Que aconteceria se você estivesse procurando a pessoa associada a um número específico?
- 36.** Dê uma vantagem de:
a. um arquivo seqüencial em relação a um arquivo indexado.

- b. um arquivo seqüencial em relação a um arquivo *hash*.
c. um arquivo indexado em relação a um seqüencial.
d. um arquivo indexado em relação a um arquivo *hash*.
e. um arquivo *hash* em relação a um seqüencial.
f. um arquivo *hash* em relação a um indexado.
- 37.** Em cada caso a seguir, indique qual estrutura de arquivo (seqüencial, texto, indexado ou *hash*) você recomendaria. Justifique a sua recomendação.
a. Um esboço superficial de um discurso.
b. Um arquivo de registros de clientes de um dentista.
c. Uma lista de correspondência.
d. Um arquivo de referência com 50.000 palavras e suas definições.
- 38.** Na Seção 8.3, vimos que a manutenção de dois índices permite que um arquivo de funcionários seja acessado tanto pelo número de identificação quanto pelo número do seguro social. Essa abordagem pode ser aplicada a um sistema *hashing*? Isto é, pode-se construir um sistema de tal forma que seus registros sejam recuperados rapidamente, transformando ora o número de identificação, ora o do seguro social?
- 39.** Suponha que um disco magnético seja dividido em setores de 512 bytes. Aproximadamente, quantos setores seriam necessários para armazenar um documento de texto com 20 páginas codificadas em ASCII? E se o documento fosse codificado em Unicode?
- 40.** De que modo um arquivo seqüencial se assemelha a uma lista ligada?
- 41.** Identifique duas técnicas que podem ser utilizadas para identificar o final de um arquivo texto.
- 42.** Explique como um arquivo seqüencial de registros de funcionários pode ser implementado usando primitivas de linguagens de programação para manipular arquivos texto.
- 43.** Defina os seguintes conceitos:
a. arquivo texto
b. arquivo indexado
c. arquivo *hash*
- 44.** Por que, para se referir a um arquivo, um programador precisa escrever um programa utilizando um identificador interno, em vez do próprio nome externo do arquivo?
- 45.** Identifique três elementos de informação que podem ser encontrados em um descritor de arquivo.
- 46.** Qual o objetivo de abrir um arquivo? Qual o objetivo de fechá-lo?
- 47.** Como a implementação de um arquivo seqüencial difere quando seu armazenamento é feito em fita e não em disco?
- 48.** Suponha que um arquivo seqüencial contenha 2000 registros. Se, durante um longo período, vários registros forem lidos do arquivo, qual será o número médio esperado de registros consultados para cada leitura? Justifique a sua resposta.
- 49.** Calcule a área necessária para armazenar um documento, escrito em folhas de papel com 40 páginas datilografadas em espaço duplo, em um arquivo texto.

Questões sociais

As seguintes questões procuram auxiliar o leitor no entendimento de alguns assuntos éticos, sociais e legais no campo da computação. O objetivo não é meramente o de fornecer respostas a tais questões. O leitor também deve justificá-las e verificar se as justificativas apresentadas preservam sua consistência de uma questão para outra.

- Até que ponto deve ser permitido às organizações a combinação de arquivos para obter informações que de outro modo não estariam disponíveis? Por exemplo, registros de impostos devem ser combinados com registros de saúde? Registros médicos devem ser combinados com registros de seguro?
- O cruzamento de informações em arquivos dá ao mantenedor a habilidade de responder a uma ação de terceiros de maneira que pareça não-relacionada, e que seria difícil de documentar.

Suponha que um cidadão, motivado por aspectos de privacidade, se recuse a participar de um censo nacional. O governo pode retaliar inspecionando minuciosamente os registros de impostos do indivíduo, ou atrasando o acesso a benefícios? O quanto você está vulnerável a tais retalições do governo, de instituições financeiras, de organizações sociais e de indivíduos? Como essas retalições podem ser distinguidas de meras coincidências?

3. Devido à sobrecarga no trabalho, um escriturário tem permissão para copiar alguns arquivos em disquete e levá-los para trabalhar em casa. Esse comportamento é aceitável? Sua resposta mudaria se o material levado para casa fosse registro impresso em vez de registro magnético? Sua resposta mudaria se o trabalhador fosse funcionário em uma universidade e os registros fossem de estudantes? Sua resposta mudaria se ele simplesmente acessasse os registros via linha telefônica de sua casa?
4. No passado recente, os registros históricos têm sido armazenados em papel ou filmes fotográficos. Atualmente, muitos têm sido armazenados em meio magnético e outros, como arquivos com a tecnologia de CD. O que é melhor? Você tem acesso a algum computador que possa ler dados de disquete de 8 polegadas, que foram populares nos anos 1970 e início dos anos 1980? E quanto aos disquetes de 5 ¼ que reinaram até os anos 1990? Você está habilitado a extrair informações gravadas nesses meios? Os pesquisadores daqui a 100 anos conseguirão extrair dados dos CDs atuais? O que conhecemos atualmente da civilização egípcia se eles tivessem feito seus registros em outros meios que não escultura e desenhos? A tecnologia aperfeiçoa ou degrada nossas atividades de manter registros?
5. Quando um arquivo é apagado de um disco, normalmente não é apagado de fato, mas apenas marcado como apagado. A informação contida em tal arquivo pode permanecer no disco durante algum tempo, até que tal seção de disco seja novamente utilizada para o armazenamento de algum outro arquivo. É ético reconstruir arquivos apagados, previamente usados por terceiros?
6. Que questões éticas seriam levantadas se um desenvolvedor de *software* projetasse um sistema que secretamente rotulasse cada arquivo criado com informações sobre seu criador? Por exemplo, um desenvolvedor de *software* para computadores pessoais poderia projetar seu *software* de tal forma que o nome do proprietário fosse secretamente associado a todos os arquivos criados por aquela máquina, e essa informação poderia viajar com o arquivo, se enviado pela Internet.
7. A maioria deve concordar que não seria ético para um técnico prestador de serviços ler os arquivos pessoais armazenados em um PC que estivesse sendo consertado. Essa ação deveria ser ilegal? Que efeito teria se ela se tornasse ilegal? Em geral, as leis devem ser usadas para garantir a ética de uma maioria (ou uma minoria)? Até que ponto as leis têm sido usadas dessa maneira?
8. A história contém numerosos casos nos quais a sociedade demorou a reconhecer as consequências negativas a longo prazo de suas ações. Exemplos incluem a chuva ácida, o buraco na camada de ozônio e os problemas de saúde associados ao ato de fumar. Poderia haver consequência similar relacionada com a revolução dos computadores?

Leituras adicionais

- Folk, M. J., B. Zoellick, and G. Riccardi. *File Structures — An Object-Oriented Approach in C++*, 3rd ed. Reading, MA: Addison-Wesley, 1998.
- Kay, D. C. and J. R. Levine. *Graphics File Formats*. 2nd ed. New York: Windcrest/McGraw-Hill, 1995.
- Means, W. S. and E. R. Harold. *XML in a Nutshell*. Cambridge, MA: O'Reilly, 2001.
- Norton, P. and J. P. Mueller. *Peter Norton's Complete Guide to Microsoft Windows XP*. Indianapolis, IN: Sams, 2001.

9

C A P Í T U L O

Estruturas de banco de dados

O tema banco de dados representa uma síntese das estruturas de dados e de arquivos. Um banco de dados moderno aplica as técnicas das duas áreas para obter um sistema único de armazenamento em massa que aparenta ter muitas organizações para atender a diversas aplicações. Neste capítulo, investigamos a composição dos sistemas de banco de dados e algumas linhas de pesquisa atuais.

- 9.1 Tópicos gerais**
- 9.2 A abordagem de implementação em níveis**
 - O sistema de gerência de banco de dados
 - Modelos de banco de dados
- 9.3 O modelo relacional**
 - Tópicos do projeto relacional
 - Operações relacionais
 - Tópicos de implementação
 - SQL
- * **9.4 Bancos de dados orientados a objeto**
- * **9.5 A preservação da integridade de bancos de dados**
 - O protocolo *commit/rollback*
 - Trancamento
- 9.6 Impacto social da tecnologia de banco de dados**

*Os asteriscos indicam sugestões de seções consideradas opcionais.

9.1 Tópicos gerais

O termo **banco de dados** refere-se a uma coleção de dados que é multidimensional, no sentido de conter ligações internas entre seus elementos, de modo que sua informação é acessível de várias perspectivas. Isso contrasta com um sistema tradicional de arquivos, às vezes chamados **arquivos achatados** (*flat files*), que é um sistema de armazenamento unidimensional no sentido em que apresenta a sua informação de um único ponto de vista. Enquanto um arquivo achatado com informações sobre compositores e suas composições pode fornecer apenas uma lista de composições ordenada por compositores, um banco de dados pode apresentar todas as obras de um único compositor, todos os compositores que escreveram um tipo particular de música, ou talvez os compositores que escreveram variações sobre a obra de outro compositor.

Historicamente, os bancos de dados evoluíram como forma de integrar sistemas de armazenamento de dados. À medida que as máquinas computacionais alcançaram uso generalizado na gerência da informação, cada aplicação tendia a ser implementada como sistema separado com sua própria coleção de dados. Por exemplo, a necessidade de processar uma folha de pagamentos deu origem a um arquivo seqüencial e, mais tarde, a necessidade de recuperação interativa pelo departamento de pessoal produziu

outro sistema de arquivos baseado em estrutura indexada. Embora cada um destes sistemas represente uma melhoria em relação às técnicas manuais correspondentes utilizadas anteriormente, de modo geral, o conjunto de sistemas independentes automatizados ainda constitui um uso limitado e ineficiente de recursos, quando comparado às possibilidades de um sistema integrado de banco de dados. Por exemplo, uma vez que cada departamento tinha seu próprio sistema de arquivos, boa parte da informação necessária à organização era duplicada no armazenamento. Como consequência, quando um funcionário mudava de endereço, era necessário percorrer os inúmeros departamentos da organização, preenchendo os respectivos formulários de alteração de endereços. Erros tipográficos, cartões extraviados e apatia de funcionários podiam resultar logo em dados errados e contraditórios dentro dos vários sistemas. Após a mudança, a correspondência da empresa para este funcionário poderia chegar no novo endereço, mas com o nome errado, enquanto os registros de folha de pagamento poderiam continuar emitindo o endereço antigo. Nesta situação, os sistemas de banco de dados emergiram como meios de consolidar a informação armazenada e mantida por uma organização (Figura 9.1). Com tal sistema, a folha de pagamento e a correspondência da empresa poderiam ser processados a partir de um único sistema integrado de dados.

Banco de dados distribuídos

Como visto no texto, os bancos de dados foram originalmente construídos como forma de consolidar ou centralizar a informação. A perspectiva mais moderna, contudo, é vê-los como um meio de integrar a informação que pode estar armazenada em diferentes máquinas em uma rede ou em uma *internet*. Por exemplo, uma corporação internacional pode guardar e manter registros de funcionários locais em cada instalação e ligar esses registros por uma rede para criar um banco de dados distribuídos – um único banco de dados grande e integrado que consiste em dados residentes em diferentes máquinas.

Um banco de dados distribuídos pode conter dados fragmentados e/ou duplicados. O primeiro caso é exemplificado na situação anterior do registro de funcionários, na qual diferentes fragmentos do banco de dados são armazenados em diversas localizações. No segundo caso, duplicatas de alguns componentes do banco de dados são guardadas em diferentes localizações. Essa cópia pode ocorrer para reduzir o tempo de recuperação da informação. Os dois casos introduzem problemas que não havia nos sistemas centralizados mais tradicionais – como o disfarce da natureza distribuída do banco de dados, de modo que ele funcione como um sistema coerente, ou a garantia de que as partes duplicadas continuem duplicadas quando ocorrerem atualizações. Assim, o estudo de banco de dados distribuídos é uma área de pesquisa corrente.

Outra vantagem de um sistema integrado de dados é o controle obtido por uma organização quando a diversidade de informações que possui é depositada em um único ambiente comum. Enquanto cada departamento tem completo controle sobre seus próprios dados, esses dados tendem a ser usados para os interesses do departamento, e não da organização. Entretanto, quando um banco de dados central é implementado em uma grande

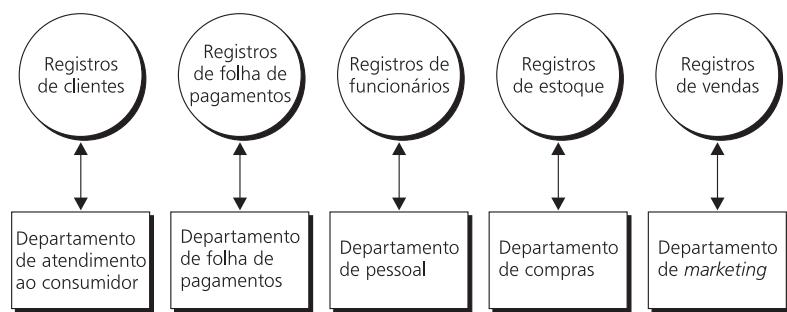
organização, o controle de informação normalmente se concentra no cargo administrativo conhecido como **administrador do banco de dados** (DBA)*, que pode ou não ser mantido por uma única entidade. Este administrador central (ou posição administrativa) conhece os dados disponíveis internamente na organização e as necessidades dos vários departamentos. Portanto, é nesta estrutura que as decisões relativas à organização de dados e acessos podem ser tomadas, considerando-se toda a organização da empresa.

Juntamente com os benefícios da integração de dados, surgem as desvantagens. Uma grande preocupação se refere ao controle de acesso a dados cruciais. Por exemplo, alguém que trabalha no jornal da companhia tem acesso ao nome e ao endereço dos funcionários, mas não deveria ter acesso aos dados da folha de pagamentos; do mesmo modo, um funcionário que processa a folha de pagamentos não deveria ter acesso aos outros registros financeiros da corporação. Assim, a capacidade de controlar o acesso à informação no banco de dados em geral é tão importante quanto a de compartilhá-la.

Para diferenciar privilégios de acesso, os sistemas de banco de dados normalmente utilizam esquemas e subesquemas. Um **esquema** é uma descrição da estrutura completa do banco de dados, que o software de banco de dados utiliza para mantê-lo. Um **subesquema** é uma descrição apenas da parte do banco de dados que é pertinente às necessidades de um determinado usuário. Por exemplo, consideremos um esquema, para um banco de dados universitário, que indique que cada registro de estudante contém dados como o endereço atual, o número do seu telefone e o seu registro acadêmico. Além disso, indique que cada registro de estudante é ligado ao registro do seu orientador da faculdade. Por sua vez, o registro de cada orientador contém o endereço deste, o seu histórico de emprego e assim por diante. Baseado neste esquema, um sistema de ponteiros é mantido para vincular a informação sobre um estudante ao histórico do emprego do seu orientador.

Para impedir que o responsável pelos registros da universidade utilize esta ligação para obter alguma informação sigilosa da faculdade, o seu acesso ao banco de dados deve ser restrito a um subes-

a. Sistema de informação orientado a arquivos



b. Sistemas de informações orientados a banco de dados

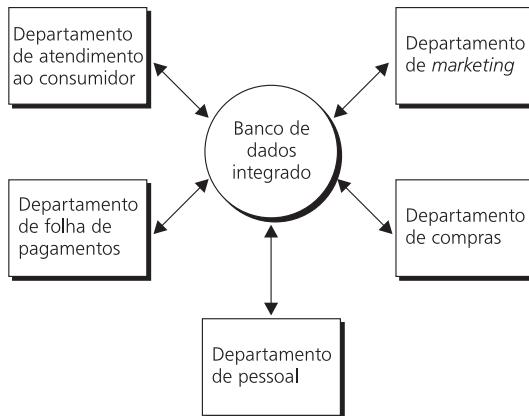


Figura 9.1 Comparação de um arquivo com uma organização de banco de dados.

*N. de T. Sigla do inglês, DataBase Administrator.

quema cuja descrição dos registros da faculdade não inclua o histórico do emprego. Neste subesquema, um usuário pode descobrir qual membro da faculdade é o orientador de um determinado estudante, mas não tem como acessar informação adicional sobre ele. Entretanto, o subesquema para o departamento de folha de pagamentos fornece o histórico do emprego de cada orientador, mas não inclui o vínculo entre estudantes e orientadores. Assim, o departamento de folha de pagamentos pode modificar o salário de um membro, mas não descobrir quais estudantes estão sob sua orientação.

Há outras desvantagens presentes na evolução da tecnologia de banco de dados, além destas, diretamente associadas com a segurança. O tamanho e o âmbito dos bancos de dados aumentaram rapidamente. Hoje, conjuntos extremamente grandes de dados podem ser agrupados e consultados com relativa facilidade, por grandes extensões de áreas geográficas e, com este aumento de dimensão, aumentam a desinformação e o mau uso da informação. São freqüentes os incidentes de injustiças decorrentes de relatórios de crédito imprecisos, registros incorretos de antecedentes criminais e discriminações resultantes de acessos não-autorizados ou não-éticos à informação pessoal.

Em outros casos, o problema subjacente trata, em primeiro lugar, do direito de coletar e armazenar informação. Que tipo de informação uma companhia de seguro tem direito de coletar a respeito dos seus clientes? Um governo tem direito de manter um registro dos votos de um cidadão? Uma companhia de cartão de crédito tem o direito de vender registros do perfil de consumo de seus clientes para empresas de propaganda? Estas perguntas representam alguns dos assuntos com que a sociedade terá de lidar como resultado da difusão da tecnologia de banco de dados.



QUESTÕES/EXERCÍCIOS

1. Identifique dois departamentos de uma indústria que façam usos diferenciados de dados cadastrais iguais ou similares.
2. Identifique diversos conjuntos de dados encontrados em um ambiente universitário que possam ser integrados em um único banco de dados.
3. Descreva como os subesquemas adotados para os dois departamentos da Questão 1 poderiam diferir.

9.2 A abordagem de implementação em níveis

Para entender os sistemas de banco de dados, é importante apreciar a distinção entre o *software* que se comunica com o usuário do banco de dados e o *software* que de fato o manipula.

O sistema de gerência do banco de dados

Um sistema de banco de dados comum consiste em duas camadas de *software* — uma camada de aplicação e uma de gerência do banco de dados (Figura 9.2). O *software* de aplicação trata da comunicação com o usuário (talvez uma pessoa, mas algumas vezes outro computador). Assim, é o *software* de aplicação que determina as características externas do sistema. Ele pode, por exemplo, se comunicar com o usuário por meio de diálogos de pergunta e resposta ou preenchimento de campos em branco em um formulário. Ele pode usar um formato de texto ou uma GUI.

Um *software* de aplicação não manipula diretamente o banco de dados. A manipulação efetiva do banco de dados é realizada por outro nível de *software*, chamado **sistema de gerência do banco de dados (DBMS)**^{*}. Uma vez determinada pelo *software* de aplicação a ação que o usuário está solicitando,

*N. de T. Sigla do inglês *Database Management System*.

ele usa o SGBD como ferramenta abstrata para obter os resultados esperados. Se a solicitação for para acrescentar ou eliminar dados, será o SGBD que de fato irá alterar o banco de dados. Se for para recuperar informação, será o SGBD que de fato realizará as buscas necessárias.

Essa dicotomia entre o *software* de aplicação e o SGBD possui vários benefícios. Um é que ela permite a construção e o uso de ferramentas abstratas, as quais, como já vimos várias vezes, constituem um conceito simplificador essencial no projeto de *software*. Se os detalhes como manutenção de índices, tratamento de excedentes, atualização de ponteiros, forem isolados dentro do SGBD, então o projeto do *software* de aplicação será muito simplificado. Além disso, considere o caso de um banco de **dados distribuídos** (um banco de dados espalhado em várias máquinas de uma rede). Sem os serviços do SGBD, o *software* de aplicação teria de conter rotinas para manter a localização corrente de várias partes do banco de dados. Entretanto, esses tópicos podem ser incorporados em um SGBD bem projetado de forma que o projeto do *software* de aplicação seja independente de o banco de dados estar distribuído em várias máquinas ou armazenado inteiramente em um único computador.

Uma segunda vantagem de separar o *software* de aplicação do SGBD é que tal organização permite o controle de acesso ao banco de dados. Uma vez definido que todo acesso deve ser feito pelo SGBD, tal sistema deverá fazer valer as restrições de acesso impostas pelos vários subesquemas. Por exemplo, o SGBD pode usar todo o esquema para suas necessidades internas, mas exigir que cada usuário se restrinja aos limites descritos pelo seu subesquema.

Uma razão adicional para separar, em dois pacotes de *software* diferentes, a interface do usuário e a manipulação efetiva de dados é conseguir **independência de dados** — a habilidade de mudar a própria organização do banco de dados sem modificar o *software* de aplicação. Por exemplo, o departamento de pessoal pode precisar acrescentar um campo adicional ao registro de cada funcionário para indicar se este escolheu participar do novo programa de seguro de saúde da companhia. Se o *software* de aplicação acessasse diretamente o banco de dados, tal modificação no formato de dados exigiria alterações em todos os programas que utilizassem o mesmo banco de dados. Como resultado, a alteração sugerida pelo departamento de pessoal causaria modificações à folha de pagamentos, bem como ao programa de impressão de etiquetas para a correspondência da companhia.

A distinção entre o *software* de aplicação e o SGBD elimina a necessidade de tal reprogramação. Para implementar uma alteração solicitada por um único usuário, modificam-se apenas o esquema, utilizado pelo sistema central, e os subesquemas dos usuários atingidos pela modificação. Todos os demais subesquemas permanecem inalterados, e os softwares de aplicação correspondentes, baseados em subesquemas inalterados, não precisam ser modificados.

Modelos de banco de dados

Em nossa discussão de estruturas de dados, vimos que as rotinas de *software* pré-escritas podiam ser usadas para traduzir solicitações (como as de empilhar e desempilhar) feitas em termos de uma estrutura conceitual (uma pilha) em ações apropriadas no sistema real de armazenamento. De maneira similar, um SGBD contém rotinas que traduzem os comandos feitos em termos de uma visão conceitual do banco de dados em ações apropriadas ao sistema de armazenamento de dados real. Essa visão conceitual do banco de dados é chamada **modelo do banco de dados**. Assim, usando essas rotinas do

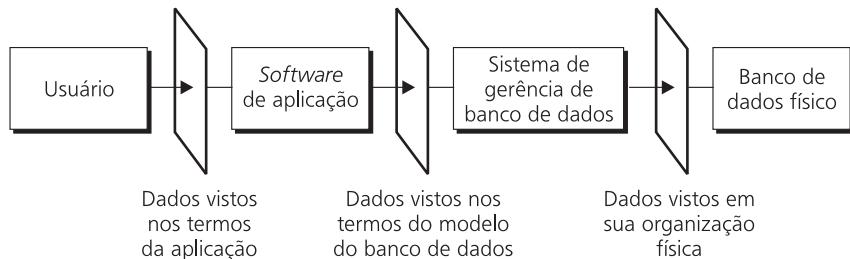


Figura 9.2 Os níveis conceituais de um banco de dados.

SGBD, o *software* de aplicação pode ser escrito como se a informação guardada no banco de dados estivesse armazenada de acordo com o modelo conceitual, em vez do sistema real de armazenamento.

Nas seções seguintes, vamos considerar o modelo de banco de dados relacional e o modelo de banco de dados orientado a objeto. No caso do modelo de banco de dados relacional, a visão conceitual do banco de dados é de uma coleção de tabelas que consistem em linhas e colunas. Por exemplo, as informações sobre funcionários de uma companhia podem ser vistas como uma tabela que contém uma linha para cada funcionário e colunas correspondentes ao nome, endereço, número de identificação etc. Por sua vez, o SGBD contém rotinas que permitem ao *software* de aplicação selecionar elementos de uma linha específica da tabela ou talvez determinar a faixa de valores encontrados na coluna de salário.

Essas rotinas formam as ferramentas abstratas usadas pelo *software* de aplicação para acessar o banco de dados. Mais precisamente, softwares de aplicação normalmente são escritos em linguagens de programação de propósito geral, como as discutidas no Capítulo 5. Tais linguagens apresentam os elementos básicos para expressar algoritmos, mas não fornecem operações para manipular bancos de dados. As rotinas existentes no SGBD têm o efeito de estender os recursos da linguagem utilizada (como veremos nas seções seguintes) de forma a dar suporte à imagem conceitual do modelo de banco de dados. O conceito de linguagem de propósito geral, vista como um fundamento que é estendido pelas rotinas do SGBD, faz a linguagem original ser conhecida como **linguagem hospedeira***.

Assim, a linguagem hospedeira estendida pelo SGBD proporciona o ambiente de programação usado no desenvolvimento de *software* de aplicação do banco de dados, e neste ambiente o banco de dados é manipulado como se estivesse organizado de acordo com o seu modelo conceitual.

A procura de melhores modelos de banco de dados é um processo contínuo. O objetivo é encontrar modelos que permitam que sistemas complexos de dados sejam facilmente conceituados, conduzam a meios concisos de expressar solicitações de informação e produzam sistemas eficientes de gerência de banco de dados.



QUESTÕES/EXERCÍCIOS

1. O uso de um arquivo indexado para processamento de folha de pagamentos e para recuperação interativa de dados fornece independência de dados?
2. Em um formato similar ao da Figura 9.2, desenhe um diagrama que represente as visões da linguagem de máquina, da linguagem de alto nível e do usuário final de um computador.
3. Resuma os papéis desempenhados pelo *software* de aplicação, pelo SGBD e pelas rotinas de manipulação propriamente dita dos dados, na recuperação de informação a partir de um banco de dados.

9.3 O modelo relacional

Nesta seção, veremos mais de perto o modelo relacional de banco de dados, que atualmente é o mais popular. Sua popularidade tem origem na simplicidade de sua estrutura. Ele retrata os dados armazenados em tabelas chamadas **relações**, que são similares ao formato no qual a informação é mostrada pelos programas de planilha. Como exemplo, o modelo relacional permite que a informação relativa aos funcionários de uma empresa seja representada por uma relação, tal como ilustra a Figura 9.3.

Uma linha de uma relação é chamada **ênupla**^{**}. Na relação da Figura 9.3, cada ênupla representa a informação acerca de um determinado funcionário. As colunas de uma relação são chamadas

*N. de T. Em inglês, *host language*.

^{**}N. de T. Em inglês, *tuple*.

atributos, pois cada elemento de uma coluna descreve alguma característica, ou atributo, da entidade representada pela ênupla correspondente.

Tópicos do projeto relacional

O projeto de um banco de dados, em termos do modelo relacional, é centrado no projeto das relações que compõem este banco de dados. Embora pareça uma tarefa simples, muitas sutilezas podem causar dificuldades a um projetista menos avisado.

Suponhamos que, além da informação contida na relação da Figura 9.3, desejemos juntar outra, sobre os cargos dos funcionários. Associado a cada um, podemos incluir o histórico do seu trabalho, composto de atributos como título do cargo (secretário, gerente de escritório, zelador), um código de identificação do cargo (único para cada cargo), um código de nível de qualificação associado a cada cargo, departamento a que este pertence, e período durante o qual o funcionário o ocupou, dado pelas datas de ingresso e desligamento. (Colocamos um asterisco na data de desligamento se o cargo representar a atual posição do funcionário.)

Uma abordagem para enfrentar este problema é estender a relação da Figura 9.3, incluindo tais atributos por meio de colunas adicionais na tabela, conforme ilustra a Figura 9.4. Entretanto, um exame mais detalhado do resultado desse procedimento revela diversos problemas. Um deles é a falta de eficiência devido à redundância. De fato, a relação não contém mais uma ênupla para cada funcionário, mas uma ênupla para cada associação deste com um cargo. Se um funcionário progrediu na companhia passando por uma seqüência de cargos, várias ênuplas estarão associadas a esse único funcionário. O problema é que a informação contida na relação original (nome de cada funcionário, seu endereço, número de identificação e do seguro social) deverá ser repetida. (Por exemplo, a informação pessoal acerca de Baker e Smith está repetida porque ambos exerceram mais de um cargo.) Além disso, se um dado cargo for exercido por muitos funcionários, o departamento correspondente a este cargo, bem como o respectivo código de qualificação, deverá ser identificado em cada ênupla que estiver associada a esse cargo. (Por exemplo, a descrição do cargo de gerente foi duplicada porque esse cargo foi exercido por mais de uma pessoa.)

Outro problema, talvez mais sério, relativo à nossa relação ampliada, aflora quando alguma informação deve ser removida do banco de dados. Por exemplo, suponhamos que Joe E. Baker seja o único funcionário exercer o cargo identificado como D7. Se ele for desligado da companhia e removido do banco de dados representado pela Figura 9.4, perderemos a informação sobre o cargo D7. De fato, a única ênupla

IdFuncionário	Nome	Endereço	NúmSeguroSocial
25X15	Joe. E. Baker	33 Nowhere St.	111223333
34Y70	Cherryl H. Clark	563 Downtown Ave.	999009999
23Y34	G. Jerry Smith	1555 Circle Dr.	111005555
.	.	.	.
:	:	:	:

Figura 9.3 Uma relação com informações acerca de um funcionário.

Sistemas de banco de dados para PC

Os computadores pessoais são usados em diversas aplicações, desde as mais elementares até as mais sofisticadas. Nas aplicações “elementares”, como o registro de aniversários e da liga de boliche, as planilhas freqüentemente são usadas em lugar dos softwares de banco de dados, uma vez que as aplicações demandam pouco mais que guardar, imprimir e ordenar dados. Contudo, existem sistemas mais sofisticados de banco de dados no mercado de PC, um dos quais é o Access da Microsoft. Ele é um sistema completo de banco de dados relacional com todos os recursos discutidos na Seção 9.3, bem como um software gerador de quadros e relatórios. A maior diferença sob o ponto de vista do usuário é que o Access usa uma interface gráfica que permite que as cláusulas de recuperação sejam postas de uma maneira gráfica em vez da sintaxe SQL apresentada no texto. Além disso, novas versões do Access fornecem recursos para a Internet, permitindo, por exemplo, que um elemento em uma ênupla seja uma URL, o que significa acesso direto a um sítio na Web por meio do banco de dados.

Id Funcionário	Nome	Endereço	Número SeguroSocial	IdCargo	NomeDo Cargo	Código Hab.	Departamento	Data Entrada	Data Saída
25X15	Joe E. Baker	33 Nowhere St.	111223333	F5	Gerente Júnior	FM3	Vendas	01-09-2001	30-09-2002
25X15	Joe E. Baker	33 Nowhere St.	111223333	D7	Chefe de Depart.	K2	Vendas	01-10-2002	*
34Y70	Cherryl H. Clark	563 Downtown Ave.	999009999	F5	Gerente Júnior	FM3	Vendas	01-10-2001	*
23Y34	G. Jerry Smith	1555 Circle Dr.	111005555	S25X	Secretária	T5	Pessoal	01-03-1999	30-04-2001
23Y34	G. Jerry Smith	1555 Circle Dr.	111005555	S26Z	Secretária	T6	Contabilidade	01-05-2001	*
•	•	•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•	•	•

Figura 9.4 Uma relação que contém redundância.

que contém a informação de que o cargo D7 exige um nível de qualificação K2 é a relativa a Joe Baker. Então, se apagarmos todas as referências a ele e depois retornarmos ao banco de dados para obter informação sobre o cargo, esta não será mais encontrada.

Poder-se-ia argumentar que a possibilidade de apagar somente em parte uma ênupla resolveria o problema. Isto, no entanto, introduziria novas dificuldades. (A informação sobre o cargo F5 também deveria ser mantida em uma ênupla parcial, ou essa informação existe em alguma outra posição da relação?) Além disso, a tentação de usar ênuplas parciais é uma forte indicação de que o projeto do banco de dados pode ser melhorado.

A fonte desses problemas é que combinamos mais de um conceito em uma única relação. Conforme proposto, a informação contida na relação ampliada da Figura 9.4 lida diretamente com dados sobre o funcionário (nome, número de identificação, en-

Relação FUNCIONÁRIO

IdFuncionário	Nome	Endereço	NúmSeguroSocial
25X15	Joe E. Baker	33 Nowhere St.	111223333
34Y70	Cherryl H. Clark	563 Downtown Ave.	999009999
23Y34	G. Jerry Smith	1555 Circle Dr.	111005555
•	•	•	•
•	•	•	•
•	•	•	•

Relação CARGO

IdCargo	NomeDoCargo	CódigoHabilidade	Departamento
S25X	Secretária	T5	Pessoal
S26Z	Secretária	T6	Contabilidade
F5	Gerente Júnior	FM3	Vendas
•	•	•	•
•	•	•	•
•	•	•	•

Relação ATRIBUIÇÃO

IdFuncionário	IdCargo	DataEntrada	DataSaída
23Y34	S25X	01-03-1999	30-04-2001
34Y70	F5	01-10-2001	*
23Y34	S26Z	01-05-2001	*
•	•	•	•
•	•	•	•

Figura 9.5 Um banco de dados de funcionários, constituído de três relações.

dereço, número do Seguro Social), os cargos disponíveis na companhia (identificação e título do cargo, departamento, código de nível de qualificação) e o relacionamento entre funcionários e cargos (data de entrada, data de saída). Feita essa observação, concluímos que nossos problemas poderão ser solucionados reprojetando o sistema para que passe a empregar três relações — uma para cada tópico. Utilizando o nosso novo sistema, poderemos manter inalterada a relação original (a qual iremos chamar, agora, de relação FUNCIONÁRIO) e inserir a informação adicional na forma de duas novas relações, CARGO e ATRIBUIÇÕES, o que produz o banco de dados da Figura 9.5.

Um banco de dados constituído destas três relações mantém informações sobre funcionários por meio da relação FUNCIONÁRIO, sobre cargos disponíveis pela relação CARGO, e sobre histórico de cargos, mediante a relação ATRIBUIÇÕES. Fica implicitamente disponível qualquer informação adicional pela combinação da informação extraída de diferentes relações. Por exemplo, se soubermos o número de identificação de um funcionário, encontraremos os departamentos em que ele trabalhou pesquisando, primeiramente, todos os cargos que o funcionário ocupou, mediante a relação ATRIBUIÇÕES e, em seguida, localizando os departamentos associados a tais cargos, por meio da relação CARGO (Figura 9.6). Por meio de processos como estes, qualquer informação que puder ser extraída a partir da única grande relação poderá ser igualmente obtida das três relações menores, evitando os problemas previamente citados.

Infelizmente, dividir a informação em várias relações não é sempre tão inequívoco como se mostrou no exemplo anterior. Para ilustrar, compare a relação original na Figura 9.7, que tem os atributos IdFuncionario, NomeDoCargo e Departamento com a decomposição proposta em duas relações. À primeira vista, pode parecer que o sistema com duas relações contém a mesma informação do sistema com uma só, mas isto não ocorre. Por exemplo, considere-se o problema de determinar o departamento em que um funcionário trabalhou. Isto é efetuado com facilidade no sistema com uma única relação, bastando verificar a ênupla que contém o número de identificação do funcionário desejado, e nela consultar o departamento correspondente. Entretanto, no sistema com duas relações, a informação desejada não está necessariamente disponível. Podemos localizar o nome do cargo do funcionário em questão e um departamento que possui tal cargo, mas isto não significa, necessariamente, que o funcionário tenha trabalhado nesse departamento, pois às vezes vários departamentos possuem cargos com o mesmo nome.

Em alguns casos, uma relação pode ser decomposta em relações menores, sem que isto cause uma perda de informação (esta operação é chamada **decomposição sem perda**), mas há casos em que a informação ou parte dela se perde. A classificação de tais características tem sido, e ainda é, um tópico de pesquisa na Ciência da Computação. O objetivo é identificar as características das relações que podem levar a situações problemáticas no projeto do banco de dados e encontrar meios de reorganizar essas relações para remover tais características.

Banco de dados temporais

Os bancos de dados tradicionais são usados para manter registros atuais. Um banco de dados de estoque, por exemplo, é usado para registrar o estoque corrente com o objetivo de manter os níveis de estoque adequados. Nesses casos, os registros do estoque passado freqüentemente estão disponíveis apenas em cópias de segurança do banco de dados ou quando explicitamente se guardam as datas como parte dos dados. Existem muitas aplicações, contudo, onde o acesso mais conveniente a registros antigos, bem como a facilidade de se registrar informação futura, é vantajoso. É o caso do registro acadêmico em uma universidade, onde é preciso manter informações acerca de disciplinas oferecidas, classes, salas alocadas, notas obtidas etc., não apenas para o semestre corrente, mas para os semestres passados e futuros. Os bancos de dados temporais são bancos de dados projetados para essas aplicações. Quando novos elementos entram no banco de dados temporais, a informação antiga não é apagada. Em vez disso, os novos elementos simplesmente se tornam a última informação em um registro histórico, onde qualquer parte dele é prontamente acessível.

Os bancos de dados temporais representam uma área ativa da pesquisa atual. A meta é encontrar modos eficientes de armazenar e manter a informação passada, presente e futura, desenvolver técnicas de buscar tais registros para compor a informação requerida e desenvolver linguagens para expressar as solicitações de informação temporal.

Relação FUNCIONÁRIO

IdFuncionário	Nome	Endereço	NúmSeguroSocial
25X15	Joe E. Baker	33 Nowhere St.	111223333
34Y70	Cherryl H. Clark	563 Downtown Ave.	999009999
23Y34	G. Jerry Smith	1555 Circle Dr.	111005555
•	•	•	•
•	•	•	•
•	•	•	•

Relação CARGO

IdCargo	NomeDoCargo	CódigoHabilidade	Departamento
S25X	Secretária	T5	Pessoal
S26Z	Secretária	T6	Contabilidade
F5	Gerente Júnior	FM3	Vendas
•	•	•	•
•	•	•	•
•	•	•	•

Pertencem aos departamentos de pessoal e contabilidade

Relação ATRIBUIÇÃO

IdFuncionário	IdCargo	DataEntrada	DataSaída
23Y34	S25X	01-03-1999	30-04-2001
34Y70	F5	01-10-2001	*
23Y34	S26Z	01-05-2001	*
•	•	•	•
•	•	•	•
•	•	•	•

Figura 9.6 Como encontrar os departamentos nos quais o funcionário 23Y34 trabalhou.

Operações relacionais

Agora que você tem uma compreensão básica da estrutura utilizada no modelo relacional, veremos como tal organização pode ser utilizada do ponto de vista de um programador. Iniciemos observando algumas operações que podemos executar sobre as relações.

Às vezes, precisamos selecionar certas ênuplas de uma relação. Para obter informações sobre um funcionário, devemos selecionar a ênupla que apresente o valor apropriado no atributo de identificação da relação FUNCIONARIO. Para obter uma lista dos títulos dos cargos de um departamento, devemos selecionar as ênuplas da relação CARGO em que o departamento conste no seu atributo de departamento. O resultado desta seleção é uma outra relação (outra tabela), constituída de ênuplas selecionadas da relação ancestral.

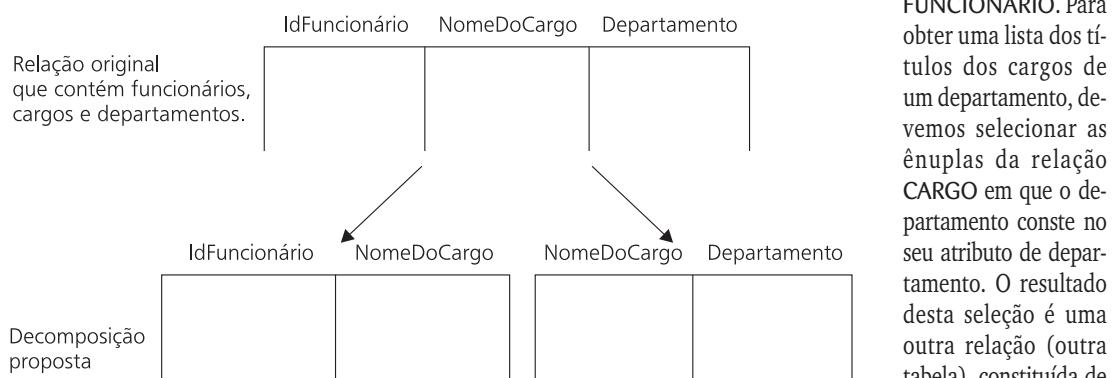


Figura 9.7 Uma relação e uma decomposição proposta.

Selecionar informação sobre um funcionário resulta em uma relação que contém somente uma das ênuplas da relação FUNCIONÁRIO. Selecionar as ênuplas associadas a um certo departamento provavelmente resultará em um subconjunto das ênuplas da relação CARGO.

Resumindo, uma das operações que desejaremos executar sobre uma relação será a de selecionar ênuplas que possuam determinadas características e organizá-las para formar uma nova relação. Para expressar esta operação, adotaremos a sintaxe:

```
NOVA ← SELECT from FUNCIONÁRIO where IdFuncionário = "34Y70"
```

A semântica desta instrução é criar uma relação nova chamada NOVA que contenha as ênuplas (deve haver somente uma neste caso) da relação FUNCIONARIO cujo atributo IdFuncionario seja igual a 34Y70 (veja a Figura 9.8).

Diferentemente da operação SELECT, que extrai linhas de uma relação, a operação PROJECT extrai colunas. Por exemplo, suponhamos que, procurando os títulos dos cargos de um certo departamento, selecionemos por meio da operação SELECT as ênuplas da relação CARGO que pertencem ao departamento em questão, e que coloquemos as ênuplas em uma nova relação, chamada NOVA1. A lista que estamos buscando é a coluna NomeDoCargo desta nova relação. A operação PROJECT nos permite extrair a informação desta coluna (ou colunas, se preciso for) e colocar o resultado em uma nova relação. Expressemos tal operação por

```
NOVA2 ← PROJECT NomeDoCargo from NOVA1
```

O resultado é a criação de outra nova relação (chamada NOVA2) que contém uma coluna dos valores encontrados na coluna NomeDoCargo da relação NOVA1.

Como outro exemplo da operação PROJECT, a instrução

```
POSTAL ← PROJECT Nome, Endereço from FUNCIONÁRIO
```

pode ser usada para obter uma lista dos nomes e endereços de todos os funcionários. Esta lista é armazenada na relação recém gerada (duas colunas) chamada POSTAL (Figura 9.9).

A terceira operação que introduziremos é a operação JOIN. É usada para combinar diferentes relações em uma só. A operação JOIN, aplicada sobre duas relações, produz uma nova relação, cujos atributos são os mesmos das relações originais (Figura 9.10). Os nomes desses atributos são os mesmos

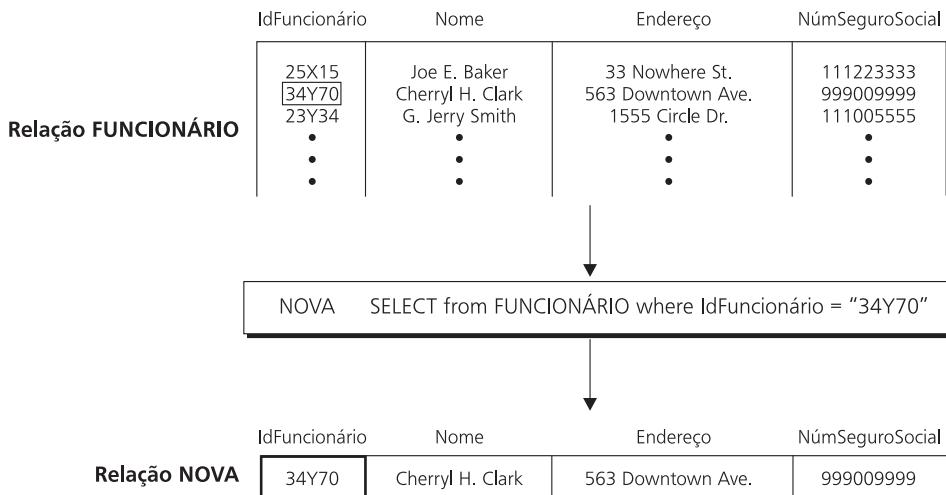


Figura 9.8 A operação SELECT.

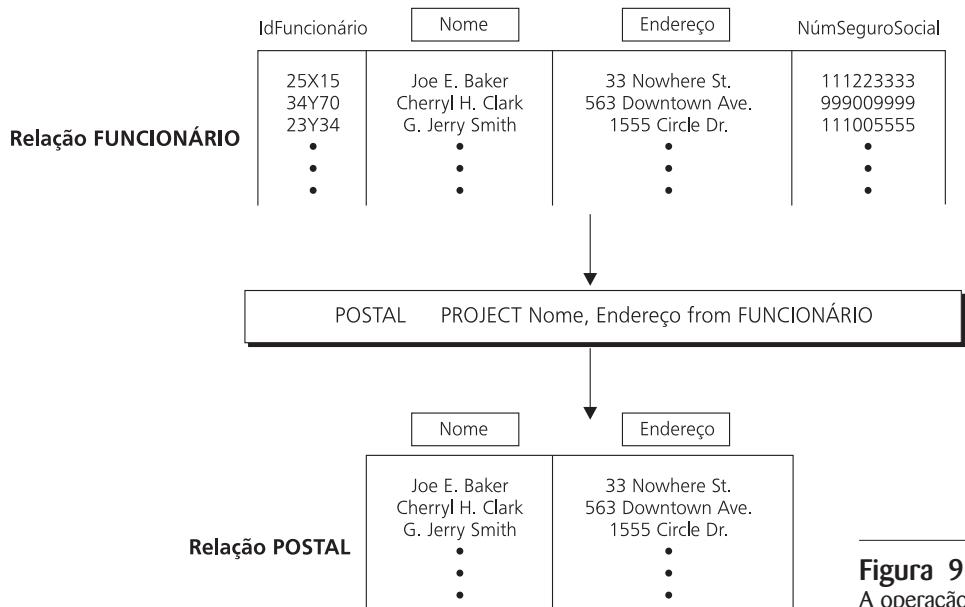


Figura 9.9
A operação PROJECT.

usados nas relações originais, exceto pelo fato de cada um apresentar como prefixo o nome da sua relação original. (Se a relação A, com os atributos V e W, for unida pela operação JOIN à relação B, que contém os atributos X, Y e Z, o resultado apresentará cinco atributos, chamados A.V, A.W, B.X, B.Y e B.Z.) Esta convenção assegura que os atributos da nova relação tenham seus próprios nomes, embora as relações originais possam apresentar atributos em comum.

As ênuplas (linhas) da nova relação são produzidas concatenando-se ênuplas das duas relações originais (Figura 9.10). As ênuplas que de fato são unidas para formar as ênuplas na nova relação são determinadas pela condição em que foi criada a operação JOIN. Uma destas condições é que os atributos tenham o mesmo valor. Com efeito, este é o caso representado na Figura 9.10, na qual demonstramos o resultado obtido pela execução da instrução

$C \leftarrow \text{JOIN } A \text{ and } B \text{ where } A.W = B.X$

Neste exemplo, uma ênupla da relação A será concatenada com uma ênupla da relação B se e somente se os atributos W e X forem iguais nas duas ênuplas correspondentes. Assim, a concatenação da ênupla (r, 2) da relação A com a ênupla (2, m, q) da relação B aparece na relação final porque o valor do atributo W na primeira ênupla é igual ao do atributo X na segunda. Por outro lado, o resultado da concatenação da ênupla (r, 2) da relação A com a ênupla (5, g, p) da relação B não aparece na relação final, porque tais ênuplas não compartilham valores comuns nos atributos W e X.

Como exemplo adicional, a Figura 9.11 apresenta o resultado da execução da instrução

$C \leftarrow \text{JOIN } A \text{ and } B \text{ where } A.W < B.X$

Note-se que as ênuplas resultantes são exatamente aquelas em que o atributo W da relação A é menor do que o atributo X da relação B.

Verifiquemos, agora, como a operação JOIN pode ser utilizada no banco de dados da Figura 9.5 para obter uma lista de todos os números de identificação de funcionários, juntamente com o departamento em que cada um trabalhou. A nossa primeira observação é a de que a informação solicitada está distribuída em mais de uma relação e, portanto, o processo de recuperação de informação necessitará de recursos adicionais às operações SELECT e PROJECT. De fato, a solução de que precisamos é a instrução

```
NOVA1 ← JOIN ATRIBUIÇÃO and
CARGO where
ATRIBUIÇÃO.NomeDoCargo =
CARGO.NomeDoCargo
```

que produz a relação NOVA1, ilustrada na Figura 9.12. A partir desta relação, o nosso problema poderá ser resolvido selecionando (operação SELECT) primeiro, as ênuplas nas quais o termo ATRIBUIÇÃO.DataSaída seja igual a “*” (que indica “ainda ativo”) e, depois, projetando (operação PROJECT) os atributos ATRIBUIÇÃO.IdFuncionario juntamente com o CARGO.Departamento. Em resumo, a informação desejada pode ser obtida do banco de dados da Figura 9.5 executando as instruções

```
NOVA1 ← JOIN ATRIBUIÇÃO and
CARGO where
ATRIBUIÇÃO.NomeDoCargo =
CARGO.NomeDoCargo
NOVA2 ← SELECT from NOVA1 where
ATRIBUIÇÃO.DataSaída = “*”
LISTA ← PROJECT ATRIBUIÇÃO.IdFuncionario, CARGO.Departamento from NOVA2
```

Tópicos de implementação

Agora que já introduzimos as operações relacionais básicas, vamos considerar a estrutura geral de um sistema de banco de dados. Convém lembrar que os dados de um banco de dados são de fato guardados em um sistema de armazenamento em massa. Para poupar o programador da aplicação dos detalhes de tais sistemas, um sistema de gerência de banco de dados permite que o *software* de aplicação seja escrito em termos do modelo do banco de dados, como o relacional que discutimos. É dever do SGBD aceitar comandos em termos do modelo relacional e convertê-los para a forma de ações relativas à verdadeira estrutura física de armazenamento dos dados. Isto é realizado mediante um conjunto de rotinas que podem ser usadas pelo *software* de aplicação como ferramentas abstratas. Assim, um SGBD baseado no modelo relacional poderá oferecer ao seu usuário rotinas para executar as operações SELECT, PROJECT e JOIN, as quais poderiam ser então chamadas pelo *software* de aplicação através de estruturas sintáticas compatíveis com a linguagem hospedeira. Dessa maneira, o *software* de aplicação pode ser escrito como se os dados realmente estivessem armazenados na forma das tabelas simples do modelo relacional.

É instrutivo considerar como um sistema de gerência de banco de dados pode armazenar

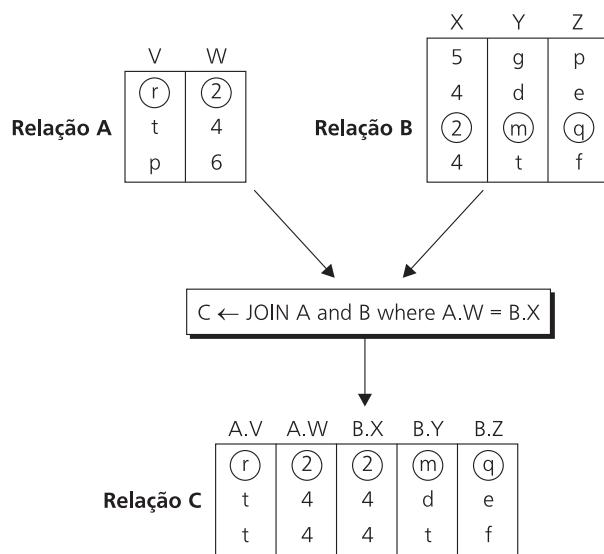


Figura 9.10 A operação JOIN.

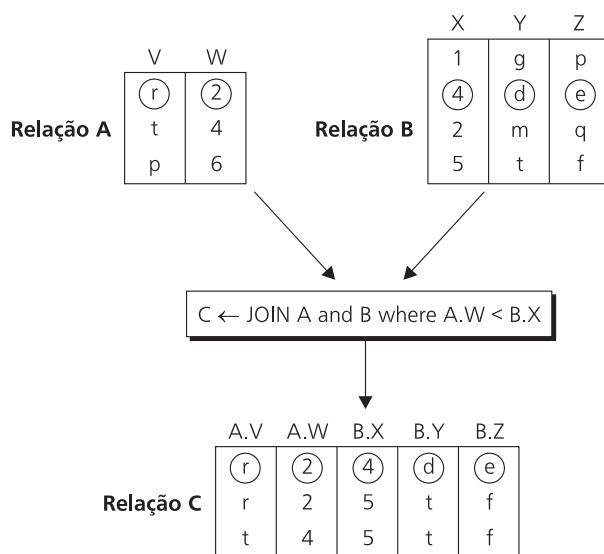


Figura 9.11 Outro exemplo da operação JOIN.

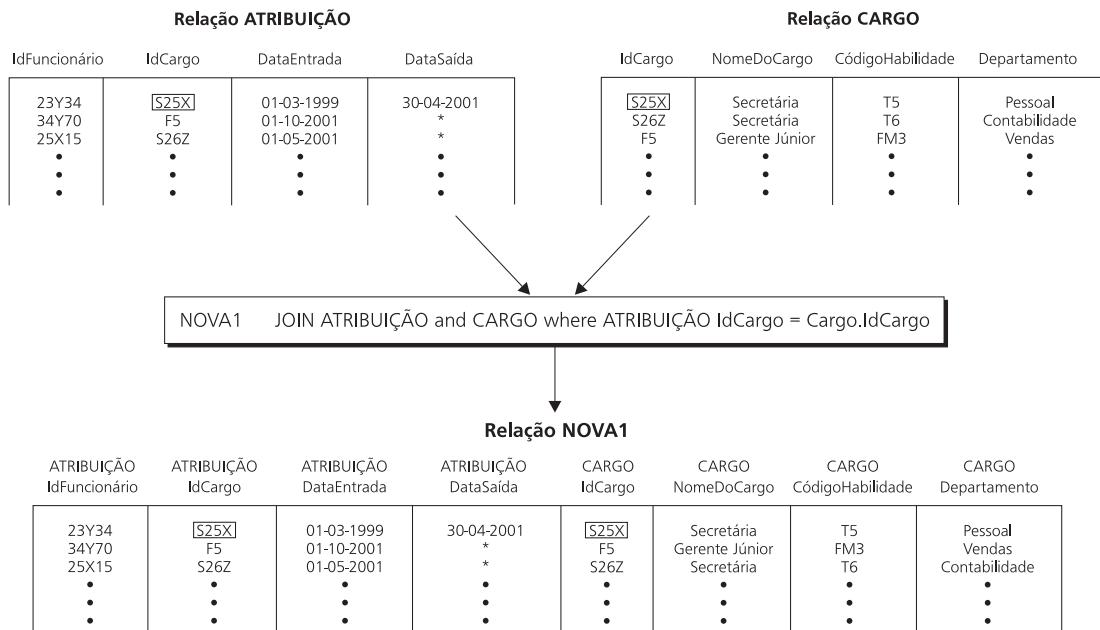


Figura 9.12 Uma aplicação da operação JOIN.

os dados em um banco de dados e como esse armazenamento afeta as suas operações. Por exemplo, a maneira mais simples de um SGBD implementar uma relação é armazená-la como arquivo seqüencial no qual cada ênupla é um registro lógico. Contudo, seguindo essa estratégia, a execução de uma operação SELECT necessitaria uma busca seqüencial no arquivo — processo este que consumiria muito tempo no caso de uma relação grande. Assim, é provável que o SGBD guarde a relação como um arquivo indexado, ou talvez utilize técnicas de *hashing* para prover acesso rápido aos elementos. Por exemplo, se a relação FUNCIONÁRIO na Figura 9.5 for indexada pelo número de identificação do funcionário, então a informação de um funcionário em particular poderá ser selecionada rapidamente se o seu número de identificação for conhecido. Isso, é claro, são tópicos significativos de projeto, que freqüentemente dependem de como o banco de dados será utilizado. O ponto é que os fundamentos de estruturas de dados e estruturas de arquivos proporcionam as bases nas quais os sistemas de gerência de banco de dados são construídos.

Finalmente, devemos notar que os modernos sistemas de gerência de bancos de dados não apresentam necessariamente as operações SELECT, PROJECT e JOIN na sua forma mais simples. Ao contrário, eles apresentam operações que podem ser combinações destes passos básicos. Um exemplo é a linguagem SQL.

SQL

A linguagem chamada SQL (Structured Query Language)* é extensivamente usada pelos programadores de *software* de aplicação para manipular bancos de dados vistos em termos do modelo relacional. Uma razão para sua popularidade é o fato de ter sido padronizada pelo American National Standards

*N. de T. Em português, Linguagem de Consulta Estruturada.

Institute (ANSI). Outra razão foi o fato de ter sido originalmente desenvolvida e comercializada pela IBM, e assim beneficiada por uma vasta divulgação. Nesta seção, explicaremos como as consultas aos bancos de dados são expressas em SQL.

A nossa primeira observação é que uma consulta que envolve uma seqüência de operações SELECT, PROJECT e JOIN, pode ser expressa por um único comando de SQL. Além disso, embora uma cláusula escrita em SQL seja expressa de uma forma aparentemente imperativa, é na realidade um comando essencialmente declarativo. Deve-se ler um comando SQL como uma descrição da informação desejada, em vez de uma seqüência de atividades a serem realizadas. Isso significa que a SQL poupa os programadores de aplicação da obrigatoriedade de desenvolver algoritmos para manipular relações — eles precisam simplesmente descrever a informação desejada.

Como nosso primeiro exemplo de comando SQL, consideremos nossa última consulta, em que desenvolvemos um processo de três passos para obter todos os números de identificação de funcionários, juntamente com os respectivos departamentos. Em SQL, a consulta inteira poderia ser descrita com um único comando:

```
select IdFuncionario, Departamento
  from ATRIBUICAO, CARGO
 where ATRIBUICAO.IdCargo = CARGO.IdCargo
   and ATRIBUICAO.DataSaida = '*'
```

Conforme está ilustrado neste exemplo, cada comando de consulta em SQL pode conter três cláusulas — select, from e where. Analisando em termos gerais, tal comando é um pedido para realizar a operação JOIN de todas as relações listadas na cláusula from, a operação SELECT das ênuplas que satisfazem às condições da cláusula where e, então, a operação PROJECT de tais ênuplas levantadas na cláusula select. (Note-se que a terminologia fica um tanto invertida, porque a cláusula select em um comando de SQL identifica os atributos usados na operação PROJECT.) Consideremos alguns exemplos simples.

O comando

```
select Nome, Endereco
  from FUNCIONARIO
```

produz uma lista com todos os nomes e endereços dos funcionários contidos na relação FUNCIONARIO. Note-se que ele é apenas uma operação PROJECT.

O comando

```
select IdFuncionario, Nome, Endereco, NumSeguroSocial
  from FUNCIONARIO
 where Nome = 'Cheryl H. Clark'
```

extrai toda a informação da ênupla associada a Cheryl H. Clark da relação EMPREGADO. Este comando é, essencialmente, uma operação SELECT.

O comando

```
select Nome, Endereco
  from FUNCIONARIO
 where Nome = 'Cheryl H. Clark'
```

extrai o nome e o endereço de Cheryl H. Clark como está contido na relação FUNCIONARIO. Este comando é uma combinação das operações SELECT e PROJECT.

O comando

```
select FUNCIONARIO.Nome, ATRIBUICAO.DataEntrada
  from FUNCIONARIO, ATRIBUICAO
 where FUNCIONARIO.IdFuncionario = ATRIBUICAO.IdFuncionario
```

produz uma lista com todos os nomes dos funcionários e suas datas de ingresso na companhia. Note-se que este é o resultado da junção (JOIN) das relações FUNCIONARIO e ATRIBUICAO e, em seguida, da seleção (SELECT) e projeção (PROJECT) das ênuplas apropriadas e atributos descritos pelas cláusulas where e select.

Percebemos que a SQL inclui comandos para definir a estrutura das relações, criar relações e modificar o conteúdo das mesmas, bem como executar consultas ao banco de dados. Apresentamos a seguir exemplos de comandos insert into, delete from e update.

O comando

```
insert into FUNCIONARIO  
values ('42Z12', 'Sue A. Burt', '33 Fair St.', '444661111')
```

acrescenta uma ênupla à relação FUNCIONARIO que contém os valores indicados;

```
delete from FUNCIONARIO  
where Nome = 'G. Jerry Smith'
```

remove a ênupla relativa a G. Jerry Smith da relação FUNCIONARIO; e

```
update FUNCIONARIO  
set Endereco = '1812 Napoleon Ave.'  
where Nome = 'Joe E. Baker'
```

modifica o endereço da ênupla associada a Joe E. Baker da relação FUNCIONARIO.



QUESTÕES/EXERCÍCIOS

1. Responda às seguintes perguntas, baseando-se na informação parcial dada pelas relações FUNCIONARIO, CARGO e ATRIBUICAO da Figura 9.5:
 - a. Quem é a secretária do departamento de contabilidade com experiência em departamento de pessoal?
 - b. Quem é o gerente do departamento de vendas?
 - c. Qual é o cargo que G. Jerry Smith exerce no momento?
2. Baseado nas relações FUNCIONARIO, CARGO e ATRIBUICAO, apresentadas na Figura 9.5, escreva uma sequência de operações relacionais para obter uma lista de todos os nomes de cargos dentro do departamento de pessoal.
3. Baseado nas relações FUNCIONARIO, CARGO e ATRIBUICAO, apresentadas na Figura 9.5, escreva uma sequência de operações relacionais para obter uma lista de nomes de funcionários ao lado dos respectivos departamentos.
4. Converta suas respostas às perguntas 2 e 3 para SQL.
5. Como o modelo relacional permite independência de dados?
6. Como as diferentes relações em um banco de dados relacional são interligadas?

9.4 Bancos de dados orientados a objeto

Uma das áreas mais novas na pesquisa de bancos de dados envolve a aplicação do paradigma orientado a objeto à construção de um banco de dados. O resultado é um **banco de dados orientado a objeto**, que consiste em objetos que são interligados para refletir seus relacionamentos. Por exemplo, uma implementação orientada a objeto do banco de dados de funcionários das seções anteriores poderia

consistir em três classes (tipos de objetos): FUNCIONARIO, CARGO e ATRIBUICAO. Um objeto da classe FUNCIONARIO poderia conter elementos como IdFuncionario, Nome, Endereco e NumSeguroSocial; um objeto da classe CARGO, elementos como IdCargo, NomeDoCargo, CodHabilidade e Departamento; e cada objeto da classe ATRIBUICAO, elementos como DataEntrada e DataSaida.

Uma representação conceitual de tal banco de dados é mostrada na Figura 9.13 onde os relacionamentos entre os vários objetos são representados por linhas que ligam os objetos. Se focalizarmos um objeto do tipo FUNCIONARIO, veremos que ele está ligado a uma coleção de objetos do tipo ATRIBUICAO, que representa as várias atribuições daquele funcionário em particular. Por sua vez, cada objeto do tipo ATRIBUICAO está ligado a um objeto do tipo CARGO, que representa o cargo associado com aquela atribuição. Assim, todas as atribuições de um funcionário podem ser encontradas seguindo as ligações a partir do objeto que o representa. Do mesmo modo, todos os funcionários que tiveram um cargo específico podem ser encontrados seguindo as ligações a partir do objeto que representa o cargo.

As ligações entre objetos em um banco de dados orientado a objeto normalmente são mantidas pelo SGBD, de modo que os detalhes de como essas ligações são implementadas não interessam ao programador que escreve o *software* de aplicação. Ao contrário, quando um novo objeto é acrescentado ao banco de dados, o *software* de aplicação necessita simplesmente especificar os outros objetos aos quais ele deve ser interligado. O SGBD então cria o sistema de ponteiros necessário para registrar essas associações. Especificamente, o SGBD pode interligar os objetos que representam as atribuições de um funcionário de maneira similar a uma lista ligada.

Outra tarefa de um SGBD orientado a objeto é prover armazenamento permanente para os objetos confiados a ele — uma exigência que pode parecer óbvia, mas é inherentemente distinta da maneira como os objetos normalmente são tratados. Em geral, quando um programa orientado a objeto é executado, os objetos usados durante a execução são descartados quando o programa termina. Nesse sentido, os objetos são considerados transientes. Entretanto, os objetos que são criados e acrescentados a um banco de dados devem ser persistentes — isto é, devem ser salvos após o término do programa que os criou. Por isso, prover armazenamento para objetos é um desvio significativo do normal.

Os proponentes dos bancos de dados orientados a objeto oferecem numerosos argumentos para afirmar que a abordagem orientada a objeto é melhor que a relacional. Um deles é que a abordagem orientada a objeto permite que o sistema de *software* inteiro (*software* de aplicação, SGBD e o próprio banco de dados) seja projetado no mesmo paradigma. Isso contrasta com a prática historicamente comum de se usar uma linguagem de programação imperativa para desenvolver o *software* de consulta a um banco de dados relacional. Inerente a essa tarefa está o conflito entre os paradigmas imperativo e relacional. Essa distinção é útil a nosso nível de estudo, mas tem sido a causa de muitos erros de *software* ao longo dos anos. Podemos, contudo, apreciar o fato de que um banco de dados orientado a objeto combinado com um programa de aplicação também orientado a objeto produz uma imagem homogênea de objetos dispersos no sistema que se comunicam com os outros, enquanto um banco de

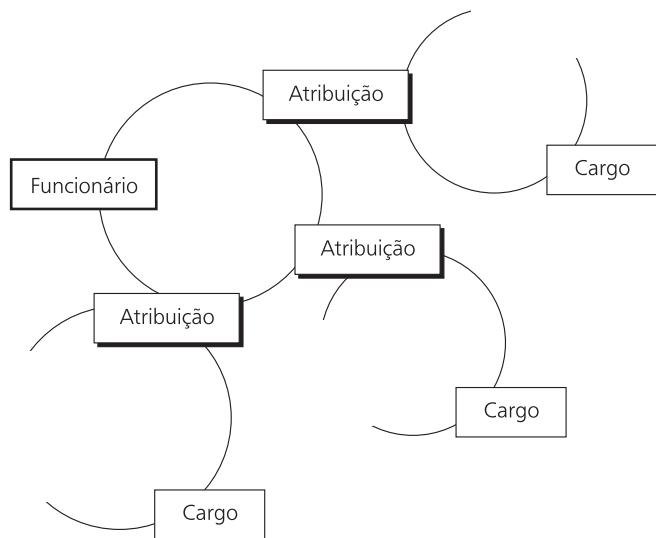


Figura 9.13 As associações entre objetos em um banco de dados orientado a objeto.

dados relacional combinado com um programa de aplicação imperativo dá imagem de duas organizações inherentemente diferentes que tentam encontrar uma interface em comum.

Para apreciar outra vantagem que os bancos de dados orientados a objeto têm sobre os relacionais, considere o problema de armazenar nomes de funcionários em um banco de dados relacional. Se o nome completo for armazenado como um único atributo em uma relação, então as consultas que consideram apenas sobrenomes serão incômodas. Por outro lado, se o nome for armazenado como três atributos separados — que permitiria um primeiro nome, nome do meio e sobrenome — então ficará complicado lidar com pessoas cujos nomes não se conformam com esse padrão de primeiro nome, nome do meio e sobrenome. Em um banco de dados orientado a objeto, esses tópicos podem ficar escondidos nos objetos que manipulam os nomes dos funcionários. O armazenamento pode ser feito por um objeto inteligente que é capaz de reportar o nome relacionado em diversos formatos. Assim, de fora desses objetos, seria igualmente fácil lidar apenas com sobrenomes, com nomes completos, nomes de solteiro, ou apelidos. Os detalhes envolvidos em cada perspectiva seriam encapsulados dentro dos objetos.

Essa habilidade de encapsular as tecnicidades de diferentes formatos de dados também é vantajosa em outros casos. Em um banco de dados relacional, os atributos em uma relação fazem parte do projeto completo do banco, e assim os tipos associados a eles permeiam o banco de dados inteiro. (Variáveis para armazenamento temporário devem ser declaradas com o tipo apropriado, e procedimentos para manipular dados de vários tipos devem ser projetados.) Assim, estender um banco de dados relacional para incluir atributos de novos tipos (áudio e vídeo) às vezes é problemático. De fato, vários procedimentos no projeto do banco de dados podem necessitar ser estendidos para incorporar esses novos tipos de dados. Em um projeto orientado a objeto, porém, os mesmos procedimentos usados para recuperar um objeto que representa um nome de funcionário são usados para recuperar um objeto que representa um filme, porque a distinção de tipos pode ficar oculta nos objetos envolvidos. Assim, a abordagem orientada a objeto parece mais compatível com a construção de bancos de dados multimídia — característica que já demonstrou ser uma grande vantagem.

Ainda outra vantagem que o paradigma orientado a objeto oferece ao projeto de bancos de dados é o potencial para armazenar objetos inteligentes, em vez de simplesmente dados. Isto é, um objeto pode conter métodos que descrevam como ele deve responder a mensagens relativas a seu conteúdo e seus relacionamentos. Por exemplo, cada objeto da classe **FUNCIONARIO** na Figura 9.13 poderia conter métodos para reportar e atualizar a informação no objeto, bem como um método para reportar o histórico de cargos do funcionário, ou talvez um método para mudar a sua atribuição de cargo. Do mesmo modo, cada objeto da classe **CARGO** teria um método para reportar as especificidades do cargo e talvez um método para reportar os funcionários que exerceram aquele cargo específico. Assim, para recuperar o histórico de cargos de um funcionário, não necessitariam escrever um procedimento extenso que descrevesse como a informação deve ser obtida. Em vez disso, simplesmente pediríamos ao objeto apropriado do funcionário que reporte o seu histórico de cargos. Assim, a habilidade de construir bancos de dados cujos componentes respondem intelligentemente a consultas oferece um conjunto empolgante de possibilidades além daquelas dos bancos de dados mais tradicionais.



QUESTÕES/EXERCÍCIOS

1. Quais métodos podem estar em uma instância de um objeto da classe **ATRIBUICAO** do banco de dados de funcionários discutido nesta seção?
2. Identifique algumas classes, bem como algumas das suas características internas, que possam ser utilizadas para cuidar do estoque de um armazém, em um banco de dados orientado a objeto.
3. Identifique uma vantagem que um banco de dados orientado a objetos pode ter em relação a um relacional.

9.5 A preservação da integridade de bancos de dados

Os sistemas baratos de gerência de bancos de dados para uso pessoal são relativamente simples. Eles tendem a possuir um único objetivo — poupar o usuário dos detalhes técnicos da implementação do banco de dados. Os bancos de dados mantidos por tais sistemas são relativamente pequenos e geralmente contêm informação cuja perda ou falta de integridade seria apenas inconveniente, sem danos maiores. Quando surge um problema, o usuário costuma corrigir os elementos errados diretamente, ou então recarrega o banco de dados a partir de uma cópia auxiliar e faz manualmente as modificações necessárias para manter atualizada tal cópia. Esse processo pode ser inconveniente, mas o custo de evitar a inconveniência tende a ser maior do que a própria inconveniência. Em todo caso, a inconveniência é restrita somente a algumas pessoas, e qualquer perda financeira geralmente é pequena.

No caso de grandes sistemas de bancos de dados comerciais multiusuários, entretanto, os riscos são muito mais altos. O custo de dados incorretos ou perdidos pode ser enorme e acarretar consequências devastadoras. Nesses ambientes, um dos principais papéis do SGBD é manter a integridade do banco, resguardando-o contra problemas como operações que por alguma razão não são executadas até o fim, ou operações diversas, que, ao interagir indevidamente, provoquem o aparecimento de inconsistências na informação contida no banco de dados. É sobre este papel dos SGBDs que discorreremos a seguir.

O protocolo *commit/rollback*

Uma única transação, seja ela a transferência de capital de uma conta bancária para outra, o cancelamento de uma reserva de passagem aérea ou a inscrição de um estudante em um curso universitário, pode envolver múltiplos passos no nível de banco de dados. Por exemplo, uma transferência de capital entre contas bancárias requer que no saldo de uma das contas seja feito um débito, e na outra, o crédito correspondente. No intervalo de tempo transcorrido entre esses dois passos, a informação no banco de dados poderá ficar inconsistente. De fato, um capital estará faltando durante o breve período após a primeira conta ser debitada e antes de a outra ser creditada. Da mesma forma, durante a alteração da designação da poltrona de um passageiro em um vôo, pode haver um momento em que o passageiro não possui qualquer assento disponível, ou então, um momento em que a lista de passageiros aparenta conter um passageiro a mais.

No caso de grandes bancos de dados, sujeitos a uma imensa quantidade de transações, é altamente provável que, em um instante aleatório, alguma transação esteja em curso no banco de dados. Existe, portanto, a possibilidade de ocorrência de alguma solicitação para a execução de uma transação, ou de um mau funcionamento do equipamento, exatamente em algum desses instantes em que o banco de dados se encontra inconsistente.

Consideremos, primeiramente, o problema de um mau funcionamento. O objetivo do SGBD é assegurar que tal ocorrência não torne definitivo o estado de um banco de dados inconsistente. Em geral, isto é possível mantendo, em um sistema de armazenamento não volá-

Bancos de dados espaciais

Suponha que você deseja projetar um sistema CAD para ser usado por engenheiros no desenvolvimento de aviões. O sistema deve armazenar, manipular e exibir informações de objetos em três dimensões. Por exemplo, alguém pode querer descrever e modificar vários projetos de asa, observar as imagens desses projetos a partir de diversas perspectivas e talvez obter os resultados de uma análise de esforço exibidos graficamente na tela do computador. O banco de dados subjacente deve, portanto, armazenar a informação acerca dos vários pontos em um ambiente tridimensional, de forma a suportar as exigências do software de aplicação. Tais bancos de dados são chamados bancos de dados espaciais.

Eles possuem numerosas aplicações em potencial (incluindo previsão meteorológica e análise de censo) e são, portanto, uma área ativa de pesquisa corrente. Como acontece com os bancos de dados temporais, as metas são encontrar meios eficientes de armazenamento e manutenção da informação espacial, desenvolver técnicas de acesso a tais registros e desenvolver linguagens para expressar as solicitações de informação espacial.

til como um disco, por exemplo, uma lista completa dos registros das atividade executadas em cada transação*. Antes que uma transação possa alterar o banco de dados, ela é, primeiro, registrada nesse diário. Assim, o diário contém um registro permanente das ações de cada transação.

O ponto em que todos os passos de uma transação já se encontram armazenados no diário é chamado de **ponto de comprometimento**.** É neste momento que o SGBD tem a informação para reconstruir a transação por si próprio caso seja necessário. Ele fica comprometido com a transação, no sentido em que assume a responsabilidade de garantir que as atividades da transação serão refletidas no banco de dados. No caso de um mau funcionamento no equipamento, o SGBD pode usar a informação contida no seu diário para reconstruir, a partir da última cópia de segurança, as transações com as quais se comprometeu.

Se ocorrer um problema antes de uma transação atingir o seu ponto de comprometimento, o SGBD pode ficar com uma transação parcialmente executada, que não possa ser completada. Neste caso, o diário será utilizado para **desfazer***** as ações já executadas pela transação. No caso de um mau funcionamento, por exemplo, o SGBD recupera o seu estado no momento do mau funcionamento, desfazendo transações incompletas (não-comprometidas).

O recurso de desfazer transações não se restringe apenas a casos de mau funcionamento do equipamento, mas em geral faz parte da operação normal de um SGBD. Por exemplo, uma transação pode ser interrompida antes de completar todos os seus passos em decorrência de uma tentativa de acesso a alguma informação privilegiada, ou estar envolvida em um enlace mortal em que transações que competem pelos mesmos dados se encontrem à espera da liberação dos mesmos. Nestes casos, o SGBD pode usar o diário para desfazer uma transação e assim evitar a produção de um banco de dados inconsistente em virtude de transações não completadas.

Para enfatizar a natureza delicada do projeto de SGBD, devemos notar a existência de problemas sutis no processo de desmanchar (anular) o efeito de transações. Desfazer uma transação pode afetar elementos do banco de dados que já tenham sido utilizados por outras transações. Por exemplo, a transação que estiver sendo desfeita talvez já tenha atualizado o saldo de uma conta bancária, e uma outra transação, baseado suas ações neste valor atualizado. Isto pode significar que tais transações adicionais também deverão ser desfeitas, o que às vezes tem efeitos adversos sobre outras transações. O resultado é o problema conhecido como **anulação em cascata**.****

Trancamento*****

Consideremos, agora, o problema de uma transação ser executada enquanto o banco de dados está em um estado de contínuas alterações por parte de uma outra transação, situação essa que pode produzir uma interação indevida entre as transações e, assim, conduzir a resultados errados. Por exemplo, um problema, conhecido como o **problema do sumário incorreto*******, surge se, durante uma transação de transferência de capital entre duas contas, outra transação tentar calcular os depósitos totais no banco. Isto levaria à obtenção de um total grande demais ou pequeno demais, dependendo da ordem em que os passos da transferência forem executados. Outra possibilidade é o **problema da atualização perdida*******, exemplificado por duas transações, que tiram conclusões a partir da mesma conta. Se uma transação ler o saldo atual da conta no instante em que a outra acabou de ler, porém

*N. de T. Esta lista é conhecida, em inglês, como *log* — ao pé da letra, em português, *diário de bordo*.

**N. de T. Em inglês, *commit*.

***N. de T. Em inglês, *rollback (undo)*.

****N. de T. Em inglês, *cascaded rollback*.

*****N. de T. Em inglês, *locking*.

*****N. de T. Em inglês, *incorrect summary problem*.

*****N. de T. Em inglês, *lost update problem*.

sem ter calculado ainda o novo saldo, então ambas tirarão suas conclusões com base no mesmo saldo inicial. Entretanto, o efeito de uma dessas deduções não ficará refletido no banco de dados.

Para resolver tais problemas, um SGBD poderia forçar a execução das transações na íntegra, uma por vez, mantendo cada nova transação em uma fila, até que as precedentes tenham sido completadas. Todavia, cada uma fica muito tempo esperando pela execução de operações em disco. Com a alternância da execução de transações, o tempo que uma transação gasta aguardando para que possa continuar a sua execução pode ser preenchido por uma outra, para processar dados já extraídos. Portanto, a maioria dos sistemas maiores de gerência de bancos de dados contém um escalador para coordenar o compartilhamento de tempo entre as transações, da mesma forma que um sistema operacional de tempo partilhado coordena a alternância de processos.

Para se resguardar contra anomalias como os dois problemas acima mencionados, tais escaladores incorporam um **protocolo de trancamento**^{*}, no qual os elementos de um banco de dados que no momento estiverem em uso por alguma transação ficam marcados como tal. Estas marcas são chamadas trancas^{**}, e os elementos assim marcados são denominados elementos trancados. Dois tipos de trancas são comuns — as **trancas compartilhadas** e as **trancas exclusivas**. Elas correspondem aos dois tipos de acesso que uma transação pode solicitar a um elemento — acesso compartilhado e acesso exclusivo. Se uma transação não for alterar o elemento, então ela solicitará acesso compartilhado, o que significa que as demais podem ler o mesmo elemento. No entanto, se a transação tiver de alterar o elemento, ela terá acesso exclusivo, ou seja, será a única transação com acesso a tal elemento.

Em um protocolo de trancamento, todas as vezes que uma transação solicitar acesso a um dado, também deverá comunicar ao SGBD o tipo de acesso desejado. Se uma transação solicitar acesso compartilhado a um dado não-trancado, ou trancado com uma tranca compartilhada, tal acesso será concedido, e o dado ficará marcado com uma tranca compartilhada. Se o dado solicitado já estiver marcado com uma tranca exclusiva, o acesso adicional será negado. Se uma transação solicitar acesso exclusivo a um dado, tal pedido somente será atendido se não houver qualquer tranca associada a ele. Desta forma, uma transação que deva alterar um dado o protegerá contra o acesso de outras transações, mediante o acesso exclusivo. Ao mesmo tempo, várias transações poderão compartilhar o acesso a um dado se nenhuma delas tiver a intenção de o modificar. É natural que, uma vez terminada uma transação sobre um dado, esta notifique o SGBD, sendo então removida a tranca correspondente, associada a este dado.

Vários algoritmos são utilizados para tratar o caso em que um pedido de acesso por parte de uma transação é rejeitado. Um desses algoritmos simplesmente força a transação a aguardar até que o dado desejado fique disponível. Esta abordagem pode conduzir a um enlace mortal, uma vez que duas transações que solicitem acessos exclusivos a um mesmo par de dados podem se bloquear mutuamente, pois cada uma, ao obter acesso exclusivo a um dos dados, irá insistir em esperar pela outra. Para evitar tais enlaces mortais, alguns sistemas de gerência de bancos de dados dão prioridade às transações mais antigas, isto é, se uma transação mais antiga solicitar acesso a um dado que foi trancado por uma mais recente, esta última será forçada a liberar todos os seus dados, suas ações serão desfeitas (com o auxílio do diário), a transação mais antiga será autorizada a efetuar o acesso ao dado e a mais recente, forçada a recomeçar. Se por essa razão uma transação mais recente tiver seus direitos confiscados várias vezes, ela se tornará gradativamente mais antiga até que, em última instância, venha a se tornar uma das mais antigas no sistema. Este protocolo, conhecido como **wound-wait*** protocol** (transações antigas roubam o lugar de transações recentes, forçando estas a esperar pelas antigas), assegura que toda transação terá, em alguma ocasião, o direito de completar sua tarefa.

*N. de T. Em inglês, *locking protocol*.

**N. de T. Em inglês, *locks*.

***N. de T. Em inglês, *wound* está empregado no sentido de usurpar, prejudicar, roubar o lugar.



QUESTÕES/EXERCÍCIOS

1. Qual a diferença entre uma transação que atingiu o seu ponto de comprometimento e outra que não chegou a tanto?
2. Como um SGBD poderia resguardar-se de extensas anulações em cascata?
3. Mostre de que forma a alternância descontrolada entre duas transações, sendo que uma debita \$100 de uma conta e a outra debita \$200 da mesma conta, poderia produzir saldos finais de \$100, \$200 e \$300, supondo que o saldo inicial seja de \$400.
4. a. Resuma os possíveis resultados de uma transação que solicita acesso compartilhado a um elemento de um banco de dados.
b. Resuma os possíveis resultados de uma transação que solicita acesso exclusivo a um elemento de um banco de dados.
5. Descreva uma seqüência de eventos capaz de levar a um enlace mortal entre transações que estejam executando operações em um sistema de banco de dados.
6. Descreva como o enlace mortal da sua resposta à questão 5 poderia ser eliminado. Sua solução faz uso do diário do sistema de gerência do banco de dados? Justifique a sua resposta.

9.6 Impacto social da tecnologia de banco de dados

No passado, as coleções de dados eram vistas como entidades inertes e passivas. Cada uma era projetada e usada com um propósito específico. Uma biblioteca local mantinha uma lista física com nomes e endereços de seus usuários. Cada livro continha um cartão removível com o nome do livro. Quando ele era emprestado, o cartão era removido, o nome do usuário era registrado no cartão e este era arquivado na biblioteca. Quando o livro retornava, o cartão era colocado de volta; se o livro não retornasse no prazo, os funcionários da biblioteca poderiam identificar o usuário por meio de sua lista física.

Com esse sistema manual, era possível obter uma lista de todos os livros emprestados a um único indivíduo. Entretanto, isso exigia procurar em todos os livros da biblioteca para ver que cartões continham o nome do indivíduo, e o custo dessa pesquisa a tornava impraticável. Assim, embora os registros da biblioteca contivessem informações sobre os hábitos de leitura de seus usuários, estes ficavam seguros de que tal informação era privada. Atualmente, porém, a maioria das bibliotecas é informatizada e o perfil do的习惯 de leitura de um indivíduo é fácil de acessar. Agora é possível que as bibliotecas fornecam tais informações a empresas comerciais, agências locais, partidos políticos, empregadores e indivíduos diversos. As ramificações são enormes.

Esse exemplo da biblioteca é representativo do potencial que permeia o espectro completo das aplicações de banco de dados. A tecnologia facilitou a coleta de dados e a fusão ou comparação de diferentes coleções de dados para obter relacionamentos que de outra forma ficariam escondidos.

As coleções de dados agora são conduzidas em larga escala. Em alguns casos, o processo é prontamente aparente; em outros, é mais sutil. Exemplos dos primeiros ocorrem quando as solicitações de informação são explícitas. Isso pode ser feito de maneira voluntária, por meio de formulários de pesquisa, ou de maneira involuntária, quando imposto por exigência legal. Algumas vezes, o fato de ser ou não voluntário depende do ponto de vista. O fornecimento de informação pessoal, quando se reivindica um empréstimo, é voluntário ou involuntário? A distinção depende de o empréstimo ser uma conveniência ou uma necessidade. Para usar cartão de crédito em algumas lojas, atualmente se exige que a assinatura seja registrada em um formato digitalizado. Mais uma vez, o fornecimento da informação é voluntário ou involuntário, dependendo da situação do comprador.

Casos mais sutis de coleções de dados evitam a comunicação direta com os envolvidos. Exemplos incluem uma companhia de crédito que registra as práticas de compra dos usuários de seus cartões, um

sítio na Web que registra as identidades de quem o visita e ativistas sociais que registram as placas dos carros estacionados no pátio de uma determinada instituição. Nesses casos, o envolvido pode desconhecer que a informação está sendo coletada e tem menos probabilidade de saber da existência do banco de dados que está sendo construído.

Algumas vezes, as atividades subjacentes de coleta de dados são evidentes se a pessoa parar para pensar. Por exemplo, uma mercearia pode oferecer descontos aos clientes regulares que se cadastrarem previamente. O processo de cadastro pode envolver a confecção de cartões de identificação que devem ser apresentados no momento da compra para se ter o desconto. Como resultado, a mercearia está apta a registrar as compras dos clientes — registro esse cujo valor excede em muito o valor do desconto.

Obviamente, a força que impele essa demanda por coleta é o valor intrínseco dos dados; ela é amplificada pelos avanços da tecnologia dos bancos de dados que permitem que os dados sejam interligados, de modo a revelar informações que de outra forma permaneceriam desconhecidas. Por exemplo, os hábitos de compra dos portadores de cartão de crédito podem ser classificados e usados para se obter o perfil de consumidores, com grande valor de mercado. Formulários de lojas especializadas em produtos para modelar o corpo podem ser enviados a quem recentemente comprou equipamento de ginástica, enquanto formulários similares de empresas especializadas em adestrar cães podem ser enviados a pessoas que recentemente compraram ração para cachorro. As maneiras alternativas de se combinar informação às vezes são muito imaginativas. Os registros do serviço de assistência social têm sido comparados com os registros criminais para encontrar e prender os que transgrediram as normas da liberdade condicional e, em 1984, o serviço de alistamento militar nos Estados Unidos usou a lista de aniversários de uma cadeia de sorveterias para identificar os cidadãos que se recusaram a se alistar.

Existem muitas abordagens para proteger a sociedade do uso abusivo dos bancos de dados. Uma delas é a aplicação de recursos legais. Infelizmente, a elaboração de uma lei contra uma ação não impede a sua ocorrência, mas simplesmente a torna ilegal. Um exemplo marcante nos Estados Unidos é o *Privacy Act* de 1974, cujo propósito era proteger os cidadãos do uso abusivo dos bancos de dados governamentais. Uma cláusula dessa lei era a exigência de que as agências governamentais publicassem a existência de seus bancos de dados no Registro Federal — o objetivo era permitir aos cidadãos ter acesso e corrigir a informação acerca deles próprios. Contudo, as agências governamentais têm sido morosas no cumprimento da lei. Isso não implica necessariamente intenções maliciosas. Em muitos casos, o problema é a burocacia. Entretanto, o fato de a burocacia construir bancos de dados pessoais que ela não pode identificar não é uma situação tranqüilizante.

Outra abordagem, talvez mais efetiva, para controlar o uso dos bancos de dados é a opinião pública. Os bancos de dados não serão abusivos se as penalidades pesarem mais do que os benefícios, e uma penalidade que as empresas temem muito é a opinião pública — ela atinge diretamente as bases. Nos idos de 1990, foi a opinião pública que suspendeu definitivamente a venda de listas pessoais pelas agências de crédito. Mais recentemente, a America Online (um grande provedor da Internet) se dobrou ante a pressão pública contra a sua política de vender informação relacionada aos consumidores para agências de telemarketing. Até agências governamentais se curvam à opinião pública. Em 1997, a administração do Seguro Social nos Estados Unidos modificou os seus planos e tornou os seus registros disponíveis pela Internet, quando a opinião pública questionou a segurança da informação.

Mineração de dados

A mineração de dados (*data mining*) é o processo de extrair informação de coleções de dados. Ela tem se tornado ferramenta em contextos que podem parecer pouco comuns, exemplificados por seu uso na identificação de funções de genes específicos, codificadas em moléculas de DNA, e na caracterização das propriedades de organismos como os vírus. A mineração de dados freqüentemente toma a forma de reconhecimento de padrões e determinação sobre se esses padrões são significativos ou meras coincidências. Por exemplo, o fato de uma casa lotérica específica ter vendido muitos bilhetes premiados provavelmente não será considerado significativo, enquanto a descoberta de que os clientes que compram alimentos para lanches também tendem a comprar pratos congelados pode constituir informação significativa para um gerente de mercearia.

Nesses casos, os resultados foram obtidos em dias — violento contraste ao tempo associado aos processos legais.

É claro, em muitos casos, as aplicações de bancos de dados são benéficas, tanto ao mantenedor quanto ao envolvido nos dados, mas em todos os casos existe uma perda de privacidade que não pode ser desconsiderada. Tais tópicos de privacidade são sérios quando a informação é correta, mas se tornam gigantescos quando é equivocada. Imagine o sentimento de desespero que você teria ao constatar que seu crédito foi afetado negativamente por informação equivocada. Imagine como os seus problemas aumentarão em um ambiente no qual essa desinformação seja prontamente compartilhada com outras instituições.

Os problemas de privacidade são e serão o principal efeito colateral do avanço da tecnologia em geral e das técnicas de banco de dados em particular. As soluções para tais problemas exigirão uma cidadania educada, alerta e ativa.



QUESTÕES/EXERCÍCIOS

1. As agências legais podem ter acesso a bancos de dados com o propósito de identificar indivíduos com tendência criminais, mesmo que eles não tenham se envolvido em delito algum?
2. As corretores de seguro podem ter acesso a bancos de dados com o propósito de identificar indivíduos com problemas médicos em potencial, mesmo que eles não tenham apresentado qualquer sintoma?
3. Suponha que você esteja em situação financeira confortável. Que benefícios você teria com a divulgação de sua situação entre diversas instituições? Que penalidades sofreria com a distribuição dessa mesma informação? O que aconteceria se você se encontrasse em uma situação financeira difícil?
4. Que papel a imprensa livre desempenha no controle do abuso em bancos de dados? (Por exemplo, até que ponto a imprensa livre afeta a opinião pública?)

Problemas de revisão de capítulo

(Os problemas marcados com asterisco se referem às seções opcionais.)

1. Resuma a diferença entre um simples arquivo e um banco de dados.
2. Qual o significado de independência de dados?
3. Qual o papel de um SGBD na abordagem de implementação em níveis para bancos de dados?
4. Qual a diferença entre um esquema e um subesquema?
5. Identifique duas vantagens de fazer distinção entre software de aplicação e SGBD.
6. Identifique os níveis, em um sistema de banco de dados (usuário final, programador do software de aplicação, projetista do software de SGBD), nos quais ocorrem as seguintes situações ou atividades:
 - a. Como os dados deveriam ser armazenados em um disco para maximizar a eficiência?
 - b. Há lugar vago no vôo 243?
 - c. Uma relação poderia ser armazenada como um arquivo seqüencial?
 - d. Quantas vezes deveria ser permitido a um usuário entrar com a senha errada antes de interromper a conversação?
 - e. Como a operação PROJECT pode ser implementada?
7. Descreva como as seguintes informações sobre linhas aéreas, vôos (para um determinado dia) e

passageiros seriam representados em um banco de dados relacional?

Companhias aéreas: Clear Sky, Long Hop e Tree Top.
Vôos da Clear Sky: CS205, CS37 e CS102.

Vôos da Long Hop: LH67 e LH89.

Vôos da Tree Top: TT331 e TT809.

Smith tem reservas para os vôos CS205 (assento 12B), CS37 (assento 18C) e LH 89 (assento 14A).

Baker tem reservas para os vôos CS37 (assento 18B) e LH89 (assento 14B).

Clark tem reservas para os vôos LH67 (assento 5A) e TT331 (assento 4B).

8. Suponha que você deva construir um banco de dados relacional com informações sobre a escala de vôos de uma companhia aérea. Como você construiria a estrutura de armazenamento subjacente de modo que certos vôos fossem selecionados (operação SELECT) rapidamente?
9. Até que ponto é significativa a ordem em que as operações SELECT e PROJECT são aplicadas a uma relação? Isto é, sob que condições a seleção e projeção produzem o mesmo resultado que a projeção e a seleção?
10. Considerando-se as relações abaixo, como ficará a relação RESULTADO após executar estas instruções?

Relação X			Relação Y	
U	V	W	R	S
A	Z	5	3	J
B	D	3	4	K
C	Q	5		

- a. $\text{RESULTADO} \leftarrow \text{PROJECT W from X}$
- b. $\text{RESULTADO} \leftarrow \text{SELECT from X where } W = 5$
- c. $\text{RESULTADO} \leftarrow \text{PROJECT S from Y}$
- d. $\text{RESULTADO} \leftarrow \text{JOIN X and Y where } X.W \$ Y.R$
11. Utilizando os comandos SELECT, PROJECT e JOIN, escreva uma sequência de instruções para responder às seguintes consultas sobre peças e seus fabricantes, em termos do seguinte banco de dados:
 - a. Quais companhias produzem o Parafuso 2Z?
 - b. Obtenha uma lista dos produtos feitos pelo Fabricante X ao lado do custo de cada peça.

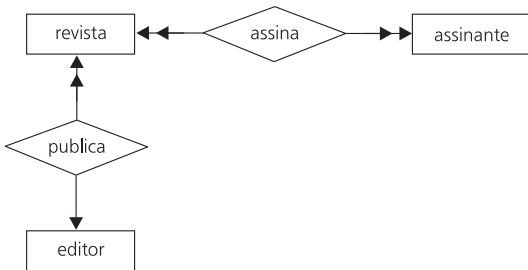
Relação PEÇA	
NomeDaPeça	Peso
Parafuso 2X	1
Parafuso 2Z	1,5
Porca V5	0,5

Relação FABRICANTE		
NomeDoFabricante	NomeDaPeça	Custo
Fabricante X	Parafuso 2Z	0,03
Fabricante X	Porca V5	0,01
Fabricante Y	Parafuso 2X	0,02
Fabricante Y	Porca V5	0,01
Fabricante Y	Parafuso 2Z	0,04
Fabricante Z	Porca V5	0,01

- c. Quais Fabricantes fazem uma peça com peso 1?
12. Refaça o Problema 11 usando SQL.
13. Qual redundância será introduzida se as informações contidas nas relações PEÇA e FABRICANTE do Problema 11 forem combinadas em uma única relação?
14. Utilizando comandos como SELECT, PROJECT e JOIN, escreva seqüências para responder às seguintes consultas sobre a informação das relações FUNCIONARIO, CARGO e ATRIBUICAO da Figura 9.5:
 - a. Obtenha uma lista dos nomes e endereços dos funcionários da companhia.
 - b. Obtenha uma lista dos nomes e endereços dos que trabalharam ou estão trabalhando no departamento de pessoal.
 - c. Obtenha uma lista dos nomes e endereços dos que estão trabalhando no departamento de pessoal.
15. Responda ao Problema 14 usando SQL.
16. Projete um banco de dados relacional com informações sobre compositores musicais, suas vidas e suas composições. (Evite redundâncias similares às da Figura 9.4.)
17. Projete um banco de dados relacional com informações sobre instrumentistas, suas gravações e os compositores das músicas por eles gravadas. (Evite redundâncias similares às da Figura 9.4.)
18. Projete um banco de dados relacional com informações sobre fabricantes de equipamentos

de computação e seus produtos. (Evite redundâncias similares às da Figura 9.4.)

- 19.** Projete um banco de dados relacional com informações sobre editores, revistas e assinantes, sendo que as relações entre tais entidades estão representadas pelo seguinte diagrama de entidade-relacionamento. (Evite redundâncias similares às da Figura 9.4.)



- 20.** Projete um banco de dados relacional com informações sobre peças, fornecedores e clientes. Cada peça é produzida por vários fornecedores e solicitada por muitos clientes. Cada fornecedor produz muitas peças e possui muitos clientes. Cada cliente pede muitas peças de muitos fornecedores e uma mesma peça até mesmo é solicitada de mais de um fornecedor. (Evite redundâncias similares às da Figura 9.4.)
- 21.** Quais mudanças são necessárias no projeto do banco de dados relacional mostrado na Figura 9.5 para se obter uma lista de todos os funcionários com sobrenome Smith, utilizando somente a operação SELECT?
- 22.** Que inconveniência pode ocorrer por causa da nossa decisão de combinar dia, mês e ano em um único atributo no banco de dados relacional da Figura 9.5?
- 23.** Escreva uma seqüência de instruções (usando as operações SELECT, PROJECT e JOIN) para obter o IdCargo, DataEntrada e DataSaida para cada cargo do departamento de contabilidade do banco de dados relacional descrito na Figura 9.6.
- 24.** Responda ao Problema 23 usando SQL.
- 25.** Escreva uma seqüência de instruções (usando as operações SELECT, PROJECT e JOIN) para obter o Nome, Endereco, NomeDoCargo e Departamento de todos os funcionários atuais do banco de dados relacional descrito na Figura 9.5.

- 26.** Responda ao Problema 25 usando SQL.

- 27.** Escreva uma seqüência de instruções (usando as operações SELECT, PROJECT e JOIN) para obter o Nome e NomeDoCargo de cada funcionário que conste no banco de dados relacional descrito na Figura 9.5.
- 28.** Responda ao Problema 27 usando SQL.
- 29.** Qual a diferença entre a informação fornecida pela única relação

Nome	Departamento	NumeroDoTelefone
Jones	Vendas	111-2222
Smith	Vendas	111-3333
Baker	Pessoal	111-4444

e as duas relações

Nome	Departamento
Jones	Vendas
Smith	Vendas
Baker	Pessoal

- 30.** Projete um banco de dados relacional com informações sobre peças de automóvel e suas partes. Esteja atento ao fato de uma peça poder conter peças menores e, ao mesmo tempo, fazer parte de peças ainda maiores.
- 31.** Com base no banco de dados representado na Figura 9.5, expresse a consulta correspondente ao seguinte trecho de programa:
- ```

TEMP ← SELECT from ATRIBUICAO where
 DataSaida = “*”
RESULTADO ← PROJECT IdCargo DataEntrada
 from TEMP

```
- 32.** Traduza para SQL a consulta do Problema 31.
- 33.** Com base no banco de dados representado na Figura 9.5, expresse a consulta correspondente ao seguinte trecho de programa:
- ```

TEMP1 ← JOIN FUNCIONARIO and ATRIBUICAO
  where FUNCIONARIO.IdFuncionario =
    ATRIBUICAO.Id.Funcionario
TEMP2 ← SELECT from TEMP1 where
  DataSaida = “*”
RESULTADO ← PROJECT Nome, DataEntrada
  from TEMP2
  
```
- 34.** Traduza para SQL a consulta do Problema 33.

- 35.** Com base no banco de dados representado na Figura 9.5, expresse a consulta correspondente ao seguinte trecho de programa:

```
TEMP1 ← JOIN FUNCIONARIO and CARGO
    where
        FUNCIONARIO.IdFuncionario = CARGO.
            IdFuncionario
TEMP2 ← SELECT from TEMP1 where
    Departamento = "VENDAS"
RESULTADO ← PROJECT Nome from TEMP2
```

- 36.** Traduza para SQL a consulta do Problema 35.
- 37.** Traduza o comando SQL abaixo

```
select CARGO.NomeDoCargo
from ATRIBUICAO, CARGO
where ATRIBUICAO.IdCargo =
CARGO.IdCargo and
    ATRIBUICAO.IdFuncionario
= "34Y70"
```

para a forma de uma seqüência de operações
SELECT, PROJECT e JOIN.

- 38.** Traduza o comando SQL abaixo:

```
select ATRIBUICAO.DataEntrada
from ATRIBUICAO, FUNCIONARIO
where ATRIBUICAO.IdFuncionario =
funcionario.IdFuncionario
and FUNCIONARIO.Nome = "Joe E. Baker"
```

para a forma de uma seqüência de operações
SELECT, PROJECT e JOIN.

- 39.** Descreva o resultado do seguinte comando SQL, usando o banco de dados do Problema 11.

```
insert into FABRICANTE
values ('Fabricante Z', 'Parafuso
2X', 0.03)
```

- 40.** Descreva o resultado do seguinte comando SQL, usando o banco de dados do Problema 11.

```
update FABRICANTE
set Custo = 0,03
where NomeDoFabricante =
    'Fabricante Y' and NomeDaPeça =
    'Parafuso 2X'
```

- 41.** Por que seria vantajoso armazenar uma relação em um arquivo indexado em vez de em um simples arquivo seqüencial?

- *42.** Identifique alguns dos objetos que você esperaria encontrar em um banco de dados orientado a

objeto, utilizado para manter um estoque de um supermercado. Que métodos você espera encontrar em cada objeto?

- *43.** Identifique alguns dos objetos que você esperaria encontrar em um banco de dados orientado a objeto para manter registros do acervo de uma biblioteca. Que métodos você esperaria encontrar em cada objeto?
- *44.** Compare a relação existente entre uma classe e um objeto (em um ambiente orientado a objeto) com a relação existente entre um esquema de banco de dados e o próprio banco de dados.
- *45.** Que informação incorreta é gerada pela seqüência de transações T1 e T2? T1 foi projetada para calcular a soma da contas A e B, e T2 para transferir \$100 da conta A para a conta B. T1 começa consultando o saldo da conta A; em seguida, T2 executa a sua transferência, e, finalmente, T1 obtém o saldo da conta B e relata a soma dos valores por ela obtidos.
- *46.** Explique de que maneira o protocolo de trancamento descrito no texto solucionaria o erro produzido no Problema 45.
- *47.** Que efeito o protocolo de *wound-wait* teria sobre a seqüência de eventos do Problema 45 se T1 fosse a transação mais recente? E se essa transação fosse T2?
- *48.** Suponha que uma transação tente somar \$100 a uma conta cujo saldo é \$200, enquanto outra tenta retirar \$100 da mesma conta. Descreva uma alternância dessas transações que conduza a um saldo final de \$100. Descreva uma outra alternância, que conduza a um saldo final de \$300.
- *49.** Qual é a diferença entre uma transação que tenha acesso exclusivo a um item de um banco de dados e outra que tenha acesso compartilhado ao mesmo item? Por que essa distinção é importante?
- *50.** Os problemas discutidos na Seção 9.5 não se limitam aos ambientes de banco de dados. Que problemas similares aparecem quando se tem acesso a um documento usando um processador de texto? (Se você possui um PC com um processador de texto, tente acessar o mesmo documento a partir de duas ativações do processador de texto e veja o que acontece.)

Questões sociais

As seguintes questões procuram auxiliar o leitor no entendimento de alguns assuntos éticos, sociais e legais no campo da computação. O objetivo não é meramente o de fornecer respostas a tais questões. O leitor também deve justificá-las e verificar se as justificativas apresentadas preservam sua consistência de uma questão para outra.

1. Nos Estados Unidos, os registros de DNA de todos os prisioneiros federais são agora armazenados em um banco de dados para uso em investigações criminais. Seria ético liberar essa informação para outras finalidades — por exemplo, para pesquisa genética? Se afirmativo, para que finalidades? Senão, por que não? Quais são os prós e contras de cada caso?
2. Até que ponto uma universidade pode divulgar as informações a respeito de seus estudantes? Que tal seus nomes e endereços? E as distribuições de notas sem identificar os estudantes? A sua resposta é coerente com a da Questão 1?
3. Que restrições são apropriadas quando se trata da construção de bancos de dados sobre indivíduos? Que tipo de informação um governo tem direito de manter em relação aos seus cidadãos? Que tipo de informação uma companhia de seguros tem direito de manter sobre os seus clientes? Que tipo de informação uma empresa pode manter em relação a seus funcionários? Deveriam ser implementados controles sobre tais situações? Se a resposta for afirmativa, como fazê-lo?
4. É adequado a uma empresa de cartão de crédito comercializar o perfil de consumo de seus clientes para empresas de propaganda? É aceitável que uma empresa, que aceita por correspondência encomendas de carros-esporte, venda a lista de nomes e endereços de seus clientes para uma revista de carros-esporte? É aceitável que um órgão de recolhimento de Imposto de Consumo venda para corretores de títulos uma lista de nomes e endereços dos contribuintes que apresentem altos ganhos de capital? (Se a sua resposta não for um enfático sim ou não, o que você proporia como política aceitável?)
5. Até que ponto o projetista de um banco de dados é responsável pela maneira como a informação nele contida é usada?
6. Suponha que, de maneira equivocada, um banco de dados permita acessos não-autorizados às informações que possui. Se estas informações forem indevidamente obtidas e utilizadas, até que ponto os projetistas do banco de dados serão responsáveis por isso? Essa resposta depende do grau de esforço necessário para que o criminoso localize no projeto de banco de dados alguma brecha que lhe permita acesso à informação confidencial em questão?
7. O predomínio da mineração de dados levanta numerosas questões de ética e de privacidade. A sua privacidade será invadida se a mineração de dados revelar certas características gerais de sua comunidade? O uso da mineração de dados promove práticas comerciais salutares ou fanatismo? Até que ponto é adequado forçar os cidadãos a participar de censos, levando-se em conta que será extraída mais informação dos dados do que a explicitamente solicitada nos questionários individuais? A mineração de dados dá às agências de propaganda uma vantagem injusta em relação às entrevistas infensivas? Até que ponto o estabelecimento de perfis é bom ou mau?
8. Muitas bibliotecas oferecem um serviço pelo qual os usuários podem solicitar a assistência de um bibliotecário quando procuram uma certa informação. A existência da Internet e dos bancos de dados tornam esse serviço obsoleto? Se afirmativo, terá sido um passo para a frente ou para trás? Senão, por que não? Como a existência da Internet e da tecnologia de banco de dados afeta a existência das bibliotecas?
9. É sua responsabilidade trancar a sua casa, de modo que intrusos não entrem, ou é responsabilidade das pessoas permanecer fora de sua casa se não for convidado? É responsabilidade de uma organização construir bancos de dados invioláveis aos invasores ou é responsabilidade dos invasores deixar os bancos de dados em paz?

Leituras adicionais

Date, C. J. *An Introduction to Database Systems*, 7th ed. Boston: Addison-Wesley, 2002.

Han, J. and M. Kamber, *Data Mining: Concepts and Techniques*. San Francisco, CA: Morgan Kaufmann, 2001.

Ramakrishnan, R. *Database Management Systems*, 2nd ed. New York: McGraw-Hill, 2000.

Silberschatz, A., H. Karth, and S. Sudarshan *Database Systems Concepts*, 4th ed. New York: McGraw-Hill, 2002.

Ullman, J. D. and J. D. Widom. *A First Course in Database Systems*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2002.

P A R T E

4

O potencial das máquinas

Na Parte 4, consideraremos o potencial dos processos algorítmicos e das máquinas que os executam. Iniciaremos investigando, no Capítulo 10, o tópico relativo à inteligência artificial. Nele verificaremos os principais avanços realizados na produção de máquinas que imitam as ações humanas e, assim, projetam a imagem do comportamento inteligente. Esta tecnologia levanta a questão sobre quais são, se existirem, as limitações das máquinas.

Tal questão será analisada no Capítulo 11, em que estudaremos a Teoria da Computação. Nele, verificaremos a existência de limites para as tarefas realizadas por máquinas. Além disso, veremos que aspectos de ordem prática limitam ainda mais essas tarefas. Isto é, tarefas que, embora estejam dentro das capacidades teóricas dos computadores, gastam tanto tempo de execução que se tornam inviáveis na prática, apesar de todos os avanços tecnológicos.

CAPÍTULO 10

Inteligência artificial

Um dos principais objetivos dos cientistas da computação é o desenvolvimento de máquinas que interajam com os seus ambientes da maneira sensorial que tradicionalmente caracteriza os seres humanos e que desempenhem suas funções de forma inteligente, sem a necessidade da intervenção humana. A concretização deste objetivo freqüentemente exige que a máquina “entenda” ou perceba o estímulo recebido e seja capaz de tirar conclusões mediante alguma forma de processo de raciocínio.

Tanto a percepção quanto o raciocínio se enquadram na categoria das atividades corriqueiras que, embora naturais para a mente humana, representam grandes dificuldades para as máquinas. O resultado disto tudo é que a área de pesquisa relacionada com esse objetivo, a inteligência artificial, está em uma fase muito primitiva, em comparação com seus alvos e expectativas.

10.1 Inteligência e máquinas

Desempenho *versus* simulação
O teste de Turing
A máquina do quebra-cabeças de oito peças

10.2 Compreensão das imagens

10.3 Raciocínio

Sistemas de produções
Árvores de busca
Heurística

10.4 Redes neurais artificiais

Propriedades básicas
Uma aplicação específica
Memória associativa

10.5 Algoritmos genéticos

10.6 Outras áreas de pesquisa

Processamento de linguagens
Robótica
Sistemas de banco de dados
Sistemas especialistas

10.7 Considerando as consequências

10.1 Inteligência e máquinas

Embora o computador muitas vezes seja personificado, há uma importante distinção entre suas propriedades e as da mente humana. Os computadores são capazes de executar tarefas precisamente definidas, com velocidade e exatidão. Todavia, essas máquinas são desprovidas de bom senso. Quando estão diante de uma situação não prevista pelo programador, o seu desempenho cai sensivelmente. Embora a mente humana em geral tenha dificuldades com cálculos complexos, ela é capaz de compreender e de raciocinar. Por conseguinte, embora a máquina consiga ultrapassar o ser humano no cálculo das soluções de problemas de física nuclear, o ser humano tem mais chances de compreender os resultados e determinar qual deve ser o próximo cálculo a ser realizado.

Desempenho versus simulação

Se desejarmos construir máquinas capazes de realizar tarefas complexas sem intervenção humana, elas terão de se tornar mais parecidas com o ser humano, isto é, deverão possuir (ou pelo menos simular) a capacidade de raciocinar. Conscientes desta necessidade, os cientistas da computação dirigiram suas atenções para outras ciências, como a psicologia e a biologia, na esperança de descobrir os princípios que poderiam ser aplicados à construção de máquinas e programas mais flexíveis. O resultado indicou uma constante dificuldade de distinção entre as pesquisas nessas outras áreas e as de um cientista da computação. A diferença não está no que eles fazem, mas, sobretudo, nos seus objetivos. Um psicólogo, por exemplo, procura aprender mais sobre a mente humana e seus processos de pensamento; um cientista da computação busca a construção de máquinas mais úteis.

Para esclarecer, suponhamos que um cientista da computação e um psicólogo, cada qual trabalhando de modo independente, se envolvam em projetos para desenvolver um programa para jogar pôquer. O cientista da computação pode aplicar os fundamentos da probabilidade e estatística. O resultado seria um programa que jogaria de forma extravagante, blefando aleatoriamente, sem demonstrar emoções, maximizando, por conseguinte, suas probabilidades de vencer. Por outro lado,

o psicólogo desenvolveria um programa baseado nas teorias do pensamento e do comportamento humano. O projeto poderia resultar até mesmo na produção de vários programas diferentes: um deles jogaria agressivamente, enquanto outro seria facilmente intimidado. Em contraste ao programa do cientista da computação, o programa do psicólogo poderia ficar “emocionalmente envolvido” no jogo e, assim, perder tudo o que havia ganho.

Hipoteticamente falando, a preocupação principal do cientista da computação, ao desenvolver o programa, seria relativa ao desempenho final do mesmo. Tal abordagem é denominada **orientada ao desempenho**. Em contraste, o psicólogo estaria mais interessado em entender os processos da inteligência natural, e assim tratar o projeto como uma oportunidade para testar teorias, construindo modelos computacionais baseados nelas. Sob este ponto de vista, o desenvolvimento do programa “inteligente” é, na verdade, um efeito colateral de um outro objetivo — o progresso no conhecimento do pensamento e do comportamento humano. Esta abordagem é chamada **orientada à simulação**.

Outro exemplo que distingue as metas da abordagem orientada ao desempenho e da abordagem orienta-

As origens da inteligência artificial

O desejo de construir máquinas que imitem o comportamento humano possui uma longa história, mas muitos concordam em que o campo moderno da inteligência artificial teve suas origens em 1950, com a publicação do *Computing Machinery and Intelligence*, de Alan Turing. Foi aí que Turing propôs que as máquinas poderiam ser programadas para exibir um comportamento inteligente. O nome do campo — *inteligência artificial* — foi dado alguns anos mais tarde na agora lendária proposta escrita por John McCarthy, a qual sugeria que “um estudo da inteligência artificial seja feito durante o verão de 1956 no Dartmouth College” para explorar “a conjectura segundo a qual qualquer aspecto de aprendizado ou qualquer outra característica da inteligência pode, em princípio, ser precisamente descrito de forma que uma máquina possa ser feita para o simular.”

da à simulação é encontrado em áreas de processamento de linguagens naturais e na lingüística. Esses campos são intimamente relacionados e se beneficiam mutuamente das pesquisas, embora as metas subjacentes sejam diferentes. Os lingüistas se interessam pelo modo como as pessoas processam a linguagem, enquanto os pesquisadores no campo do processamento de linguagens naturais se interessam pelo desenvolvimento de máquinas que possam “entender” uma linguagem natural. Os sistemas de processamento de linguagens naturais, como os tradutores de documentos e os sistemas pelos quais as máquinas respondem a comandos verbais, se apóiam fortemente no conhecimento obtido pelos lingüistas, mas freqüentemente aplicam “atalhos” que funcionam bem nos ambientes restritos onde o sistema está sendo projetado. Por exemplo, uma casca de sistema operacional não precisa se preocupar com os vários significados da palavra *copiar* (É um substantivo ou um verbo? Deve implicar conotação de plágio?), mas simplesmente distingui-la de outras palavras, como *renomear* ou *apagar*. Assim, um sistema de processamento de linguagem natural pode desempenhar a sua tarefa corretamente, embora não seja esteticamente agradável a um lingüista.

O teste de Turing

As duas abordagens são sólidas e contribuem significativamente para o campo da inteligência artificial. Contudo, também levantam questões filosóficas evasivas dentro da disciplina. Por exemplo, consideremos a discussão que surgiria se fosse solicitado a um grupo decidir se os programas de jogo de pôquer desenvolvidos pelo cientista da computação e pelo psicólogo possuem de fato inteligência e, nesse caso, eleger o programa mais inteligente. Medimos a inteligência pela capacidade de ganhar ou de se assemelhar mais ao ser humano?

A segunda opção foi adotada por Alan Turing em 1950, quando propôs um teste (agora conhecido como o **teste de Turing**) para avaliar o comportamento inteligente em uma máquina. A proposta de Turing era permitir que um ser humano, a quem chamamos interrogador, se comunique com um objeto de teste, por meio de uma máquina de escrever, sem saber se o objeto é uma pessoa ou uma máquina. Nesse ambiente, declarar-se-ia que a máquina tem comportamento inteligente naquela situação se o interrogador não conseguisse distingui-la de um ser humano. Assim, o teste de Turing mede a capacidade da máquina de se assemelhar ao ser humano. Turing previu que, por volta do ano 2000, as máquinas teriam 30% de chance de passar em um teste de cinco minutos — conjectura que se revelou surpreendentemente correta.

Um exemplo famoso de um cenário do teste de Turing surgiu como resultado do programa DOCTOR (uma versão do sistema mais geral chamado ELIZA), desenvolvido por Joseph Weizenbaum em meados da década de 1960. Este programa interativo foi projetado para representar a imagem de um analista rogeriano, que conduzisse uma entrevista psicológica; o computador representou o papel do analista, enquanto o usuário, o do paciente. Internamente, tudo o que o DOCTOR fazia era reestruturar, de acordo com algumas regras bem definidas, as declarações feitas pelo paciente, devolvendo-as então à tela do computador. Por exemplo, em resposta a uma declaração como “Estou cansado hoje”, o DOCTOR poderia responder com “Por que você acha que está cansado hoje?” Se o DOCTOR não fosse capaz de reconhecer a estrutura da oração, iria apenas responder com algo como “Continue” ou “Isso é muito interessante”.

O propósito de Weizenbaum com o DOCTOR era o de estudar a comunicação em linguagem natural. Sob este ângulo, o assunto da psicoterapia representou um papel secundário, o de criar um ambiente (ou uma área de discussão) no qual o programa pudesse operar. Contudo, contrariando Weizenbaum, vários psicólogos propuseram realmente utilizar o programa para psicoterapia. (A tese rogeriana é a de que o paciente, e não o analista, deveria conduzir o diálogo durante a terapia, e eles argumentaram que, por essa razão, um computador provavelmente poderia conduzir uma discussão tão bem quanto um terapeuta.) Além disso, o programa DOCTOR representou tão fortemente a imagem da compreensão que muitos dos que se “comunicaram” com ele se descobriram relatando pensamentos e sentimentos íntimos e, em muitos casos, se tornaram realmente subservientes ao diálogo pergunta-resposta da máquina. Em um certo sentido, o DOCTOR passou no teste de Turing. O resultado foi o questionamento de tópicos éticos e técnicos.

Se uma máquina passa no teste de Turing, ela possui inteligência? Em última instância, a principal dificuldade em determinar se uma máquina possui inteligência está ligada à dificuldade de distinguir entre a mera aparência de inteligência e a sua existência de fato. A inteligência é uma característica interior, cuja presença é externamente detectada apenas de modo indireto, com base no contexto do diálogo estímulo/reação. Contudo, reações inteligentes implicam a existência da inteligência?

A máquina do quebra-cabeças de oito peças

Com tais questões filosóficas nos alicerces da inteligência artificial, não surpreende que muitos dos seus tópicos sejam acompanhados de uma aura de mistério, explorada pela mídia, pelos escritores de ficção e pelos produtores de cinema. Felizmente, nossa tarefa não é resolver essas questões mas investigar como as máquinas podem ser programadas para *parecer* inteligentes. Abordamos essa tarefa considerando o projeto de uma máquina que possui características inteligentes elementares.

Nossa máquina apresenta a forma de uma caixa de metal, equipada com uma pinça, uma câmera de vídeo e um dedo equipado com uma extremidade de borracha para que não deslize quando estiver empurrando alguma coisa (Figura 10.1). Imagine tal máquina colocada sobre uma mesa na qual está um quebra-cabeças de oito peças. É um quebra-cabeças composto de oito peças quadradas, numeradas de 1 a 8 e montadas sobre um tabuleiro com capacidade para armazenar um total de nove peças, em um arranjo de três linhas por três colunas. Entre as peças do tabuleiro, há um espaço vazio, para o qual qualquer peça adjacente pode ser empurrada. As peças estão, no momento, organizadas conforme a Figura 10.2.

Para começar, apanhamos o quebra-cabeças e o reorganizamos, empurrando sucessivamente, para o espaço vago, peças escolhidas aleatoriamente. Em seguida, ligamos a máquina, e a pinça fica abrindo e fechando, como que a pedir o quebra-cabeças. Colocamos o quebra-cabeças na pinça, e esta se fecha para segurá-lo. Depois de algum tempo, o dedo se abaixa e começa a empurrar as peças pelo tabuleiro (de uma forma ordenada), até que voltem à posição original. Nesse momento, a máquina coloca o quebra-cabeças de volta sobre a mesa e se autodesliga. Como tal máquina possui percepção elementar, bem como capacidades dedutivas, o seu projeto provê a base necessária para a apresentação dos tópicos das duas próximas seções.

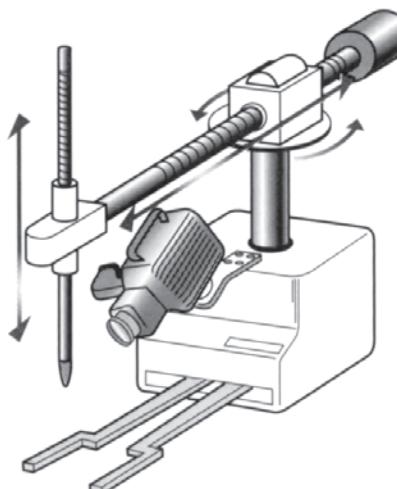


Figura 10.1 Nossa máquina de resolução de quebra-cabeças.

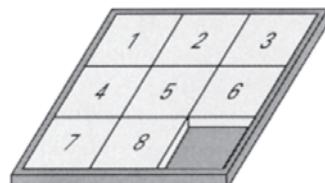


Figura 10.2 A configuração do quebra-cabeças de oito peças resolvido.



QUESTÕES/EXERCÍCIOS

1. Uma planta colocada em um quarto escuro com uma única fonte de luz cresce em direção à luz. Esta resposta da planta é inteligente? A planta possui inteligência? Qual é, então, a sua definição de inteligência?
2. Suponha que uma máquina de venda automática de produtos tenha sido projetada para liberar diversos produtos, dependendo do botão acionado. Você diria que a máquina está “ciente” de qual botão está sendo acionado? Qual é, então, a sua definição de ciente?
3. Se uma máquina passasse no teste de Turing, você concordaria que ela é inteligente? Se não, você concordaria que ela aparenta ser inteligente?

10.2 Compreensão das imagens

A abertura e o fechamento da pinça em nossa máquina não apresentam qualquer problema sério, e a capacidade de detectar a presença do quebra-cabeças na pinça durante este processo é direta, pois a nossa aplicação requer muito pouca precisão. (Abridores automáticos de porta de garagem são capazes de detectar e reagir à presença de um obstáculo na entrada quando estiverem se fechando.) Até mesmo o problema de focalizar a câmera sobre o quebra-cabeças pode ser tratado de modo simples, bastando mover o braço de modo que o quebra-cabeças seja posicionado em uma posição predeterminada. Por conseguinte, o primeiro comportamento inteligente requerido por nossa máquina é a extração de informação através de meios visuais.

É importante perceber que o problema a ser solucionado pela nossa máquina ao observar o quebra-cabeças não é simplesmente produzir e armazenar uma imagem. Há muitos anos, a tecnologia tem conseguido fazer isso, como é o caso da fotografia tradicional e dos sistemas de televisão. Pelo contrário, o problema é interpretar a imagem para dela extrair o estado corrente do quebra-cabeças (e depois monitorar o movimento das peças). Esta é uma diferença significativa em relação à atividade de um receptor de televisão, o qual simplesmente converte a imagem de um meio para outro, sem o entendimento conceitual da mesma. Em suma, a nossa máquina deve exibir capacidade de percepção.

No caso da nossa máquina de resolução de quebra-cabeças, as opções são relativamente limitadas quanto ao que as imagens poderiam ser. Podemos pressupor que o que aparece é sempre uma imagem do quebra-cabeças, com os dígitos 1 a 8 em um padrão bem organizado. O problema se resume simplesmente em extraír o arranjo destes dígitos. Para isto, imaginamos que o tabuleiro do quebra-cabeças tenha sido codificado em *bits*, na memória do computador, e que cada *bit* represente o nível de brilho de um elemento da figura, chamado *pixel*. Supondo que a imagem apresente um tamanho uniforme (a máquina segura o quebra-cabeças à sua frente, em uma posição predeterminada em relação à câmera), podemos descobrir a posição de cada peça comparando as diferentes partes da figura com os modelos pré-armazenados, os quais consistem nos padrões de *bits*, independentemente produzidos pelos diversos dígitos utilizados no quebra-cabeças. Assim que as coincidências são encontradas, a condição do quebra-cabeças fica determinada.

Esta técnica de reconhecimento de imagens é um método utilizado em leitores de caracteres óticos. Entretanto, possui a desvantagem de exigir uma certa uniformidade entre o estilo, o tamanho e a orientação dos símbolos na leitura. Em particular, o padrão de *bits* produzido por um caractere fisicamente grande não é compatível com o modelo de uma versão menor do mesmo símbolo, apesar de as formas serem as mesmas. Além disso, você pode imaginar como tais problemas aumentariam de proporção no caso do processamento de material manuscrito.

Outra abordagem utilizada para solucionar problemas de reconhecimento de caracteres está baseada na identificação das características geométricas, e não na aparência exata dos símbolos. Nestes casos, o dígito 1 seria caracterizado como uma única linha vertical, 2 seria uma linha curva aberta ligada ao final a uma linha horizontal, e assim por diante. Esse método de reconhecimento de símbolos envolve dois passos — o primeiro é extraír as características da imagem que está sendo processada, o segundo é

compará-las com as dos símbolos conhecidos. Como acontece com a abordagem da coincidência de padrões, essa técnica de reconhecimento de caracteres não é perfeitamente segura, pois erros secundários na imagem podem produzir um conjunto de características geométricas completamente diferentes, como é o caso de distinguir entre um O e um C ou, no caso do quebra-cabeças de oito peças, diferenciar um 3 de um 8.

Tivemos sorte na nossa aplicação do quebra-cabeças, pois não foi preciso reconhecer e interpretar imagens de cenas genéricas tridimensionais. Considere, por exemplo, a grande vantagem de estarmos seguros de que as formas a serem reconhecidas (os dígitos de 1 a 8) estejam isoladas em diferentes regiões da figura, sem aparecer como imagens sobrepostas, o que é comum em situações mais gerais. Em uma fotografia genérica, por exemplo, há não apenas o problema do reconhecimento de um objeto em diferentes ângulos, mas também o fato de que algumas partes do mesmo podem estar invisíveis.

A tarefa de entender imagens genéricas geralmente é abordada como um processo de dois passos: (1) **processamento da imagem**, que se refere à identificação de características da imagem e (2) **análise da imagem**, relacionada ao processo de entender o quê essas características significam. Já percebemos essa dicotomia no contexto de reconhecimento de símbolos por meio de suas características geométricas. Lá, vimos que o processamento da imagem é representado pelo processo de identificar as características geométricas encontradas na imagem e a análise da imagem, pelo processo de identificar o significado dessas características.

O processamento da imagem engloba numerosos tópicos. Um deles é a melhoria das bordas, que é o processo de aplicar técnicas matemáticas para esclarecer as fronteiras entre regiões em uma imagem. De certo modo, a melhoria das bordas é uma tentativa de converter uma fotografia em um desenho de linhas. Outra atividade na análise da imagem é conhecida como determinação de regiões. Esse é o processo de identificar as áreas na imagem que possuem propriedades em comum, tais como brilho, cor e textura. Tal região provavelmente representa uma seção da imagem pertencente a um único objeto. Essa habilidade de reconhecer regiões permite aos computadores colorir desenhos ou antigos filmes em preto e branco. Uma outra atividade no escopo do processamento de imagens é suavizar, que é o processo de remover falhas na imagem. O ato de suavizar evita que erros na imagem confundam os outros passos no seu processamento, mas, em excesso, pode causar perda da informação importante.

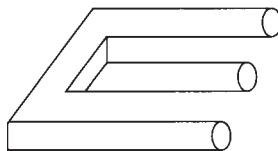
O ato de suavizar, a melhoria das bordas e a determinação de regiões são passos no sentido de identificar os vários componentes de uma imagem, cuja análise é o processo de determinar o quê esses componentes representam e, em última instância, o que a imagem significa. Aqui nos defrontamos com problemas como o de reconhecimento de objetos parcialmente obstruídos a partir de perspectivas diferentes. Uma abordagem à análise de imagens é começar com uma suposição de o quê a imagem dever ser e então tentar associar seus componentes com os objetos cuja presença foi conjecturada. Essa parece ser a abordagem adotada pelas pessoas. Por exemplo, algumas vezes achamos difícil identificar um objeto inesperado em uma situação onde nossa visão está embacada, mas uma vez que tenhamos uma indicação de o quê o objeto deve ser, podemos reconhecê-lo facilmente.

Os problemas associados à análise de imagens em geral são enormes, e muita pesquisa na área ainda deve ser feita. Tarefas que são realizadas com rapidez e aparente facilidade pela mente humana continuam além das capacidades das máquinas. Entretanto, existem indicações de que as máquinas com arquitetura alternativa podem algum dia resolver os problemas que nos escapam atualmente (veja a Seção 10.4).

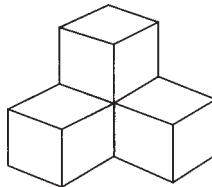


QUESTÕES/EXERCÍCIOS

1. Como os requisitos de um sistema de vídeo diferem para um robô se as imagens forem utilizadas pelo próprio robô para controlar suas atividades, em vez de enviadas para um ser humano que comande o robô por controle remoto?
2. No desenho seguinte, o que indica que ele é absurdo? Como tal percepção pode ser programada em uma máquina?



3. Quantos blocos existem no conjunto abaixo? Como uma máquina poderia ser programada para responder tais questões corretamente?



10.3 Raciocínio*

Uma vez que a nossa máquina de resolução de quebra-cabeças já decifrou as posições das peças a partir da imagem visual, sua tarefa passa a ser a determinação dos movimentos necessários para resolver o quebra-cabeças. Uma abordagem a esse problema que pode vir à mente é pré-programar a máquina com as soluções para todos os arranjos possíveis das peças. Então, a tarefa da máquina se restringe apenas a selecionar e executar o programa adequado. Entretanto, o quebra-cabeças de oito peças possui um total de 181.440 configurações diferentes, e a idéia de apresentar uma solução explícita para cada uma certamente não é convidativa e, provavelmente, nem mesmo possível, considerados os fatores tempo e área de armazenamento.

Portanto, somos forçados a programar a máquina de forma que possa construir por si própria as soluções do quebra-cabeças. Isto é, a máquina deve ser programada para tomar decisões, tirar conclusões e, em suma, executar ações racionais elementares.

Sistemas de produções

O desenvolvimento de capacidades de raciocínio em uma máquina é um tópico de pesquisa há muitos anos. Um dos resultados dessa pesquisa é o reconhecimento de que uma grande classe de problemas relacionados ao raciocínio possui características em comum. Elas são isoladas em um sistema conhecido como **sistema de produções**, que consiste em três componentes principais:

1. *Um conjunto de estados.* Cada estado é uma situação que pode ocorrer no ambiente da aplicação. O estado a partir do qual o ambiente é iniciado é chamado de **estado inicial**; o(s) estado(s) desejado(s) é(são) chamado(s) **estado(s)-meta**. (No nosso caso, um estado é a configuração do quebra-cabeças; o estado inicial, a configuração do quebra-cabeças ao ser fornecido à máquina; e o estado-meta, é a configuração do quebra-cabeças resolvido, conforme a Figura 10.2.)
2. *Um conjunto de produções* (regras ou movimentos). Uma **produção** é uma operação que pode ser executada no ambiente da aplicação para migrar de um estado para outro. Cada produção pode estar associada a precondições. Em outras palavras, as precondições são as condições que devem obrigatoriamente estar presentes no ambiente para que a produção possa ser aplicada. (No nosso caso, as produções representam a movimentação das peças. Cada movimento de uma peça possui como precondição o fato de a posição vazia estar vizinha à peça em questão.)

*N. de T. Em inglês, *reasoning*.

3. *Um sistema de controle.* O **sistema de controle** consiste na lógica necessária para resolver o problema de conduzir o quebra-cabeças do estado inicial para o estado-meta. A cada passo do processo, o sistema de controle deve decidir qual das produções, cujas precondições estejam satisfeitas, deve ser aplicada a seguir. (Dado um estado particular no nosso exemplo de quebra-cabeças de oito peças, pode haver várias peças vizinhas à posição vazia e, portanto, várias produções são aplicáveis. O sistema de controle deve decidir qual dessas peças deverá ser movida.)

Note que a tarefa atribuída à nossa máquina de resolver o quebra-cabeças pode ser formulada no contexto de um sistema de produções. Ái, o sistema de controle toma a forma de um programa, o qual inspecciona o estado corrente do quebra-cabeças, identifica uma seqüência de produções que conduz ao estado-meta e a executa. É, portanto, nossa tarefa projetar um sistema de controle para resolver o quebra-cabeças de oito peças.

Um conceito importante no desenvolvimento de um sistema de controle é o de **grafo de estados**, que é uma forma adequada para representar ou, pelo menos, conceituar todos os estados, produções e precondições em um sistema de produções. Aqui, o termo *grafo* refere-se a uma estrutura que os matemáticos chamam **grafo dirigido**, que significa um conjunto de posições, chamadas **nós**, conectadas por setas ou **arcos**. Um grafo de estados consiste em um conjunto de nós, que representam os estados do sistema, conectados por arcos, que representam as produções, as quais produzem as transições no sistema de um estado para outro. Dois nós podem ser conectados por um arco no grafo de estados se e somente se uma produção do sistema puder ser utilizada para levar o sistema do estado associado à origem do arco para o estado referente ao destino desse arco. As precondições são implicitamente representadas pela ausência de arcos entre certos nós.

Inteligência baseada no comportamento

Os primeiros trabalhos na inteligência artificial abordavam o assunto no contexto da escrita de programas para simular a inteligência. Entretanto, muitos argumentam atualmente que a inteligência humana não é baseada na execução de programas complexos. Em vez disso, eles afirmam que a inteligência humana é baseada na evolução do comportamento que garanta a sobrevivência, de modo que é mais bem simulada por funções simples de estímulo-resposta, como as simuladas por neurônios artificiais. Essa teoria explica porque as máquinas baseadas na arquitetura de von Neumann facilmente superam os seres humanos em capacidade de cálculo, mas só com grande esforço exibem bom senso. Assim, o conceito do comportamento evolutivo promete influenciar fortemente a pesquisa na inteligência artificial. Por exemplo, técnicas baseadas no comportamento têm sido aplicadas na área da robótica para produzir robôs simples que realizam tarefas elementares por meio do comportamento evolutivo, e não pela execução de programas complexos. Outro exemplo é a área de redes neurais artificiais na qual os sistemas são ensinados a se comportar de modo desejado, em vez de explicitamente programados para realizar computações predeterminadas.

Podemos aqui enfatizar que, da mesma forma que o número de estados possíveis nos impediou de apresentar explicitamente soluções pré-projetadas para o quebra-cabeças de oito peças, o problema de tamanho nos impede de representar explicitamente, na íntegra, o grafo de estados. Portanto, embora um grafo de estados seja uma forma de conceituar o problema em questão, ele não o expressa em sua totalidade. Todavia, ele se mostra útil para analisar (e possivelmente estender) a parte do grafo de estados referente ao quebra-cabeças de oito peças exibido na Figura 10.3.

Observemos que, em termos de grafos de estados, o problema enfrentado pelo sistema de controle passa a ser o de encontrar uma seqüência de arcos que conduza do estado inicial ao estado-meta, pois ela representa uma seqüência de produções que resolve o problema original. Assim, independentemente da aplicação, a tarefa do sistema de controle pode ser vista como encontrar um caminho em um grafo de estados. Essa visão genérica dos sistemas de controle é a recompensa que se obtém ao analisar problemas que exijam raciocínio em termos dos sistemas de produções. Se um problema é caracterizado em termos de um sistema de produções, então a sua solução é formulada em termos de procurar um caminho.

Para enfatizar esse ponto, consideremos como outras tarefas podem ser enquadradadas em termos de um

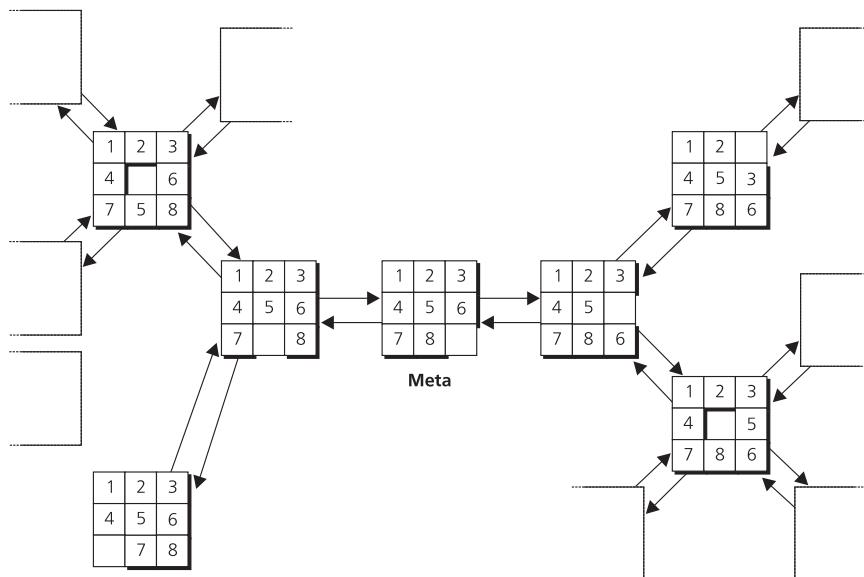


Figura 10.3 Uma pequena parte do grafo de estados do quebra-cabeças de oito peças.

sistema de produções e, assim, como sistemas de controle que encontram os caminhos nos grafos de estado. Um dos problemas clássicos da inteligência artificial é o da utilização do computador em jogos como o xadrez. Tais jogos envolvem uma complexidade moderada, em um contexto bem definido, e consequentemente apresentam um ambiente ideal para o teste de teorias. Em xadrez, os estados são as configurações possíveis do tabuleiro, as produções são os movimentos das peças e o sistema de controle é materializado nos jogadores (humanos ou não). O nó inicial do grafo de estados representa o tabuleiro com as peças nas suas posições iniciais. Ramificações a partir deste nó são arcos que conduzem às diversas configurações do tabuleiro obtidas após o primeiro movimento em um jogo; em seguida, ramificando a partir de cada nó, obtém-se as possíveis configurações atingíveis com o próximo movimento, e assim por diante. Com esta formulação, podemos imaginar um jogo de xadrez como sendo de dois jogadores, em que cada qual tenta encontrar uma trajetória, em um grande grafo de estados, para um nó-mota à sua escolha.

Talvez um exemplo menos óbvio de um sistema de produções seja o problema de tirar conclusões lógicas a partir de determinados fatos. As produções, neste contexto, são as regras lógicas que permitem a formulação de proposições a partir de outras já existentes. Por exemplo, as proposições “Todos os super-heróis são nobres” e “Super-homem é um super-herói” podem ser combinadas para produzir “Super homem é nobre”. De modo similar, “Batman e Robin são espertos” pode ser reescrita como “Nem Batman nem Robin são não-espertos”. Os estados em tal sistema são conjuntos de proposições que se sabe serem verdadeiras em determinados pontos do processo dedutivo. O estado inicial é o conjunto de proposições básicas (normalmente chamadas axiomas) a partir das quais são tiradas as conclusões, e um estado-mota é qualquer conjunto de proposições que contenha a conclusão proposta.

Como exemplo, a Figura 10.4 apresenta a parte de um grafo de estados que poderia ocorrer quando a conclusão “Sócrates é mortal” é tirada do conjunto de proposições “Sócrates é homem”, “Todo homem é humano” e “Todos os humanos são mortais”. Nesse grafo, vemos o corpo de conhecimento sendo transferido de um estado a outro, enquanto o processo de raciocínio aplica produções adequadas para gerar proposições adicionais.

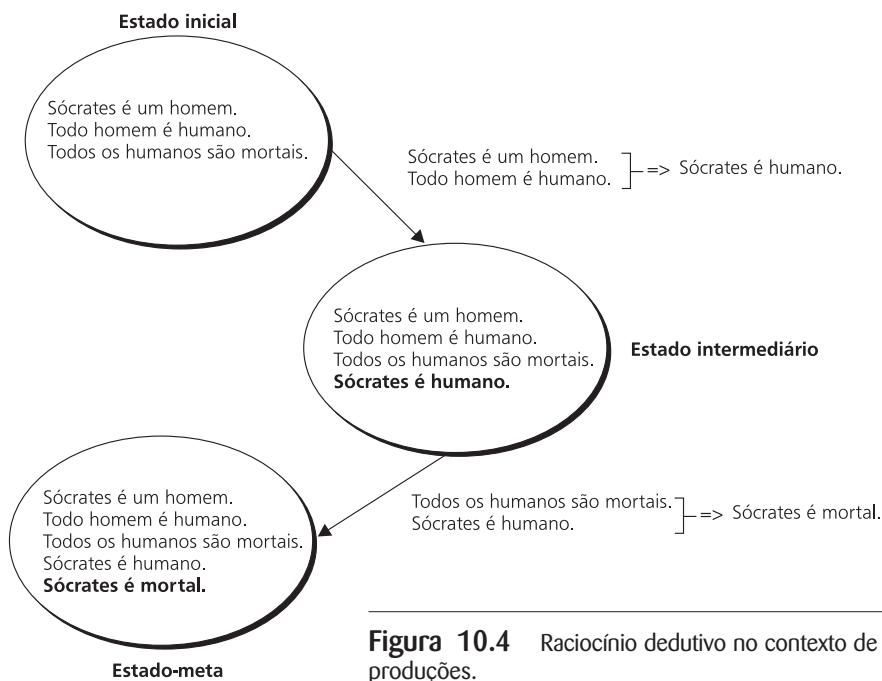


Figura 10.4 Raciocínio dedutivo no contexto de um sistema de produções.

Árvores de busca

Vimos que o trabalho do sistema de controle envolve a busca no grafo de estados para encontrar um caminho que leve do nó inicial até a meta desejada. Um método comumente utilizado para a execução de tal busca consiste em percorrer cada arco que sai do estado inicial e armazenar o seu estado-destino, em seguida percorrer os arcos que partem desses novos estados armazenar novamente os resultados, e assim por diante. A nossa busca da meta se propaga a partir do estado inicial, de uma forma semelhante a uma gota de tinta que se espalha na água. Este processo continua até que um dos novos estados seja a própria meta, o que significa que uma solução foi encontrada. O sistema de controle necessita apenas aplicar as produções encontradas ao longo da trajetória descoberta, quando esta é percorrida desde o estado inicial até a meta.

O resultado desta estratégia é a construção de uma árvore, chamada **árvore de busca**, que é a parte do grafo de estados consultada pelo sistema de controle. O nó-raiz da árvore de busca é o estado inicial, e os filhos de cada nó são os estados que o nó ancestral pode atingir aplicando uma produção. Cada arco que liga nós, em uma árvore de busca, representa a aplicação de uma única produção, e cada trajetória da raiz para uma folha representa uma trajetória entre os estados correspondentes do grafo de estados.

Se o quebra-cabeças de oito peças fosse configurado originalmente como está na Figura 10.5, a Figura 10.6 representaria a sua árvore de busca. O ramo mais à esquerda desta árvore representa uma tentativa de solução para o problema, movendo-se, primeiramente, a peça 6 para cima; o ramo central representa a tentativa de mover a peça 2 para a direita, e o ramo mais à direita, o deslocamento da peça 5 para baixo. Além disso, a árvore de busca mostra que, se iniciarmos deslocando a peça 6 para cima, a única produção permitida a seguir será mover a peça 8 para a direita. (Neste ponto, também poderíamos mover a peça 6 para baixo, mas retornariamos ao estado representado pelo nó-raiz, de modo que seria um movimento inútil.)

O estado-meta ocorre no último nível da árvore de busca da Figura 10.6. Dado que isto representa o término da busca, uma vez alcançado este ponto, o sistema de controle não precisará construir níveis adicionais da árvore. Assim que este nó for encontrado, o sistema de controle encerrará o seu procedimento de busca, e começará a construir a seqüência de instruções a ser utilizada para resolver o quebra-cabeças no ambiente externo. Isto corresponde ao simples processo de percorrer a árvore de busca, em direção ao topo, a partir do nó-meta, enquanto deposita em uma pilha as produções representadas pelos arcos, na ordem em que são encontradas. A aplicação desta técnica à árvore de busca da Figura 10.6, resulta na pilha de produções da Figura 10.7. Observe que o sistema de controle pode agora resolver o quebra-cabeças no mundo externo, pela execução das instruções que forem sendo desempilhadas.

Um ponto ainda permanece. Lembre que as árvores discutidas no Capítulo 7 utilizam um sistema de ponteiros que apontam árvore abaixo, permitindo-nos, assim, percorrê-la de um nó ancestral para os seus descendentes. No entanto, no caso de uma árvore de busca, o sistema de controle deve ser capaz de mover-se de um nó-filho para o seu ancestral, do modo como se move na árvore, do estado-meta para o estado inicial, porém em direção à raiz. Tais árvores são construídas com os seus sistemas de ponteiros apontando para cima, e não para baixo. Isto é, cada nó-filho contém um ponteiro para o seu ancestral, em vez de cada nó-ancestral conter ponteiros para seus filhos. (Em alguns casos, dois conjuntos de ponteiros são usados para permitir movimentos pela árvore nas duas direções).

1	3	5
4	2	
7	8	6

Figura 10.5 Um quebra-cabeças de oito peças, embaralhado.

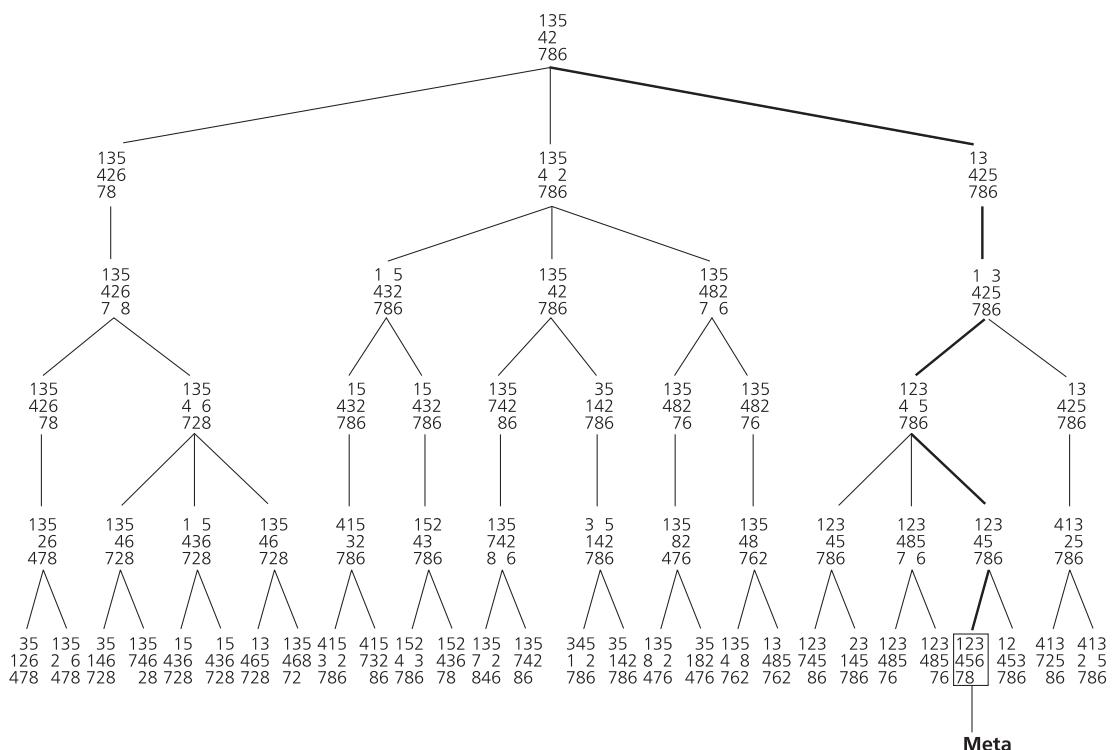


Figura 10.6 Uma árvore de busca simples.

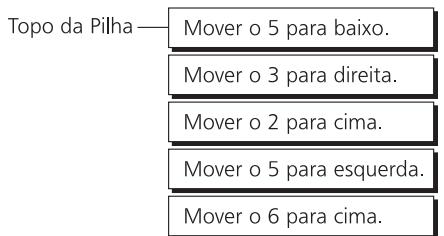


Figura 10.7 Produções empilhadas para posterior execução.

re de busca poderá ficar bem grande, se a meta não for rapidamente alcançada. Como resultado, verificamos que desenvolver uma árvore completa de pesquisa pode ser tão impraticável quanto representar o grafo de estados completo.

Uma estratégia para contornar esse problema consiste em modificar a ordem de construção da árvore de busca. Em vez de construí-la desenvolvendo-a **em largura**^{*} (isto é, a árvore é construída nível por nível), podemos iniciar desenvolvendo os caminhos mais prováveis, atingindo profundidades maiores, considerando as demais opções somente se constatarmos que as escolhas iniciais são inadequadas. Isto resulta na construção da árvore de busca **em profundidade**^{**}, o que significa que esta é construída com base em trajetórias verticais, e não em níveis horizontais.

A abordagem da busca em profundidade é similar à estratégia que nós, humanos, aplicaríamos quando confrontados com o quebra-cabeças de oito peças. Dificilmente seguiríamos várias opções simultaneamente, como a abordagem em largura implica. Em vez disso, provavelmente selecionaríamos a opção que nos parecesse mais promissora e a seguiríamos. Note que dissemos *parecesse* mais promissora. Raramente temos certeza de qual opção é melhor em cada situação. Simplesmente seguiríamos a nossa intuição, que obviamente pode nos levar a um beco sem saída. Não obstante, o uso de tal informação intuitiva parece dar aos seres humanos uma vantagem em relação aos métodos de força bruta, nos quais cada opção recebia a mesma atenção, e, portanto, parece prudente aplicar métodos intuitivos nos sistemas de controle automatizados.

Para isso, necessitamos uma maneira de identificar que estado parece ser o mais promissor. Nossa abordagem é usar uma **heurística**, que é um valor quantitativo associado a cada estado que tenta medir a “distância” desse estado até a meta mais próxima. Em um certo sentido, uma heurística é uma medida do custo projetado. Dada uma escolha entre dois estados, aquele com menor valor de heurística é por onde parece que a meta pode ser alcançada com o menor custo. Esse estado, portanto, deve nos dar a direção a seguir.

Uma heurística deve ter duas características. A primeira constitui uma estimativa razoável da quantidade de trabalho que ainda falta para a solução quando o estado associado é alcançado. Isso significa que ela pode fornecer informação valiosa ao selecionar a opção — quanto melhor a estimativa da heurística, melhor será a decisão baseada nessa informação. A segunda característica é que a heurística seja fácil de calcular. Isso significa que o seu uso pode representar uma contribuição ao processo de busca, e não um obstáculo. (Entretanto, embora o número verdadeiro de movimentos necessários para alcançar a meta a partir de um determinado estado seja uma excelente informação para auxiliar na tomada de decisão, calculá-lo implica encontrar antes a solução real.)

Uma heurística simples no caso do quebra-cabeças de oito peças seria estimar a “distância” à meta contando-se o número de peças que estão fora de posição — a conjectura é que um estado no qual

Heurística

Para o nosso exemplo da Figura 10.6, escolhemos uma configuração inicial que produz uma árvore de busca com dimensões manejáveis. No entanto, é possível imaginar que a árvore de busca gerada para resolver um problema mais complexo deva crescer muito mais do que a deste exemplo. Em um jogo de xadrez, existem 20 movimentos iniciais possíveis, e então o nó- raiz da árvore de busca neste caso tem 20 filhos no lugar dos três do caso do quebra-cabeças de oito peças, e um jogo de xadrez facilmente apresenta de 30 a 35 pares de movimentos, em contraste com os cinco movimentos imediatos do nosso exemplo.

Até mesmo no exemplo do quebra-cabeças de oito peças, a árvo-

*N. de T. Em inglês, *breadth-first*.

**N. de T. Em inglês, *depth-first*.

quatro peças estão fora de posição fica mais distante (e portanto é menos promissor) do que um estado no qual apenas duas peças estão fora de posição. Contudo, essa heurística não leva em conta o quanto cada peça está distante da respectiva posição final. Se as duas peças estiverem longe de suas posições, muitas produções serão necessárias para movê-las no quebra-cabeças.

Uma heurística ligeiramente melhor seria medir a distância entre cada peça e o seu destino e somar tais valores para obter uma única quantidade.

Uma peça imediatamente adjacente ao seu destino final é associada a uma distância de valor igual a um, enquanto uma cujo vértice toca o quadrado de seu destino final é associada a uma distância de valor igual a dois (pois deverá mover-se pelo menos uma posição na vertical e outra, na horizontal). Essa heurística é fácil de calcular e produz uma aproximação do número de movimentos necessários para alcançar a meta a partir deste estado. Por exemplo, a heurística associada à configuração da Figura 10.8 é sete (pois as peças 2, 5 e 8 estão a uma distância 1 de seus destinos, cada uma, enquanto as peças 3 e 6 estão, cada qual, a uma distância 2 do destino). Na verdade, seriam necessários sete movimentos para retornar esta configuração do quebra-cabeças à configuração-meta.

Agora que temos uma heurística para o quebra-cabeças de oito peças, o próximo passo é incorporá-la ao nosso processo de decisão. Para essa finalidade, lembramos que um ser humano, diante de uma decisão, tende a selecionar a opção que lhe parece mais próxima da meta. Então, o nosso procedimento de busca deve considerar a heurística de cada nó-folha da árvore e iniciar a busca a partir do nó-folha associado ao menor valor. Essa é a estratégia adotada na Figura 10.9, que apresenta um algoritmo para desenvolver uma árvore de busca e executar a solução obtida.

Aplicemos este algoritmo ao quebra-cabeças de oito peças, a partir da configuração inicial da Figura 10.5. Primeiramente, estabeleçamos o estado inicial como o nó-raiz e armazenemos a sua heurística, que é igual a cinco. Então, o primeiro passo no corpo da estrutura enquanto nos instrui a acrescentar os três nós que podem ser alcançados a partir do estado inicial, como consta na Figura 10.10. Observe

1	5	2
4	8	
7	6	3

Figura 10.8 Um quebra-cabeças de oito peças, embaralhado.

Definir o nó inicial do grafo de estados como a raiz da árvore de busca e armazenar a sua heurística.

enquanto (o nó-meta não foi alcançado) **faça**

[Selecionar o nó-folha mais à esquerda com o menor valor da heurística dentre todos os nós-folha. Associar a ele, como nós-filhos, os nós que puderem ser alcançados a partir deste nó selecionado aplicando-se uma única produção.

Registrar a heurística de cada novo nó ao lado do nó, na árvore de busca.

]

Percorrer a árvore de busca a partir do nó-meta em direção à raiz, empilhando a produção associada a cada arco percorrido.

Resolver o problema original executando as produções à medida que forem desempilhadas.

Figura 10.9 Um algoritmo para um sistema de controle que utiliza heurística.

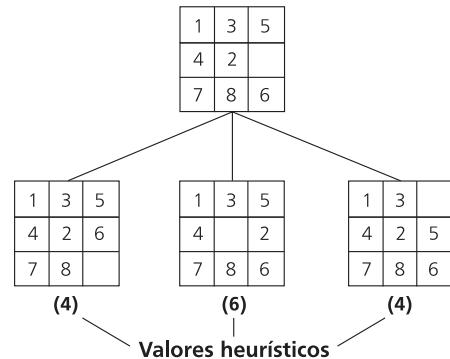


Figura 10.10 O começo da nossa busca heurística.

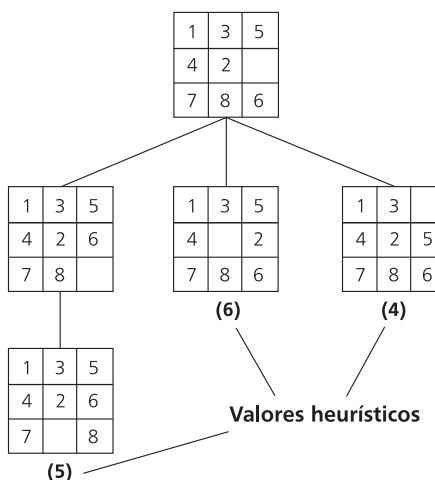


Figura 10.11 A árvore de busca, depois de duas passagens.

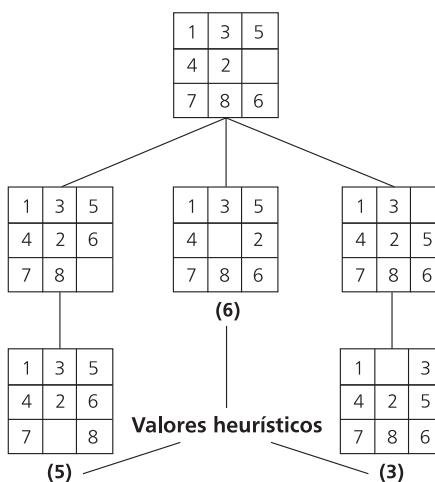


Figura 10.12 A árvore de busca, depois de três passagens.

que, no desenho, registramos a heurística de cada nó-folha entre parênteses, logo abaixo do mesmo.

Como o nó-meta não foi alcançado, executamos novamente o corpo da estrutura enquanto, desta vez estendendo a nossa busca a partir do nó mais à esquerda (“o nó-folha mais à esquerda com a menor heurística dentre todos os nós-folha”). Depois disto, a árvore de busca apresenta a forma descrita pela Figura 10.11.

O valor da heurística do nó-folha mais à esquerda agora é cinco, o que sugere que, afinal de contas, não é uma boa escolha. O algoritmo se adapta a esse fato e, na próxima passagem pela instrução iterativa, instrui-nos a expandir a árvore a partir do nó mais à direita (que agora é o nó-folha mais à esquerda com a menor heurística). Expandida dessa forma, a árvore de busca aparece conforme a Figura 10.12.

Neste ponto, o algoritmo parece estar no caminho correto. Dado que a heurística deste último nó é igual a três, a estrutura enquanto nos instrui a prosseguir nesta trajetória e, finalmente, a busca alcança a meta, o que resulta na árvore de busca descrita na Figura 10.13. Comparando-a à da Figura 10.6, verificamos que, embora com uma escolha temporariamente incorreta assumida no início pelo novo algoritmo, o uso da informação heurística reduziu sensivelmente a dimensão da árvore de busca, produzindo um processo muito mais eficiente.

Depois de atingir o estado-meta, a estrutura enquanto termina, e navegamos árvore acima, a partir do nó-meta, até a raiz, empilhando as produções encontradas ao longo do caminho. A pilha resultante está representada na Figura 10.7.

Finalmente, somos instruídos a executar tais produções à medida que forem desempilhadas. Nesse momento, observaremos a máquina de resolução de quebra-cabeças abaixar o seu dedo e começar a mover as peças.

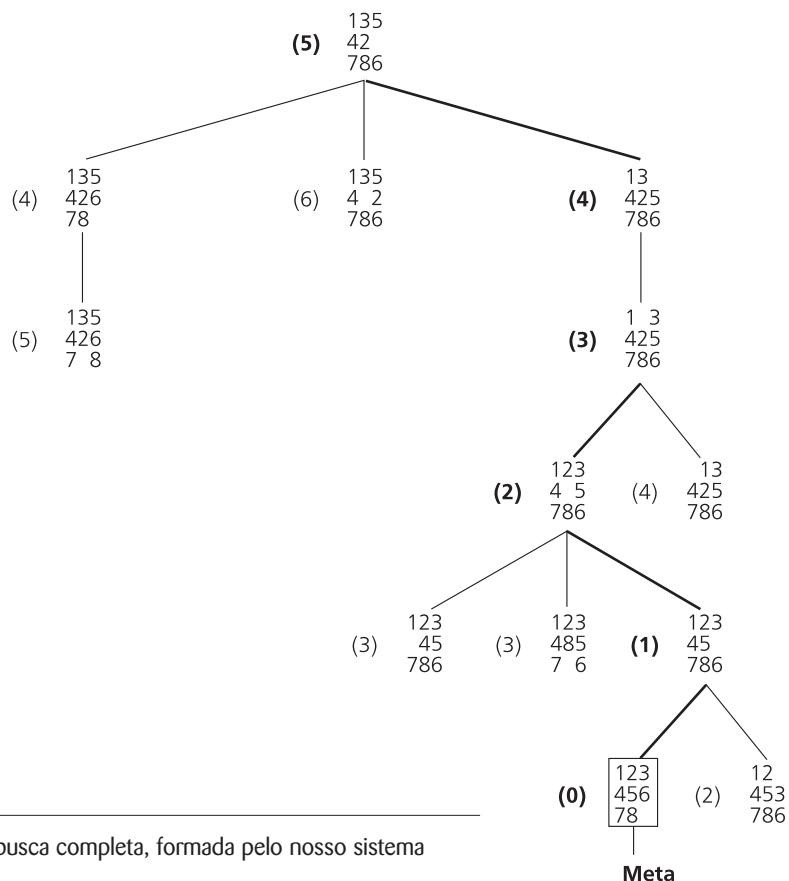


Figura 10.13 A árvore de busca completa, formada pelo nosso sistema heurístico.



QUESTÕES/EXERCÍCIOS

1. Qual é a importância dos sistemas de produções na inteligência artificial?
2. Faça uma parte do grafo de estados do quebra-cabeças de oito peças que cerca o nó que representa o seguinte estado:

4	1	3
	2	6
7	5	8

3. Utilizando o método de busca em largura, desenhe a árvore de busca construída por um sistema de controle ao resolver o quebra-cabeças de oito peças com o seguinte estado inicial:

1	2	3
4	8	5
7	6	

4. Com lápis e papel e usando o método de busca em largura, construa a árvore de busca resultante do quebra-cabeças de oito peças com o seguinte estado inicial.

4	3	
2	1	8
7	6	5

(Você não precisa terminar.) Que problemas você encontrou?

5. Qual analogia pode existir entre o nosso sistema heurístico para a resolução de quebra-cabeça de oito peças e um alpinista que tenta alcançar o cume considerando somente o terreno local e sempre prosseguindo em direção a uma posição ascendente?
6. Utilizando a informação heurística apresentada nesta seção, aplique o algoritmo de sistema de controle da Figura 10.9 para o problema de resolução do seguinte quebra-cabeças de oito peças:

1	2	3
4		8
7	6	5

7. Refine o nosso método de cálculo do valor da heurística para um estado, de forma que o algoritmo de busca da Figura 10.9 não faça uma escolha errada, como fez no exemplo nesta seção. Seria possível encontrar um exemplo em que o seu sistema proposto continuasse a encaminhar a busca incorretamente?

10.4 Redes neurais artificiais

Apesar de todo o progresso alcançado na inteligência artificial, muitos problemas na área continuam desafiando a capacidade dos computadores baseados na arquitetura de von Neumann. Unidades centrais de processamento que executam seqüências de instruções não parecem capazes de perceber e raciocinar de forma comparável aos multiprocessadores da mente humana. Por isso, muitos pesquisadores estão se voltando para máquinas com outras arquiteturas. Uma delas é a rede neural artificial.

Propriedades básicas

Conforme introduzido no Capítulo 2, as redes neurais artificiais são construídas a partir de muitos processadores individuais, que chamaremos de *unidades de processamento* (ou simplesmente *unidades*), arranjados de uma forma que imita as redes de neurônios existentes em sistemas biológicos. Um neurônio biológico é uma única célula com tentáculos de entrada chamados dendrites e um tentáculo de saída chamado axônio (Figura 10.14). Os sinais transmitidos pelo axônio da célula refletem o estado em que ela se encontra: inibida ou excitada. Esse estado é determinado pela combinação de sinais recebidos pelos seus dendrites, os quais pegam os sinais de outros axônios de outras células através de pequenos espaços. Tais conexões são conhecidas como sinapses. Os pesquisadores sugerem que a condutividade ao longo de uma única sinapse seja controlada pela composição química da sinapse. Isto é, a possibilidade de um sinal particular de entrada ter efeito inibidor ou excitante no neurônio é determinada pela composição química da sinapse. Assim, acredita-se que uma rede neural biológica aprenda ajustando essas conexões químicas entre neurônios.

Uma unidade de processamento em uma rede neural artificial é um dispositivo simples, que imita esse entendimento básico sobre o neurônio biológico. Ela produz uma saída 1 ou 0, dependendo de a entrada efetiva desta unidade exceder ou não um determinado valor de referência. Esta entrada efetiva é uma soma ponderada das entradas correntes, conforme está representado na Figura 10.15. Nessa figura, as saídas de três unidades de processamento (denotadas por v_1 , v_2 e v_3) são usadas como entradas para outra unidade. As entradas para esta quarta unidade estão associadas ao receptor multiplica cada valor de entrada em seguida, soma esses resultados, com o valor de referência da unidade de próprio, ela produzirá um 0 em sua saída.

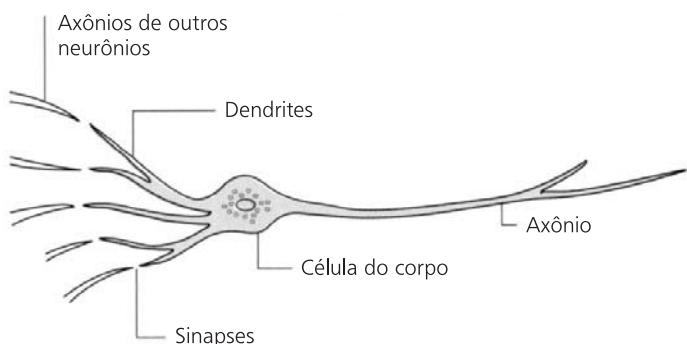


Figura 10.14 Um neurônio em um sistema biológico vivo.

para outra unidade. As entradas para esta quarta unidade estão associadas a valores chamados pesos (denotados por w_1 , w_2 e w_3). A unidade receptora multiplica cada valor de entrada pelo peso associado à posição de entrada correspondente e, em seguida, soma esses resultados, obtendo a entrada efetiva ($v_1w_1 + v_2w_2 + v_3w_3$). Se esta exceder o valor de referência da unidade de processamento, a unidade produzirá uma saída igual a 1; caso contrário, ela produzirá um 0 em sua saída.

Seguindo as indicações da Figura 10.15, adotamos a convenção de representar as unidades de processamento por meio de retângulos. Na extremidade de entrada da unidade, colocamos um retângulo menor para cada entrada, com o seu peso associado. Finalmente, escrevemos o valor de referência da unidade no retângulo maior. Para ilustrar, a Figura 10.16 representa uma unidade de processamento com três entradas e um valor de referência de 1,5. A primeira entrada é considerada com peso -2; a segunda, com peso 3 e a terceira, com peso -1. Assim, se a unidade receber as entradas 1, 1 e 0, sua entrada efetiva será $(1)(-2) + (1)(3) + (0)(-1) = 1$, e portanto sua saída será 0. Entretanto, se a unidade receber 0, 1 e 1, sua entrada efetiva será $(0)(-2) + (1)(3) + (1)(-1) = 2$, que excede o valor de referência, de modo que a saída da unidade será 1.

O fato de um peso poder ser positivo ou negativo significa que as entradas correspondentes têm efeitos de inibição e de excitação sobre a unidade receptora. (Se o peso for negativo, então uma entrada 1 nesta posição reduzirá a soma ponderada e, portanto, tenderá a manter a entrada efetiva abaixo do valor de referência. Entretanto, um peso positivo faz com que a entrada associada tenha um efeito cres-

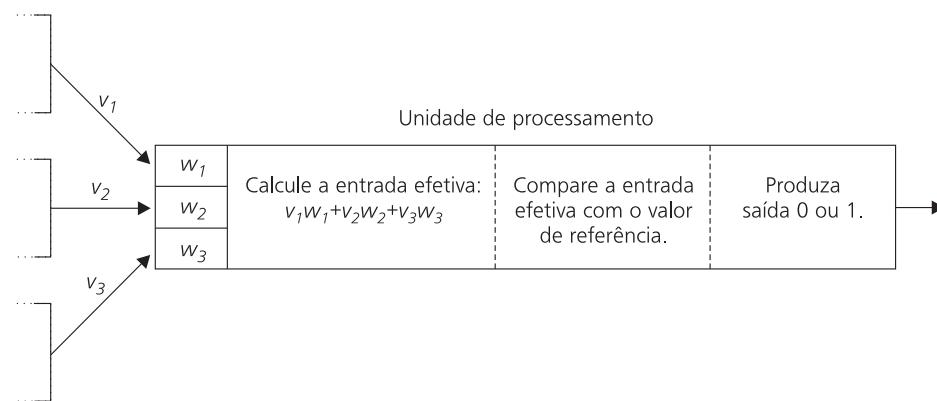


Figura 10.15 As atividades internas de uma unidade de processamento.



Figura 10.16 Representação de uma unidade de processamento.

cente sobre a soma ponderada, aumentando as possibilidades de que a soma efetiva exceda o valor de referência.) Além disso, o valor real do peso controla o grau em que a entrada correspondente inibe ou excita a unidade receptora. Por conseguinte, ajustando os valores dos pesos ao longo de uma rede neural artificial, podemos programá-la para responder a diferentes entradas, de uma forma predeterminada.

Como exemplo, a rede simples apresentada na Figura 10.17(a) está programada para produzir uma saída 1 se suas duas entradas forem diferentes e uma saída 0 em caso contrário.

Devemos notar que a rede na Figura 10.17 é muito simplista em comparação com uma rede biológica real. O cérebro humano contém aproximadamente 10^{11} neurônios, e cada um pode participar de 10^4 sinapses. De fato, os dendrites em um neurônio biológico são tão numerosos que parecem mais uma rede fibrosa do que tentáculos individuais, como representado em nossas figuras.

Uma aplicação específica

Para sentir o poder das redes neurais artificiais, retornemos ao problema do reconhecimento de caracteres. Em particular, consideremos o problema de distinguir entre as letras maiúsculas C e T, conforme está ilustrado na Figura 10.18. Deseja-se identificar qualquer letra colocada no campo de visão, independentemente de sua orientação. Todos os padrões da Figura 10.19(a) devem ser identificados como Cs, e todos os da parte (b) como Ts.

Começamos assumindo que o campo de visão consiste em pixels quadrados, do tamanho dos quadrados a partir dos quais as letras são construídas. A cada pixel corresponde um sensor que produz um 1

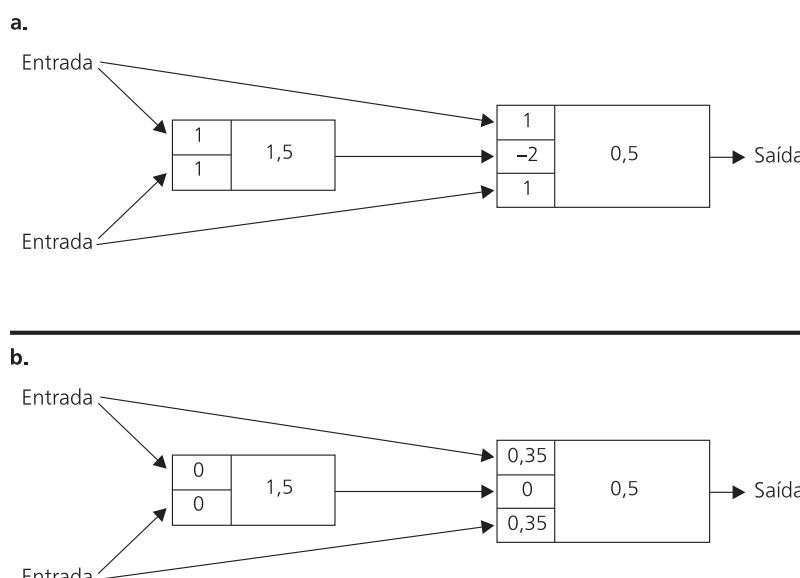


Figura 10.17 Uma mesma rede neural, com dois programas diferentes.

se este *pixel* estiver coberto pela letra observada e um 0 em caso contrário. Assim, utilizamos as saídas de tais sensores como entradas para a nossa rede neural artificial.

Ela contém dois níveis de unidades de processamento. O primeiro consiste em muitas unidades — uma para cada bloco de três por três *pixels* no campo de visão (veja a Figura 10.20). Cada unidade possui nove entradas, nas quais estão fixados os sensores associados ao bloco de três por três correspondente a essa unidade. (Note que os blocos de três por três associados com as unidades no primeiro nível se sobrepõem. Assim, cada sensor fornece entrada para as nove unidades de processamento do primeiro nível.)

O segundo nível da nossa rede consiste em uma única unidade de processamento, com uma entrada separada para cada unidade do primeiro nível. A unidade de processamento do segundo nível tem um valor de referência igual a 0,5 e a cada uma de suas entradas está associado o peso 1. Assim, esta unidade de nível mais alto produzirá uma saída 1 se e somente se pelo menos uma de suas entradas for 1.

Cada unidade de processamento do primeiro nível também possui o valor de referência 0,5. A cada entrada é dado um peso -1, com exceção da entrada associada ao *pixel* central desse bloco de três por três da unidade, que recebe peso 2. Cada unidade poderá, portanto, produzir uma saída 1 somente se receber um 1 do sensor associado ao *pixel* central do bloco de três por três.

Agora, se a letra C for colocada no campo de visão (Figura 10.21), todas unidades de processamento do primeiro nível produzem uma saída 0. Isto é porque as únicas unidades cujos *pixels* centrais são cobertos pela letra também terão no mínimo dois outros *pixels* em seus blocos três por três cobertos, e os sinais recebidos por estes sensores negarão o efeito do *pixel* central. Portanto, se a letra no campo de visão for um C, todas as entradas para a unidade de processamento do segundo nível serão 0, o que significa que a saída da rede inteira será 0.

Em contraste, suponhamos que a letra no campo de visão seja T e consideremos o bloco de três por três cujo centro é o quadrado coberto pela base da haste da letra T (Figura 10.22). A unidade de processamento associada a este quadrado recebe uma entrada efetiva de 1 (2 do *pixel* central e -1 do outro *pixel* coberto pela haste). Isto excede a referência da unidade, de modo que esta envia uma saída

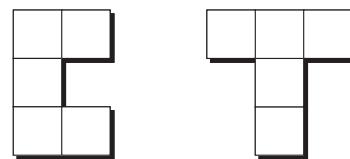


Figura 10.18 Letras maiúsculas C e T.

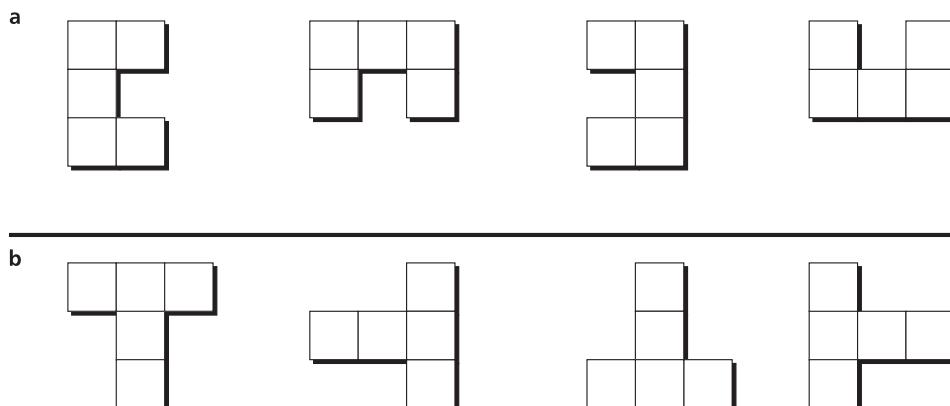


Figura 10.19 Diversas orientações das letras C e T.

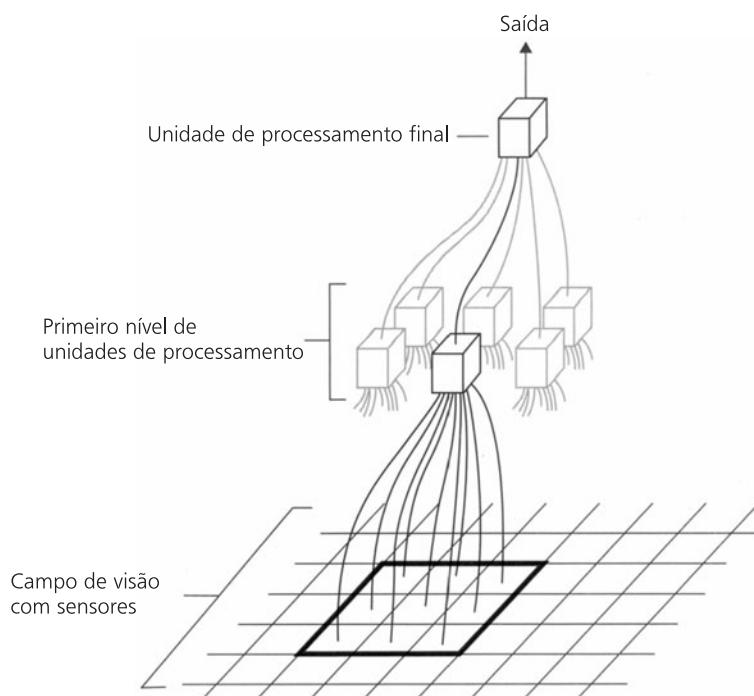


Figura 10.20 A estrutura do sistema de reconhecimento de caracteres.

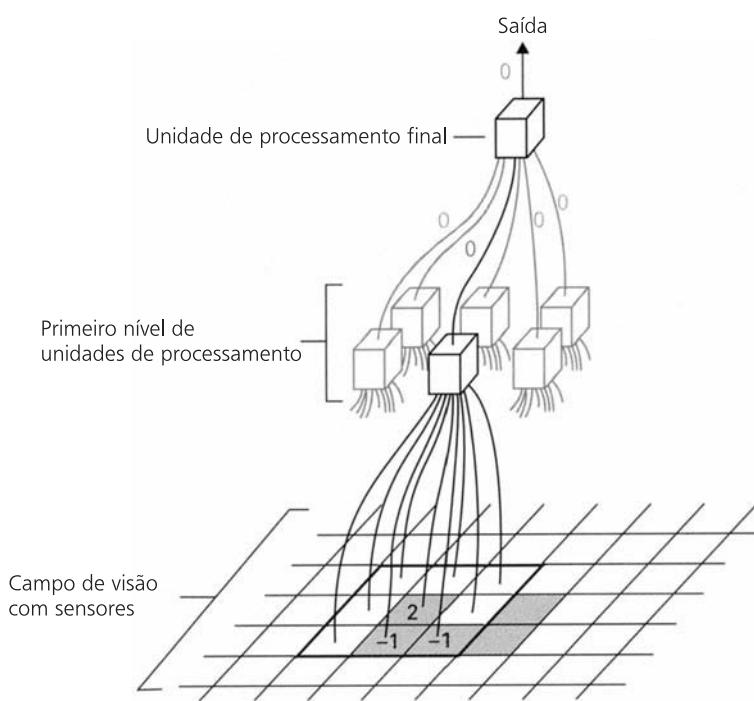


Figura 10.21 A letra C no campo de visão.

igual a 1 para a unidade de nível superior. Isto força a unidade de nível superior a produzir uma saída igual a 1.

Em resumo, temos uma rede neural artificial que distingue as letras C e T, independentemente da orientação da letra no campo de visão. Se a letra for um C, a rede produzirá 0 como saída, e se for um T, uma saída igual a 1.

Naturalmente, a capacidade de distinguir apenas entre duas letras está muito aquém da capacidade da mente humana de processar imagens. Entretanto, a astúcia das soluções justifica plenamente o prosseguimento das pesquisas na área.

O resultado é que o campo das redes neurais artificiais é uma área ativa de pesquisa. Os principais obstáculos estão associados com o projeto e a programação de tais redes. Objetivos comuns relacionados ao projeto de redes incluem a determinação de quantas unidades de processamento são necessárias para resolver certos problemas e que padrões de conexão entre essas unidades são os mais produtivos.

Quanto ao tópico de programação da rede, já mencionamos que tal tarefa consiste em associar pesos adequados às diversas entradas das unidades de processamento ao longo do sistema. Isso normalmente é feito por meio de um processo repetitivo de treinamento, no qual é aplicada à rede uma amostra de entradas para que, em seguida, os pesos sejam ajustados em pequenos incrementos, de forma que a saída corrente da rede vá se

aproximando da saída desejada. Como este processo é repetido sobre uma amostra de entradas, espera-se que os pesos precisem ser cada vez menos reajustados, até que a rede comece a operar corretamente sobre todos os dados da amostra. É necessária uma estratégia de reajuste desses pesos para que, a cada novo ajuste, a rede vá se aproximando da meta final, em vez de destruir eventuais avanços obtidos a partir das amostras anteriores.

Memória associativa

Uma característica impressionante da mente humana é a habilidade de recuperar a informação associada a um tema corrente. Quando experimentamos certos aromas, rapidamente relembramos nossa

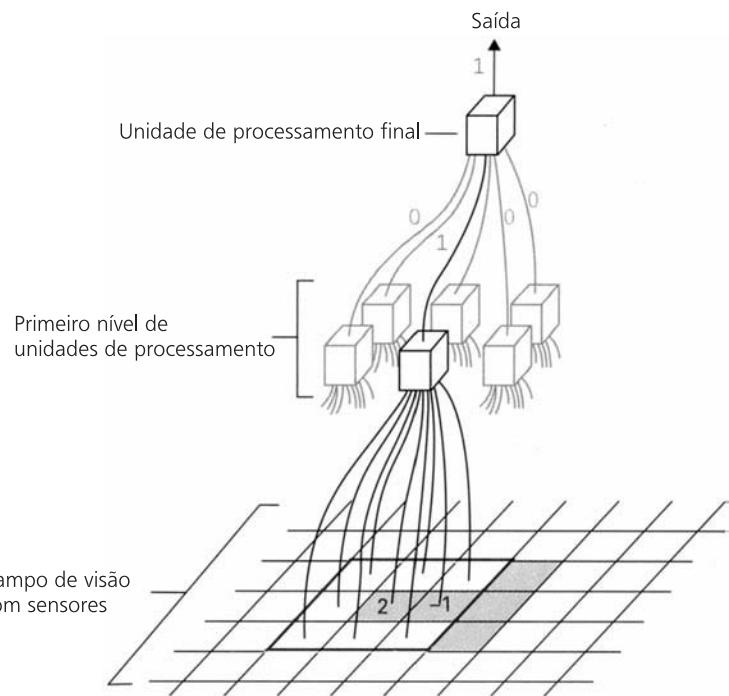


Figura 10.22 A letra T no campo de visão.

Pequena escala à grande escala

Um fenômeno recorrente no desenvolvimento e teste de teorias é a transição da experimentação em pequena escala para as aplicações em grande escala. A experimentação inicial de uma nova teoria freqüentemente envolve casos pequenos e simples. Se o sucesso é alcançado, então o ambiente experimental é estendido para sistemas mais realistas, em grande escala. Algumas teorias são aptas a sobreviver nessa transição; outras não. Às vezes, o sucesso na pequena escala é grande o suficiente para encorajar os proponentes da teoria a persistir após as falhas na grande escala terem desencorajado outros pesquisadores. Em alguns casos, essa persistência finalmente vale a pena; em outros, representa esforço desperdiçado.

Tais cenários são prontamente observados no campo da inteligência artificial. Um exemplo está na área de processamento de linguagens naturais, na qual sucessos antigos em situações limitadas levaram muitos a acreditar que o entendimento geral de linguagens naturais estava apenas pouco acima no horizonte. Infelizmente, estender o sucesso para a grande escala demonstrou ser muito mais difícil, e tem-se vencido morosamente como resultado de esforços significativos. Outro exemplo é o tema das redes neurais artificiais, tópico que entrou em cena com uma pompa significativa, declinou durante anos quando as suas capacidades de grande escala foram questionadas e agora retorna em uma atmosfera mais conquistadora. Como indicado no texto, o tema dos algoritmos genéticos atualmente está realizando o teste da transição. Se a abordagem evolucionária provará ser ferramenta útil no futuro, é uma questão em aberto.

infância. O som da voz de um amigo pode evocar a imagem de uma pessoa ou talvez memórias de bons tempos. Algumas músicas trazem lembranças de um período específico de férias. Esses são exemplos da **memória associativa** — isto é, a recuperação da informação que está associada com a informação corrente ou lhe é relevante.

A construção de máquinas com memória associativa tem sido uma meta de pesquisa há muitos anos. Uma abordagem é a aplicação das técnicas de redes neurais artificiais. Por exemplo, considere uma rede que consista em muitas unidades de processamento interconectadas para formar uma teia sem entradas nem saídas. (Em alguns projetos, chamadas redes *Hopfield*, a saída de cada unidade de processamento é conectada a cada entrada das outras; em outros casos, a saída de uma unidade é conectada apenas às suas vizinhas.) Cada unidade pode estar em seu estado inibido ou excitado. Se representarmos um estado excitado por 1 e um estado inibido por 0, a condição da rede completa poderá ser visualizada como uma configuração de 0s e 1s. Agora suponha que a rede seja programada de modo que certas configurações de 0s e 1s sejam estáveis, no sentido em que, quando a rede se encontra em uma dessas configurações, ela permanece lá. Entretanto, se a rede estiver em uma configuração não-estável, a interação das unidades de processamento causará a mudança de configuração — e continuará mudando até a rede cair em uma configuração estável.

Se iniciarmos a rede em uma configuração não-estável que fique próxima de uma estável, esperaremos que ela caia na configuração estável. Em um certo sentido, quando é fornecida uma parte de uma configuração estável, a rede é capaz de completar a configuração. Ou, em outras palavras, ela é capaz de encontrar o padrão de bits associado ao padrão parcial que lhe foi dado. Assim, se alguns bits são usados para codificar aromas e outros para codificar memórias da infância, então o ajuste dos bits de aroma de acordo com uma configuração estável poderia fazer com que os restantes encontrem seus lugares para a memória associada com a infância.

Vamos considerar a rede neural artificial mostrada na Figura 10.23. Cada círculo na figura representa uma unidade de processamento cujo valor de referência está registrado no círculo. As linhas que ligam os círculos representam as conexões entre as unidades correspondentes. Cada conexão é bidirecional — isto é, uma linha que liga duas unidades indica que a saída de cada unidade está conectada como entrada à outra. Assim, a saída da unidade central está conectada como entrada a cada unidade ao longo do perímetro, e a saída de cada unidade está conectada como entrada à unidade central. Duas unidades conectadas associam o mesmo peso à saída de cada uma. Esse peso comum está mostrado próximo à linha que liga as unidades. Assim, a unidade do topo da figura associa o peso -1 com a entrada que ela recebe da unidade central e o peso 1 com a entrada que ela recebe das suas duas vizinhas no perímetro.

De modo semelhante, a unidade central associa o peso -1 com cada valor recebido das unidades ao longo do perímetro.

A rede opera em passos discretos, nos quais todas as unidades de processamento respondem às usas entradas de maneira sincronizada. Para determinar a próxima configuração da rede a partir da configuração corrente, determinamos a entrada efetiva de cada unidade da rede e então fazemos todas as unidades responderem às suas entradas ao mesmo tempo. O efeito é que a rede inteira segue uma seqüência coordenada de calcular as entradas efetivas, responder às entradas, calcular as entradas efetivas, responder às entradas etc.

Considere a seqüência de eventos que ocorreria se iniciássemos a rede com as suas duas unidades mais à direita inibidas e as outras excitadas. (Figura 10.24a). As duas mais à esquerda teriam entradas efetivas de 1, e então continuariam excitadas. Entretanto, as suas vizinhas no perímetro teriam entradas efetivas de 0, e ficariam inibidas. Do mesmo modo, a unidade central teria uma entrada efetiva de -4 e ficaria inibida. Assim, a rede inteira se deslocaria para a configuração mostrada na Figura 10.24b, na qual apenas as duas unidades mais à esquerda estão excitadas. Uma vez que a unidade central estaria agora inibida, as condições excitadas das

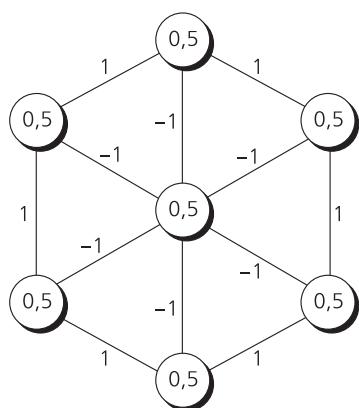


Figura 10.23 Uma rede neural artificial implementando uma memória associativa.

unidades mais à esquerda causariam novamente a excitação das unidades de cima e de baixo. Por sua vez, a unidade central permaneceria inibida, já que teria uma entrada efetiva de -2. Assim, a rede se deslocaria para a configuração mostrada na Figura 10.24c, que levaria então à configuração na Figura 10.24d. (Você pode desejar confirmar que um fenômeno pisca-pisca ocorreria se a rede fosse iniciada apenas com as quatro unidades de cima excitadas. A unidade do topo permaneceria excitada, enquanto as suas duas vizinhas no perímetro e a unidade central alternariam entre a excitação e a inibição.)

Finalmente, observe que a rede possui duas configurações estáveis: uma na qual a unidade central está excitada e as outras inibidas, e outra na qual a unidade central está inibida e as outras estão excitadas. Se iniciarmos a rede com a unidade central excitada e não mais que duas das outras unidades excitadas, ela cairá na primeira configuração estável. Se iniciarmos com no mínimo quatro unidades adjacentes no perímetro em seus estados excitados, a rede cairá na última configuração. Assim, poderíamos dizer que a rede associa a primeira configuração estável com padrões iniciais nos quais a unidade central e menos de três unidades no perímetro estejam excitadas e associa a última configuração estável com padrões iniciais nos quais quatro ou mais unidades do perímetro estão excitadas. Em síntese, a rede representa uma memória associativa elementar.

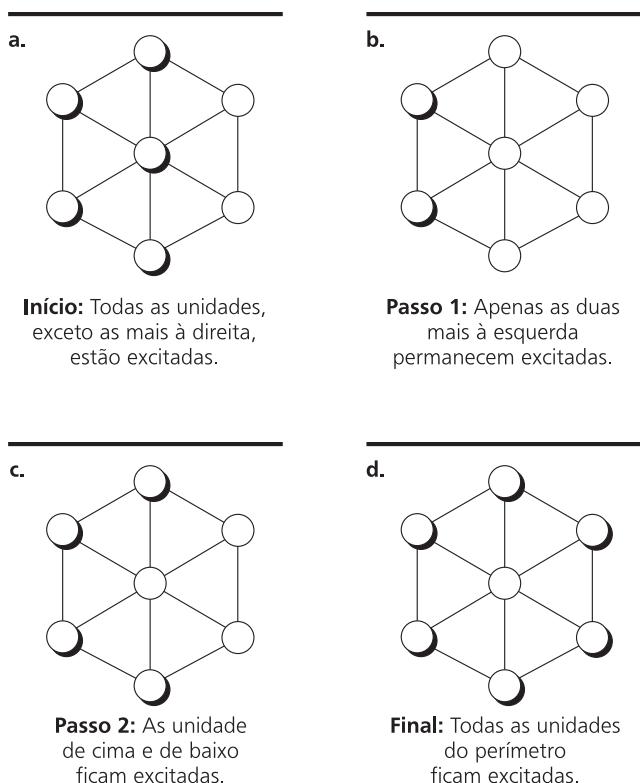
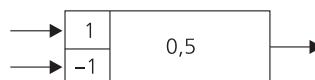


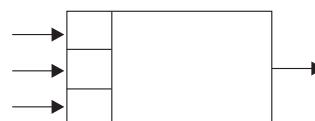
Figura 10.24 Os passos que levam a uma configuração estável.

QUESTÕES/EXERCÍCIOS

- Qual é a saída da seguinte unidade de processamento quando as suas duas entradas são 1s? E quando os padrões de entrada são 0, 0; 0, 1 e 1, 0?



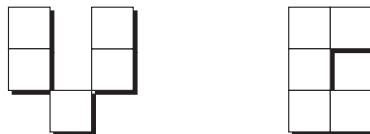
- Ajuste os pesos e o valor de referência da seguinte unidade de processamento de modo que a sua saída seja 1 se e somente se ao menos duas de suas entradas forem 1s.



3. Projete uma rede neural artificial que detecte qual dos padrões está em seu campo de visão.



4. Projete uma rede neural artificial que detecte qual dos padrões está em seu campo de visão.



5. Em qual configuração estável a rede da Figura 10.23 cairá se for iniciada com todas as suas unidades de processamento inibidas?

10.5 Algoritmos genéticos

O campo dos **algoritmos genéticos** é a área de pesquisa que procura aplicar o nosso entendimento da evolução natural à tarefa de resolução de problemas. A abordagem é misturar as melhores de uma coleção de soluções propostas para obter outra geração de soluções propostas que represente um aperfeiçoamento em relação à coleção original. Com a repetição desse processo, espera-se simular o processo evolucionário e enfim obter soluções viáveis para o problema em mãos.

Como exemplo, vamos supor que você jogue pôquer com o mesmo grupo de amigos toda quarta-feira à noite e queira desenvolver uma estratégia que maximize as suas vitórias. Uma abordagem evolucionária a esse problema começaria com a identificação das várias situações que podem ocorrer no jogo de pôquer e as respostas em potencial para elas. Isto, é claro, seria um enorme empreendimento, uma vez que haveria muitas situações a considerar. Uma vez a análise feita, porém, poderíamos construir uma estratégia de jogo atribuindo uma resposta a cada situação. Assim, representaríamos cada estratégia como uma lista na forma $S_1R_1, S_2R_2 \dots S_nR_n$, onde cada S é uma situação em potencial e o R seguinte, a resposta a ser usada nessa situação. As representações de estratégias diferentes conteriam a mesma seqüência de situações, mas essas situações seriam seguidas por diferentes respostas. Para aplicar uma determinada estratégia, você simplesmente encontra a situação apropriada na lista e então realiza a resposta associada.

O próximo passo em nosso problema do jogo de pôquer seria selecionar uma coleção inicial de estratégias, aplicá-la durante uma noite de jogo e registrar os ganhos obtidos com cada estratégia. Baseados nessa análise, selecionaríamos então as melhores estratégias iniciais e as agruparíamos em pares. De cada par, produziríamos duas novas estratégias, primeiramente partindo as listas que representam as duas estratégias do par e então juntando a cabeça de uma com a cauda da outra (Figura 10.25). Todas essas novas estratégias formariam a coleção a ser testada na semana seguinte. Cada semana, selecionaríamos novamente as melhores estratégias e as usaríamos para produzir outra geração de estratégias a ser testada na próxima semana. Assim, à medida que as semanas fossem passando, o nosso processo simulava a evolução natural, em que os sobreviventes de cada geração se reproduzem para dar origem à próxima. De fato, poderíamos até simular mutações, alterando arbitrariamente, de tempos em tempos, uma resposta dentro de uma estratégia.

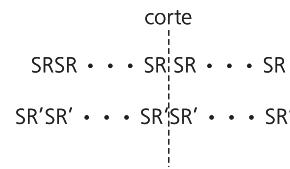
Infelizmente, o nosso exemplo do pôquer é um tanto irrealista, uma vez que não seria possível testar uma extensa geração de estratégias em uma única noite de jogo. Entretanto, se o processo de teste pudesse ser automatizado, como é caso de muitas aplicações, a abordagem evolucionária será

perfeitamente viável. Este é o processo proposto pelos pesquisadores em algoritmos genéticos. Primeiro é encontrada uma maneira de representar soluções em potencial como cadeias de símbolos. Então, uma coleção de soluções em potencial é gerada e testada. Os melhores exemplares dessa coleção são então cruzados para formar uma nova geração de soluções em potencial, que é testada e usada para formar outra. Em alguns momentos, mutações aleatórias podem ser inseridas durante o processo de cruzamento.

A abordagem evolucionária tem sido aplicada em numerosas situações com resultados promissores. Uma delas é o projeto de configurações para redes neurais artificiais. Suponha, por exemplo, que desejemos resolver um problema por meio de uma rede neural artificial que consista em um número predeterminado de unidades de processamento. Antes de começarmos a treinar na rede, precisamos decidir como as unidades serão interligadas. Para isso, concordamos em codificar as configurações da rede neural artificial como cadeias de 0s e 1s da seguinte maneira: assumindo que o número de unidades de processamento a serem usadas na rede é 5, primeiramente rotulamos as unidades com os números inteiros de 1 a 5 (Figura 10.26a). Construímos então uma tabela quadrada com 5 linhas e 5 colunas (Figura 10.26b). Colocamos um 1 na linha I e coluna J se a rede a ser codificada possuir uma conexão entre a saída de unidade I de processamento e a entrada da unidade J. A todos os outros elementos da tabela, é atribuído o valor 0. Então, reescrivemos essa tabela como uma única cadeia, dispondo as linhas uma atrás da outra (Figura 10.26c).

Agora, para encontrar uma boa configuração para o nosso problema particular, selecionamos diversas configurações em potencial e iniciamos o processo de treinamento com cada uma. Após um breve treinamento, selecionamos as configurações que parecem fazer o maior progresso, as representamos como cadeias de 0s e 1s e cruzamos essas cadeias para obter uma nova geração de configurações em potencial. Por sua vez, essa nova geração seria parcialmente experimentada e as melhores configurações seriam selecionadas para produzir outra geração. Essa abordagem tem tido bom desempenho no projeto de redes neurais artificiais simples.

- a. Duas estratégias são cortadas no mesmo lugar.



- b. A cauda de cada estratégia é anexada à cabeça da outra, produzindo duas novas estratégias.

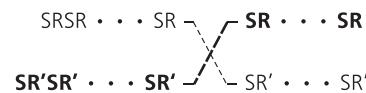
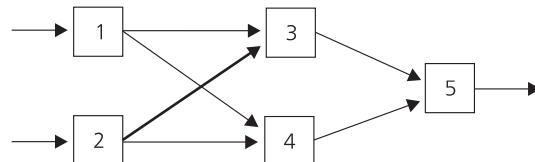


Figura 10.25 Cruzamento de duas estratégias no jogo de pôquer.

a. A configuração de uma rede neural artificial.



b. Uma tabela que indica como as unidades na rede estão conectadas.

	1	2	3	4	5
1	0	0	1	1	0
2	0	0	1	1	0
3	0	0	0	0	1
4	0	0	0	0	1
5	0	0	0	0	0

c. Versão codificada da rede neural artificial.

0011000110000010000100000

Figura 10.26 Codificação da topologia de uma rede neural artificial.

As técnicas dos algoritmos genéticos também têm sido aplicadas à tarefa de desenvolvimento de programas, levando a um campo chamado **programação evolucionária**. A meta é desenvolver programas permitindo que eles evoluam, em vez de serem explicitamente escritos. Um passo importante nesse cenário é encontrar maneiras de trocar partes de programas para produzir programas novos que façam sentido. O paradigma da programação funcional tem demonstrado ser útil nesse contexto. De fato, um programa escrito nesse paradigma consiste em funções aninhadas, onde a saída de uma função é usada como entrada para outra. Assim, é viável que uma função de um programa seja trocada por uma função de outro, sem destruir a estrutura de ambos.

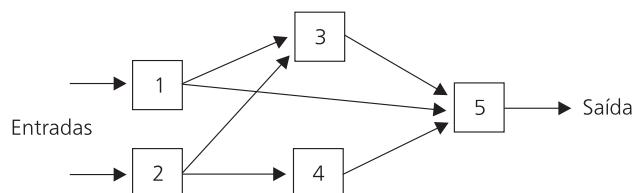
Seguindo esse raciocínio, os pesquisadores têm aplicado as técnicas da programação evolucionária ao processo de desenvolvimento de programas usando linguagens funcionais. A abordagem tem sido iniciar com uma coleção de programas que contenha uma rica variedade de funções. As funções dessa coleção inicial formam o “banco de genes”, a partir do qual futuras gerações de programas são construídas. Então, faz-se o processo evolucionário executar por muitas gerações na esperança de que, ao se produzir cada geração a partir dos melhores da geração passada, uma solução para o problema esteja finalmente se desenvolvendo.

O campo da programação evolucionária ainda está em sua infância. Todavia, tem-se obtido sucesso em casos simples. Por exemplo, as técnicas evolucionárias têm sido usadas para obter programas que calculam as propriedades de formas geométricas simples, como a área de um quadrado ou a circunferência de um círculo. Um problema maior é identificar os “melhores” em um grupo de programas no qual nenhum parece estar de modo algum próximo ao produto desejado. Como acontece em outras áreas dos algoritmos genéticos, se as técnicas em desenvolvimento serão bem-sucedidas na grande escala para resolver problemas significativos, ainda é uma questão em aberto. De qualquer modo, o campo dos algoritmos genéticos é representativo da criatividade e da imaginação que estão sendo aplicadas nas pesquisas atuais na ciência da computação.



QUESTÕES/EXERCÍCIOS

1. Codifique a configuração da rede neural artificial abaixo usando o sistema descrito nesta seção.



2. Selecione um problema que você considere adequado à abordagem evolucionária e descreva como as soluções em potencial poderiam ser codificadas como cadeias. Então, descreva como essas cadeias poderiam ser cruzadas para formar futuras gerações.
3. Apresente um argumento que sustente que o paradigma da programação funcional é mais compatível com a programação evolucionária do que o paradigma orientado a objeto.

10.6 Outras áreas de pesquisa

Vamos agora dar uma rápida olhada em algumas outras áreas de pesquisa no campo da inteligência artificial.

Processamento de linguagens

Comecemos considerando a tarefa de traduzir declarações de uma linguagem para outra. No Capítulo 5, aprendemos os rudimentos do processo de tradução (análise lexical, análise sintática e geração de código) e como esses passos são aplicados na tarefa de traduzir programas em linguagem de alto nível para linguagem de máquina de baixo nível. A capacidade de fazer a tradução dá a ilusão de que a máquina de fato entende a linguagem que está sendo traduzida. (Lembre-se, na Seção 5.1, da história contada por Grace Hopper a respeito dos gerentes que pensavam ter ela ensinado os computadores a entender o alemão.) Sem dúvida, como os primeiros tradutores foram desenvolvidos na década de 1940, muitos pesquisadores acreditavam que a capacidade de programar computadores para entender as linguagens naturais era uma questão para apenas alguns anos.

O que tais pesquisadores não perceberam era a distância entre as linguagens formais de programação e as linguagens naturais, como o inglês, o alemão e o latim. As linguagens de programação são construídas a partir de primitivas bem projetadas de modo que cada instrução possui apenas uma estrutura gramatical e somente um significado. Entretanto, uma sentença em uma linguagem natural pode ter múltiplos significados, dependendo do contexto ou até mesmo da maneira como é comunicada. Assim, para entender uma linguagem natural, as pessoas se apóiam fortemente no conhecimento adquirido e nas suas habilidades de memória associativa.

Por exemplo, as sentenças*

*Norman Rockwell painted people.***

e

*Cinderella had a ball.****

têm múltiplos significados que não podem ser distinguidos pela análise sintática ou pela tradução das palavras uma a uma. Pelo contrário, o entendimento dessas sentenças requer a habilidade de compreender o contexto em que elas foram feitas. Em outros casos, o significado real de uma sentença não é o mesmo da tradução literal. Por exemplo,

*Do you know what time it is?*****

muitas vezes significa *Please tell me what time it is*****, ou, se quem fala estiver esperando por muito tempo, pode significar *You are very late*.****

Portanto, desvendar o significado de uma oração em uma linguagem natural exige vários níveis de análise. O primeiro é a **análise sintática**, cujo componente principal é a análise gramatical. É nela que o sujeito da oração

*Mary gave John a birthday card******.

*N. de T. Os exemplos foram mantidos em inglês, visto que o texto muitas vezes alude justamente às peculiaridades desta língua, ao ilustrar as características das linguagens naturais.

**N. de T. Em português, *Norman Rockwell pintava retratos de pessoas/pintava pessoas*.

***N. de T. Em português, *Cinderela tinha um baile/uma bola*.

****N. de T. Em português, *Você sabe que horas são?*

*****N. de T. Em português, *Por favor, diga-me que horas são.*

*****N. de T. Em português, *Você está muito atrasado.*

*****N. de T. Em português, *Mary deu a John um cartão de aniversário.*

é *Mary*, enquanto o sujeito de

*John got a birthday card from Mary.**

é *John*.

Outro nível de análise é chamado **análise semântica**. Diferentemente da análise sintática, que somente identifica a função gramatical de cada palavra, a análise semântica se encarrega da tarefa de identificar a função semântica de cada palavra na oração. A análise semântica busca identificar elementos como a ação descrita, o agente desta ação (que pode ou não ser o sujeito da oração) e o objeto da ação. É por meio da análise semântica que as orações *Mary gave John a birthday card* e *John got a birthday card from Mary* devem ser reconhecidas com o mesmo significado.

Um terceiro nível de análise é a **análise contextual**, no qual o contexto da oração é trazido para o processo da compreensão. Por exemplo, é fácil identificar a função gramatical de cada palavra da seguinte oração:

*The bat flew from his hand.***

Podemos até mesmo executar a análise semântica identificando a ação como voar, o agente como o taco e assim por diante. Contudo, somente depois de considerar o contexto da oração é que o seu significado se torna claro. Realmente, apresenta um significado diferente no contexto de um jogador de beisebol e no de um explorador de cavernas. E é por meio da análise de contexto que o verdadeiro significado da pergunta *Do you know what time it is?* seria finalmente desvendado.

Devemos notar que os vários níveis de análise — sintático, semântico e contextual — não são necessariamente independentes. O sujeito da oração

*Stampeding cattle can be dangerous.****

é o substantivo *cattle* (modificado pelo adjetivo *stampeding*) se visualizarmos o próprio gado estourando. Entretanto, o sujeito seria o gerúndio *stampeding* (com objeto *cattle*) no contexto de um encenqueiro cujo entretenimento consista em iniciar um estouro. Assim, a oração possui mais de uma estrutura gramatical — qual delas é a correta depende do significado.

Outra área de pesquisa no processamento de linguagens naturais considera um documento completo, em vez de sentenças individuais. Aqui, os problemas a considerar caem em duas categorias: **recuperação e extração de informação**. A recuperação de informação se refere à tarefa de identificar documentos relacionados com o tópico em questão. Um exemplo é o problema enfrentado por promotores quando tentam encontrar toda a documentação histórica referente a um litígio em andamento. Outro exemplo é o problema enfrentado pelos usuários da *World Wide Web* quando tentam encontrar os sítios relacionados com um tópico específico.

A extração de informação se refere à tarefa de, a partir de documentos, resgatar informação em um formato que seja útil para outras aplicações. Isso pode significar efetuar a identificação da resposta a uma pergunta específica, ou armazenar a informação em um formato com o qual perguntas possam ser respondidas em ocasiões futuras. Um destes formatos é conhecido como *gabarito*****. É essencialmente um formulário preenchido com a informação específica de cada caso. Por exemplo, consideremos um sistema para leitura de jornal. O sistema pode fazer uso de diversos gabaritos, um para cada tipo de artigo existente. Se o sistema localizar um artigo sobre um roubo, preencherá os campos apropriados de um gabarito sobre roubos. Este gabarito provavelmente solicitaria elementos como o endereço onde

*N. de T. Em português, *John ganhou de Mary um cartão de aniversário*.

**N. de T. Nesta frase, *bat* pode ser traduzido como taco de beisebol ou como morcego. As duas traduções possíveis seriam: O (taco/morcego) voou de sua mão.

***N. de T. Nesta frase, *stampeding cattle* pode ser traduzida como estouro de gado ou como estourar o gado. As duas possíveis traduções seriam: Um estouro de/Estourar gado pode ser perigoso.

****N. de T. Em inglês, *template*, às vezes é chamado molde, ou ainda padrão.

ocorreu o roubo, o período e a data, os objetos roubados e assim por diante. No entanto, se o sistema identificar um artigo sobre uma catástrofe natural, o preenchimento será feito segundo um gabarito de catástrofes naturais, identificando o tipo de desastre, o nível de danos e assim por diante.

Outra forma de os extratores de informação armazenarem a informação é denominada **rede semântica**. Trata-se essencialmente de uma grande estrutura de dados ligados em que são utilizados ponteiros para indicar associações entre eles. A Figura 10.27 apresenta parte de uma rede semântica na qual está delimitada, com uma linha tracejada, a informação obtida da oração

Mary hit John^{}.*

O desenvolvimento de computadores que possam entender uma linguagem natural tornou-se uma importante área de pesquisa na inteligência artificial. Também é uma área que demonstra o quanto a pesquisa na inteligência artificial pode ser desafiadora. À medida que os pesquisadores resolvem alguns problemas, outros aparecem.

Robótica

A maioria das pesquisas em robótica moderna era apoiada por interesse comercial — a meta era desenvolver robôs para linhas de produção economicamente viáveis que aumentassem a produtividade e a uniformidade. (Uma delas é reminiscente dos teares de Jacquard, que redefiniu a indústria de vestuário em meados de 1800.) Em consequência, a robótica como disciplina era dominada pelos campos mais tradicionais das engenharias mecânica e elétrica. O desenvolvimento de dispositivos como pinça de precisão e dobradiças altamente flexíveis era freqüentemente colocado a par com o desenvolvimento da inteligência subjacente do dispositivo tinha prioridade em relação ao mesmo.

Tais prioridades eram chamativas comercialmente. Em um ambiente de linha de produção, uma máquina freqüentemente é solicitada a repetir sucessivamente uma tarefa, de modo idêntico. Se a tarefa consistir em apanhar objetos e colocá-los em caixas, os objetos chegam a uma esteira rolante em intervalos regulares e com uma orientação uniforme, e as caixas cheias são constantemente substituídas por outras vazias, na mesma posição. A máquina não apanha, de fato, um objeto, mas somente fecha uma pinça em um dado momento e posição e movimenta o seu braço para outra posição, onde, em vez de colocar o elemento em uma caixa, apenas abre a pinça. Nesse ambiente, então, robôs úteis poderiam ser desenvolvidos sem que primeiro fossem resolvidos problemas complexos que envolvessem inteligência.

Recursão em linguagem natural

As estruturas recursivas que envolvem sentenças dentro de sentenças são comuns em inglês (bem como em outras línguas), onde a sentença mais interna é chamada cláusula. De fato, as técnicas empregadas para tratar tais estruturas foram o primeiro foco de pesquisa nos sistemas computadorizados de processamento de linguagens naturais.

Algumas vezes essas estruturas envolvem vários níveis de recursão que obscurecem o significado da sentença, embora a estrutura geral esteja gramaticalmente correta. Por exemplo, considere a sentença

*The man the horse that lost the race threw was not hurt^{**}*

Esta sentença envolve três estruturas — uma dentro da outra. A mais externa é

The man was not hurt.

A próxima sentença interior identifica o homem que caiu do cavalo. Dentro dessa estrutura, está outra que identifica o cavalo como o que perdeu a corrida. As sentenças a seguir envolvem estruturas recursivas ligeiramente diferentes:

The picture the man the woman who lives next door hired hung fell down.

*The new cook the chief who yells a lot hired but who could not sauté was fired.^{***}*

^{*}N. de T. Em português, *Mary bateu em John*.

^{**}N. de T. Em português, *O homem que caiu do cavalo que perdeu a corrida não estava ferido*.

^{***}N. de T. Em português, *O quadro pendurado pelo homem contratado pela mulher que mora na porta ao lado caiu, e O novo cozinheiro contratado pelo chefe que grita à beça foi despedido por não saber preparar batata sauté*.

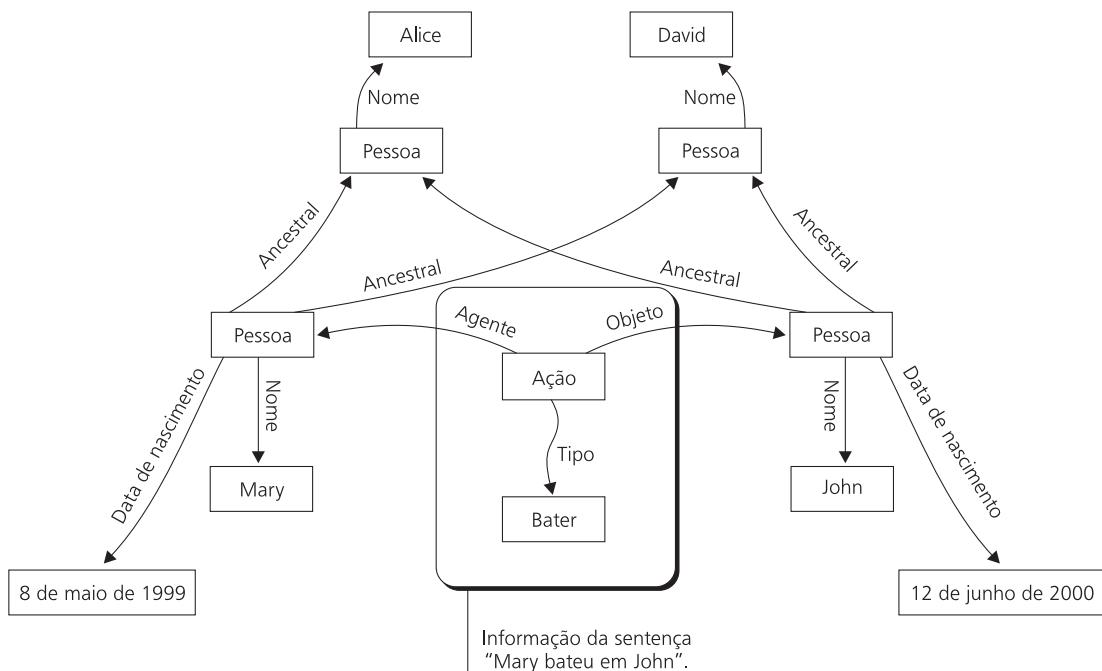


Figura 10.27 Uma rede semântica.

Obviamente, ligeiras modificações na tarefa de apanhar os objetos podem aumentar significativamente as exigências de inteligência para a máquina. Suponhamos que os objetos na esteira rolante cheguem em intervalos irregulares e com orientação também irregular. Então, a tarefa da máquina incluiria detectar os objetos que chegam, reconhecer a sua orientação e desenvolver um plano para apanhar cada um de acordo com a sua orientação. Assim, tópicos menos comerciais e mais científicos dentro do campo da robótica agora têm recebido atenção e apoio. Isso inclui empreendimentos como desenvolver máquinas que se mantêm em equilíbrio, sobem escada e caminham em terrenos acidentados.

Atualmente, um grande segmento de pesquisa na robótica independe de aplicações comerciais diretas. Uma meta primordial nesse segmento é construir robôs autônomos — isto é, que se comportem de modo a garantir a sua sobrevivência. Muitas técnicas criativas e fantasiosas têm sido aplicadas. Uma delas é a aplicação das teorias evolucionárias ao desenvolvimento de robôs, que deram origem ao campo conhecido como robótica evolucionária. Aqui encontramos a teoria da sobrevivência dos melhores sendo usada para desenvolver dispositivos que após muitas gerações adquirem seus próprios meios de equilíbrio e mobilidade. Por sua vez, a disciplina da robótica tem encontrado seu lugar adequado no campo das ciências cognitivas.

Sistemas de banco de dados

A mente humana é um dispositivo maravilhoso. Como ela armazena o seu conhecimento adquirido e mais tarde identifica e extrai itens específicos de informação que são pertinentes à tarefa corrente, continua sendo um mistério — mistério esse que permeia muitas áreas da inteligência artificial. Como a mente humana relembra a informação necessária ao reconhecimento das imagens? Como aplica o contexto adequado ao processar declarações ambíguas em linguagem natural? Como armazena o seu conhecimento do mundo real de modo a poder raciocinar sobre esse conhecimento?

Assim, um problema fundamental da inteligência artificial é de banco de dados, e o progresso em cada campo é aplicado ao outro. As técnicas da inteligência artificial são aplicadas aos sistemas tradicionais de bancos de dados para proporcionar melhores serviços, e as técnicas de bancos de dados, aos projetos de inteligência artificial, em uma tentativa de tratar as enormes massas de conhecimento do mundo real subjacentes aos processos de decisão envolvidos. Em outros casos, os esforços de pesquisa em inteligência artificial e em sistemas de bancos de dados atacam o mesmo problema a partir de perspectivas diferentes.

Um exemplo é o problema de desenvolver os sistemas de memória associativa (sistemas de bancos de dados associativos). Já vimos a necessidade de tais sistemas no campo do processamento de linguagens naturais, onde a informação relevante do mundo real deve ser aplicada para entender sentenças de acordo com seus contextos, e discutimos como as redes neurais artificiais têm sido aplicadas ao problema.

Visto da perspectiva mais tradicional de banco de dados, o problema é identificar e recuperar a informação relacionada a um tópico, em vez da informação que está sendo explicitamente solicitada. Considere, por exemplo, o problema de quem navega na Web procurando informação na rede. A abordagem tradicional é exigir que a pessoa identifique palavras-chave e frases que devem aparecer nos documentos relevantes. O sistema então procura em seu banco de dados e recupera os documentos que contêm tais palavras e frases. Infelizmente, esse sistema nada mais é do que uma peneira baseada em sintaxe, e não na semântica. Ele pode desprezar os documentos mais importantes porque lidam com “carros” em vez de “automóveis”. O que o navegante realmente deseja é um sistema de recuperação com inteligência para identificar matérias relacionadas (ou associadas), em vez de simplesmente recuperar o que foi explicitamente solicitado.

Como outro exemplo, suponha que tenhamos um banco de dados com os cursos conduzidos pelos professores de uma universidade, juntamente com as notas dadas aos estudantes. Consideremos a seguinte sequência de eventos: solicitamos ao banco de dados o número de notas A atribuídas pelo professor Johnson no semestre passado. O banco de dados responde “nenhuma”. Concluímos que o professor Johnson era um instrutor muito exigente. A seguir, solicitamos o número de notas F por ele atribuídas no semestre passado. Novamente, o banco de dados responde “nenhuma”. Concluímos que o professor Johnson considera médios todos os estudantes, sem que haja alguém nos extremos. Então solicitamos o número de notas C dadas por ele neste mesmo período. O banco de dados novamente responde “nenhuma”. Neste momento, surge a dúvida, e perguntamos se o professor Johnson lecionou algum curso no semestre passado. O banco de dados responde “não”. Se ao menos tivesse dito isso logo no início! Gostaríamos de ter um banco de dados que recuperasse a informação que necessitamos, em vez da que solicitamos.

Outro tópico perseguido pela pesquisa tanto em banco de dados como na inteligência artificial é o desenvolvimento de sistemas de armazenamento e recuperação de dados que podem fornecer a informação derivada dos dados armazenados, em vez de meramente responder com a informação explicitamente guardada. Em outras palavras, gostaríamos que o banco de dados fosse capaz de raciocinar com a informação nele contida. Considere, por exemplo, um banco de dados que contenha informações sobre os presidentes dos Estados Unidos. Quando consultado sobre a existência de algum presidente com 3 m de altura, um sistema tradicional não seria capaz de fornecer a resposta, a menos que a altura de cada presidente estivesse armazenada no banco de dados. Por outro lado, um sistema inteligente poderia responder corretamente, sem saber a altura de cada um. A linha de raciocínio poderia ser a seguinte: se tivesse existido um presidente com 3 m de altura, isto teria sido importante e, por isso, armazenado no banco de dados. Portanto, desde que nenhum presidente foi registrado com 3 m de altura, então tal presidente nunca existiu.

A conclusão de que não houve presidentes com 3 m de altura envolve um conceito importante no projeto de banco de dados — a diferença entre bancos de dados em universo fechado e bancos de dados em universo aberto. Falando abertamente, um **banco de dados em universo fechado** é o que assume a postura de conter todos os fatos verdadeiros sobre o tópico em questão, enquanto um banco de dados em universo aberto não adota tal hipótese. A capacidade de rejeitar a hipótese de um presidente

com 3 m de altura, no exemplo anterior, estava baseada na suposição, em universo fechado, de que se o fato não estiver armazenado, deve ser falso. Do mesmo modo, um banco de dados com assinantes de revistas pode usar a suposição do universo fechado para concluir que um indivíduo não é assinante de uma revista específica, mesmo sem conter a lista de todos os não-assinantes.

Embora a hipótese do universo fechado pareça ser suficientemente inocente à primeira vista, sua aplicação pode levar a complicações sutis. Considere um banco de dados constituído de uma única proposição:

Mickey é um rato OU Donald é um pato.

Com esta proposição apenas, não podemos concluir que Mickey é de fato um rato. Assim, a hipótese de universo fechado nos força a concluir que a declaração

Mickey é um rato

é falsa. Do mesmo modo, a suposição em universo fechado nos força a concluir que

Donald é um pato

é falsa. Portanto, verificamos que a hipótese de universo fechado nos conduziu à conclusão contraditória de que, embora pelo menos uma das declarações tenha de ser verdadeira, ambas são falsas. Compreender as limitações desta técnica aparentemente inocente de raciocínio é uma meta corrente de pesquisa em inteligência artificial.

Sistemas especialistas

Uma extensão importante do conceito de banco de dados inteligentes é o **sistema especialista** — pacote de *software* projetado para ajudar os seres humanos em situações nas quais é imprescindível a presença de um especialista. Tais sistemas são projetados para simular um raciocínio do tipo causa-e-efeito, que os especialistas realizariam diante das mesmas situações. Assim, um sistema especialista médico proporá o mesmo procedimento que um médico especialista proporia, sabendo que, se for notado o aparecimento de uma anormalidade cuja radiografia acuse a presença de massa em tal região, uma biopsia deverá ser realizada.

Uma das principais tarefas na construção de um sistema especialista é a obtenção do conhecimento necessário fornecido por um especialista. A maneira de resolver este problema se tornou uma importante área de pesquisa. Esse problema é, na verdade, composto de duas tarefas. Uma consiste em obter e manter a cooperação do especialista — algo nada fácil de contornar, dado que o questionário envolvido provavelmente é longo e frustrante, e que ele próprio não deseja transferir conhecimento a um sistema capaz de, em última instância, ocupar o seu lugar. O outro fator crítico é que a maioria dos especialistas nunca se dá conta do processo de raciocínio utilizado para suas conclusões. Questionado sobre o seu procedimento, em geral, a resposta é “não sei”.

Uma vez superados tais problemas, o conhecimento obtido do especialista deve ser organizado em um formato compatível com um sistema de *software*. Esta organização em geral é realizada expressando tal conhecimento por meio de um conjunto de regras na forma de implicações lógicas. Por exemplo, a regra para que uma anormalidade, confirmada por radiografias, venha a solicitar uma biopsia pode ser expressa como:

(verificada a presença de anormalidade e radiografia mostra presença de massa)
implica (execute biopsia)

(Quem leu a seção opcional sobre programação declarativa do Capítulo 5 reconhecerá a semelhança entre a estrutura de um sistema especialista e a de um programa em Prolog. Esta semelhança é uma das razões principais da popularidade do Prolog no campo da inteligência artificial. Realmente, ele é uma excelente linguagem para o desenvolvimento de sistemas especialistas.)

Observe a semelhança entre as regras de um sistema especialista e as produções de um sistema de produções. A primeira parte da regra determina, essencialmente, as precondições para executar ou concluir a segunda parte. De fato, muitos sistemas especialistas são na verdade sistemas de produções, cujas regras, obtidas a partir do especialista humano, são as produções, e o raciocínio subjacente, baseado em tais regras, é simulado pelo sistema de controle. Nesse contexto, o conjunto de produções é denominado base de conhecimento do sistema, e o sistema de controle, máquina de inferência.

Entretanto, é preciso ter cuidado para não imaginar que um sistema especialista seja apenas uma versão ampliada do sistema de resolução de quebra-cabeças anteriormente discutido. Alguns sistemas especialistas são organizados como conjuntos de sistemas de produções, que combinam os seus esforços para a resolução de problemas. Como exemplos, podemos citar os sistemas especialistas baseados no **modelo de quadro-negro**^{*}. Nele, vários sistemas de resolução de problema, denominados fontes de conhecimento, compartilham uma área comum de armazenamento, chamada quadro-negro, que contém o estado corrente do problema a ser resolvido e, por ser compartilhado por todas as fontes de conhecimento, proporciona um meio através do qual as fontes de conhecimento podem contribuir para a solução do problema. Para coordenar as atividades das fontes de conhecimento, há um módulo de controle, encarregado da tarefa de ativar a fonte de conhecimento apropriada no momento adequado. Na terminologia do modelo do quadro-negro, diz-se que este módulo de controle determina o “foco de atenção” do sistema.

Outra diferença entre um sistema especialista e um sistema simples de produções é que o primeiro não está necessariamente encarregado de alcançar uma meta predeterminada, mas, mais provavelmente, encarregado de fornecer conselhos bem fundamentados. Por exemplo, suponha que um sistema especialista esteja encarregado da tarefa de diagnosticar doenças. O ideal seria se ele pudesse concluir com uma proposição definitiva da forma: “A doença é X”, em que X é o nome da doença. Infelizmente, tal precisão pode ser inviável. Em lugar disso, a melhor resposta poderia ser do tipo: “É provável que a doença seja X”, ou talvez “A doença é ou X ou Y. Por favor execute o seguinte teste para determinar qual é a mais provável”. Por causa dessa ambigüidade, o sistema de controle contido em um sistema especialista pode seguir várias trajetórias do grafo de estados do sistema e apresentar os resultados obtidos em cada uma. Realmente, se a produção aplicada em algum desses estados for

(presença do fator reumático e o paciente apresenta dor nas articulações)
implica (80% de probabilidade de ser artrite reumática)

então qualquer outro raciocínio baseado no fato de a doença ser artrite reumática tem potencial de ser inválido.

A exemplo do que ocorre em outras áreas de pesquisa, as primeiras aplicações de sistemas especialistas ficaram restritas somente a algumas áreas. Atualmente, são inúmeras as áreas em que os sistemas especialistas encontram aplicação. Um catalisador para esta expansão foi a conscientização de que um sistema especialista pode ser separado em dois componentes: de raciocínio e de conhecimento. Removendo a base de conhecimento de um sistema especialista existente, tem-se um sistema de rotinas de raciocínio chamado máquina de inferência, que provavelmente pode ser muito bem aplicado a outras situações. Portanto, sistemas especialistas novos em outras áreas podem ser construídos simplesmente anexando uma nova base de conhecimento à máquina de inferência existente. (Quem leu a seção opcional sobre programação declarativa do Capítulo 5 reconhecerá um programa em Prolog como a base de conhecimento, e o sistema Prolog subjacente como a máquina de inferência.)

*N. de T. Em inglês, *blackboard model*.



QUESTÕES/EXERCÍCIOS

1. Identifique as ambigüidades envolvidas na tradução da oração *They are racing horses*.
2. Compare os resultados da análise sintática das duas orações abaixo. Em seguida, explique suas diferenças semânticas.
The farmer built the fence in the field.
The farmer built the fence in the winter.
3. Baseado na rede semântica da Figura 10.27, qual a relação de parentesco entre Mary e John?
4. Um banco de dados sobre assinantes de revista geralmente contém uma lista de assinantes para cada revista, mas não uma lista dos que não são. Assim, como o banco de dados identifica que uma determinada pessoa não assina uma revista?
5. Qual a diferença entre um banco de dados tradicional e uma base de conhecimento para um sistema especialista?

10.7 Considerando as consequências

Sem dúvida, os avanços feitos na inteligência artificial têm potencial para beneficiar a humanidade e é fácil tornar-se presa no entusiasmo gerado por tais benefícios. Contudo, também há perigos em potencial ocultos no futuro, cujas ramificações podem ser tão devastadoras quanto os seus benefícios são efetivos. A distinção freqüentemente é de ponto de vista ou talvez da posição ocupada na sociedade — o ganho de uma pessoa pode ser a perda de outra. É razoável, então, reservar um tempo para olhar o avanço da tecnologia a partir de perspectivas alternativas.

Alguns vêem o avanço da tecnologia como um presente para a humanidade — meio de libertar as pessoas de tarefas tediosas e preventivas e abrir as portas para estilos de vida mais agradáveis. Entretanto, outros vêem este mesmo fenômeno como uma maldição que rouba dos cidadãos o emprego e a possibilidade de enriquecimento em favor dos poderosos. Esta, de fato, era uma mensagem comum do humanitário devotado Mahatma Gandhi, que exerceu uma importante influência na luta da Índia para se libertar da Inglaterra. Ele costumava argumentar que a Índia seria melhor servida substituindo-se as grandes fiações têxteis por rodas de fiar instaladas nas casas dos camponeses. Dessa maneira, dizia ele, a produção em massa, centralizada, que emprega poucos, seria substituída por um sistema distribuído de produção em massa que beneficiaria multidões.

A história está repleta de revoluções cujas raízes estão na distribuição desproporcional das riquezas e privilégios. Se o avanço da tecnologia atual reforçar tais discrepâncias, ele poderá levar a consequências catastróficas.

No entanto, a consequência de se construir máquinas cada vez mais inteligentes é mais sutil — mais fundamental — que as relacionadas à luta pelo poder entre diferentes segmentos da sociedade. Os tópicos esbarram no coração da auto-imagem da humanidade. No século XIX, a sociedade ficou espantada com a teoria da evolução de Charles Darwin e com a idéia de que os seres humanos podem ter evoluído a partir de formas vitais inferiores. Como, então, a sociedade reagirá confrontada com o assalto de máquinas cuja capacidade mental desafia a dos seres humanos?

No passado, a tecnologia se desenvolvia devagar, dando tempo para que a nossa auto-imagem fosse preservada, ajustando o nosso conceito de inteligência. Em um certo sentido, aprendemos a definir a inteligência como “aquilo que as máquinas não podem fazer”. Nossos antepassados distantes interpretariam que os dispositivos mecânicos do século XIX tinham inteligência sobrenatural, mas atualmente não creditamos inteligência alguma àquelas máquinas. Contudo, como a humanidade reagirá se as máquinas desafiarem verdadeiramente a inteligência dos seres humanos ou, mais provavelmente, se a capacidade das máquinas começar a avançar mais rápido do que a nossa capacidade de adaptação?

Você pode argumentar que tais questões delimitam a ficção científica, e não a ciência da computação. Não faz muito tempo, porém, muitos desconsideravam a questão “Como a humanidade reagirá se os computadores dominarem a sociedade?” com a mesma atitude de “isso nunca acontecerá.” Entretanto, em muitos aspectos, esse dia já chegou. Se um banco de dados computadorizado informar equivocadamente que a sua ficha de crédito não é boa, que você possui antecedente criminal, ou que sua conta corrente está sem fundos, será a declaração do computador ou o seu clamor de inocência que prevalecerá? Se um sistema de navegação defeituoso indicar que a pista está coberta por neblina, onde o avião poussará? Se uma máquina for usada para prever a reação pública a várias decisões políticas, que decisão um político tomará? Quem (ou o que) então é responsável? Já não subordinamos a sociedade às máquinas? Nossa nível de conforto na situação atual muda dependendo de interpretarmos que as máquinas que afetam nossas vidas decidiram conscientemente baseadas em inteligência, ou meramente produziram respostas pré-programadas — muito embora as decisões sejam as mesmas?

Podemos ter uma idéia da reação em potencial da humanidade às máquinas que desafiam o nosso intelecto considerando a resposta da sociedade ao teste de QI, ocorrida na metade do século XX. Esses testes eram considerados indicadores do nível de inteligência das crianças. Nos Estados Unidos, as crianças freqüentemente eram classificadas de acordo com seus desempenhos nesses testes e dirigidas para programas educacionais compatíveis com os resultados. Por sua vez, as oportunidades educacionais ficavam abertas às crianças que se saíam bem nesses testes, enquanto as outras eram encaminhadas para programas de recuperação. Resumindo, quando é dada uma escala para medir a inteligência de um indivíduo, a sociedade tende a menosprezar as capacidades daqueles que se encontram na extremidade baixa da escala. O que a sociedade fará então se as capacidades “intelectuais” das máquinas se tornarem compatíveis, ou mesmo aparentarem ser compatíveis com as dos seres humanos? Aqueles cujas habilidades fossem consideradas “inferiores” às de uma máquina seriam banidos pela sociedade? Em caso afirmativo, quais seriam as consequências para tais membros da sociedade? A dignidade de uma pessoa deveria se sujeitar ao modo como ela se compara com uma máquina?

Já começamos a ver as potencialidades intelectuais humanas serem desafiadas por máquinas em alguns campos. Elas agora são capazes de vencer especialistas em xadrez, os sistemas especialistas computadorizados são capazes de fornecer indicações médicas, e carteiras de ações gerenciadas por programas simples freqüentemente têm desempenho superior às gerenciadas por profissionais em investimentos. Como esses sistemas afetam a auto-imagem dos indivíduos envolvidos? Como a auto-estima de um indivíduo será afetada quando ele for ultrapassado por máquinas em mais e mais áreas?

Muitos argumentam que a inteligência das máquinas será sempre inherentemente diferente da possuída pelos seres humanos, uma vez que estes são biológicos e aquelas não. Assim, as máquinas jamais reproduzirão um processo humano de decisão. Elas podem atingir as mesmas decisões dos seres

IA forte versus IA fraca

A conjectura de que as máquinas podem ser programadas para exibir comportamento inteligente é conhecida como **IA Fraca** e é mais ou menos aceita por muita gente atualmente. Contudo, a conjectura de que as máquinas podem ser programadas para possuir inteligência e, de fato, consciência, conhecida como **IA Forte**, é largamente debatida. Os oponentes da IA Forte argumentam que uma máquina é inherentemente diferente de um ser humano e, assim, jamais sentirá amor, discernirá o certo do errado e pensará sobre si mesma da maneira como os homens fazem. Entretanto, os proponentes da IA Forte argumentam que a mente humana é constituída por pequenos componentes que individualmente não são humanos nem conscientes, mas, combinados, são. Por que, argumentam eles, o mesmo fenômeno não seria possível com as máquinas?

O problema na resolução do debate sobre a IA Forte é que, como observado no texto, coisas como inteligência e consciência são características internas que não podem ser diretamente identificadas. Como assinalou Alan Turing, creditamos inteligência às pessoas porque elas se comportam intelligentemente — embora não possamos observar os seus estados mentais interiores. Estamos preparados, então, para conceder a mesma qualidade a uma máquina se ela exibir as características externas de consciência? Por que, ou por que não?

humanos, mas estas não serão tomadas com a mesma base das tomadas por seres humanos. Até que ponto, então, existem diferentes espécies de inteligência, e seria ético para a sociedade seguir os caminhos propostos por inteligência não-humana?

Encerramos com a citação do livro de Joseph Weizenbaum *Computer Power and Human Reason*, no qual ele argumenta contra a aplicação desenfreada da tecnologia da computação.

Os computadores podem tomar decisões judiciais, fazer avaliações psiquiátricas. Podem jogar moedas de uma maneira mais sofisticada do que qualquer paciente ser humano. O ponto é que não devem ser dadas a eles tais tarefas. Eles até são capazes de chegar às decisões “corretas” em alguns casos — mas sempre e necessariamente em bases que nenhum ser humano deve consentir em aceitar.

Tem havido muito debate sobre “Computadores e Mente”. O que concluo aqui é que os tópicos relevantes não são tecnológicos nem mesmo matemáticos, eles são éticos. Não podem ser postos com perguntas iniciadas por “pode”. Os limites da aplicação dos computadores são, em última instância, estabelecidos em termos de deveres. O que emerge como conclusão mais elementar é que, por não termos meios, agora, de tornar sensatos os computadores, não devemos lhes dar tarefas que demandam sabedoria.



QUESTÕES/EXERCÍCIOS

1. Quanto da população atual sobreviveria se as máquinas desenvolvidas nos últimos 100 anos desaparecessem? E nos últimos 50 anos? E nos últimos 20 anos? Onde os sobreviventes estariam localizados?
2. Até que ponto a sua vida é controlada por máquinas? Quem controla as máquinas que afetam a sua vida?
3. Onde você obtém a informação na qual se baseia para tomar as decisões diárias? E as decisões mais importantes? Que confiança você tem na exatidão dessa informação? Por quê?

Problemas de revisão de capítulo

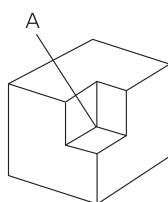
(Os problemas marcados com asterisco se referem às seções opcionais.)

1. Às vezes, a capacidade de responder a uma pergunta depende tanto de saber os limites do conhecimento como dos fatos propriamente ditos. Por exemplo, suponhamos que os bancos de dados A e B contenham uma lista completa de funcionários filiados ao programa de seguro de saúde de uma empresa, mas somente o banco de dados A tem certeza de que sua lista está completa. A que conclusão o banco de dados A poderia chegar, mas não o banco de dados B, acerca de um sócio que não figure em sua lista?
2. Acerca do texto em que discutimos os problemas de compreensão de linguagens naturais, em oposição às linguagens de programação formal, como exemplo das complexidades envolvidas em linguagens naturais, identifique as situações em que *Do you know what time it is?* apresenta significados diferentes a pergunta.
3. Conforme mostrado no Problema 2, os seres humanos podem fazer uma pergunta com outro propósito que não o de realmente formular uma questão. Outro exemplo é *Do you know that your tire is flat?*^{*}, que é utilizada para informar, e não para perguntar. Dê outros exemplos de perguntas utilizadas no intuito de confirmar, advertir e criticar.
4. Compare as funções das frases preposicionais nas duas seguintes orações (que diferem somente em uma palavra):
The pigpen was built by the barn.
The pigpen was built by the farmer.^{**}

^{*}N. de T. Em português, *Você sabe que o seu pneu está murcho?*

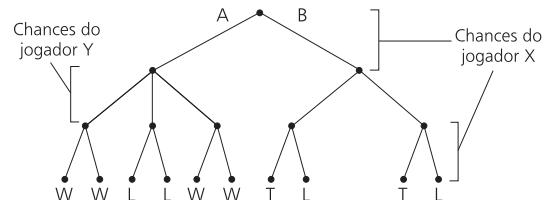
^{**}N. de T. Em português, as duas frases: *O chiqueiro foi construído (perto do celeiro/pelo fazendeiro).*

5. Se um pesquisador utilizar modelos computacionais para estudar as capacidades e os processos de memorização da mente humana, os programas desenvolvidos para a máquina necessariamente realizarão o trabalho de memorização no limite das suas possibilidades? Explique.
6. Quais das seguintes atividades você espera que sejam orientadas ao desempenho e quais à simulação?
- o projeto de um simulador de vôo
 - o projeto de um sistema de piloto automático
 - o projeto de um banco de dados que lida com materiais de biblioteca
 - o projeto de um modelo econômico de uma nação para testar teorias
 - o projeto de um programa para monitorar os sinais vitais de um paciente
7. Identifique um pequeno conjunto de propriedades geométricas a serem usadas para distinguir entre os símbolos O, G, C e Q.
8. Descreva as semelhanças entre a técnica de identificar características, comparando-as com padrões, e os códigos de correção de erros discutidos no Capítulo 1.
9. Descreva duas interpretações do seguinte desenho, com base em se o “vértice” A é convexo ou côncavo:



10. No contexto de um sistema de produções, qual a diferença entre um grafo de estados e uma árvore de busca?
11. Analise a tarefa de resolver o cubo de Rubik como um sistema de produções. (Quais são os estados, as produções etc?) Que heurística poderia ser usada para ajudar o sistema de controle a tomar decisões?
12. Analise a tarefa de investir no mercado de capitais como um sistema de produções. (Quais são os estados, as produções etc?) Que heurística poderia ser usada para ajudar o sistema de controle a tomar decisões?

13. Analise a tarefa de jogar uma partida de golfe como um sistema de produções. (Quais são os estados, as produções etc?) Que heurística poderia ser usada para ajudar na decisão de que produção aplicar?
14. No texto, mencionamos que um sistema de produções geralmente é usado como técnica para tirar conclusões a partir de fatos conhecidos. Os estados do sistema são fatos tidos como verdadeiros em cada fase do processo de raciocínio, e as produções, as regras lógicas destinadas a manipular tais fatos. Identifique algumas regras lógicas que levam à seguinte conclusão: *John is tall** a partir dos fatos: *John is a basketball player*, *Basketball players are not short* e *John is either short or tall*.**
15. A seguinte árvore representa os possíveis movimentos em um jogo competitivo, mostrando que o jogador X pode, no momento, escolher entre os movimentos A e B. Após o movimento do jogador X, o jogador Y tem direito a um movimento e, em seguida, o jogador X tem direito ao último movimento do jogo. Os nós-folha da árvore estão etiquetados com W, L ou T, conforme o resultado obtido pelo jogador X seja ganho, perda ou empate, respectivamente. O jogador X deveria escolher o movimento A ou B? Por quê? Em que sentido a tarefa de selecionar uma “produção” em uma atmosfera competitiva difere da de um jogo individual, como é o caso do quebra-cabeças de oito peças?



16. Analise o jogo de damas como um sistema de produções e descreva uma heurística que poderia ser utilizada para determinar qual de dois estados está mais próximo do objetivo. Como o sistema de controle nessa situação difere de um

*N. de T. Em português, *John é alto*.

**N. de T. Em português, *John é um jogador de basquete*, *Jogadores de basquete não são baixos*, *John é baixo ou é alto*.

outro aplicado a um jogo individual, como o quebra-cabeças de oito peças?

17. Considerando as regras da álgebra como produções, problemas que envolvem a simplificação de expressões algébricas podem ser resolvidos no ambiente de um sistema de produções. Identifique um conjunto de produções algébricas que reduzam a equação $3/(2x + 1) = 2/(2x - 2)$ para a forma $x = 4$. Cite alguns palpites (isto é, algumas regras heurísticas) aplicados ao executar essas simplificações algébricas.
18. Desenhe a árvore de busca gerada pela busca em largura, em uma tentativa de resolver o quebra-cabeças de oito peças a partir do seguinte estado inicial, sem o auxílio de qualquer informação heurística.

	1	3
4	2	5
7	8	6

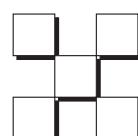
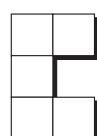
19. Desenhe a árvore de busca gerada pelo algoritmo da Figura 10.9, em uma tentativa de resolver o quebra-cabeças de oito peças com o estado inicial do Problema 18. Utilize como heurística o número de peças fora de lugar.
20. Desenhe a árvore de busca gerada pelo algoritmo da Figura 10.9, em uma tentativa de resolver o quebra-cabeças de oito peças com o estado inicial abaixo, entendendo-se que a heurística utilizada seja a mesma desenvolvida na Seção 10.3.

1	2	3
5	7	6
4		8

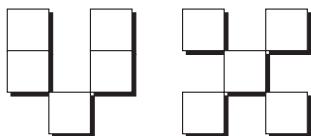
21. Quando se resolve o quebra-cabeças de oito peças, por que o número de peças fora de posição não é uma heurística tão boa quanto a usada na Seção 10.3?
22. Qual a diferença entre a técnica de decisão de qual metade considerar quando se aplica a busca binária sobre uma lista (Seção 4.5) e a decisão de que ramo seguir quando se faz uma busca heurística?

23. Observe que, se um estado de um grafo de estados de um sistema de produções apresentar um valor heurístico extremamente baixo em comparação com os outros estados, e se houver uma produção que defina a transição deste estado para si mesmo, o algoritmo da Figura 10.9 poderá entrar em um ciclo infinito, se este estado suceder a si próprio indefinidamente. Mostre que, se o custo de execução de qualquer produção do sistema for pelo menos igual a um, então, estipulando que o custo projetado seja a soma do valor heurístico com o custo necessário para atingir esse estado ao longo da trajetória percorrida, esta repetição interminável seria evitada.

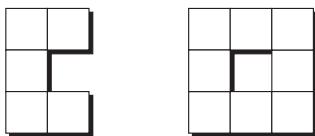
24. Qual heurística você adotaria para escolher uma rota entre duas cidades, em um grande mapa de estradas?
25. Liste duas propriedades que uma heurística deve apresentar para ser útil em um sistema de produções.
26. Suponha que você tenha dois balde. Um deles tem capacidade de 3 litros e o outro, de 5. Você pode despejar a água de um balde no outro, esvaziar ou encher um deles a qualquer momento. Seu problema consiste em colocar, exatamente, 4 litros de água no balde de 5 litros. Formule este problema como um sistema de produções.
27. Suponha que o seu trabalho seja o de supervisionar o carregamento de dois caminhões, sendo que cada um carrega no máximo 14 toneladas. A carga consiste em vários engradados cujo peso total é de 28 toneladas, mas cujos pesos individuais variam de um para outro. O peso de cada um está marcado em sua lateral. Qual heurística você adotaria para distribuir os engradados entre os dois caminhões?
28. Projete uma rede neural artificial capaz de identificar qual dos padrões está no seu campo de visão.



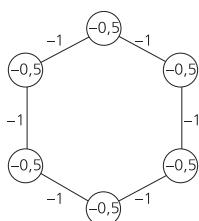
- 29.** Projete uma rede neural artificial capaz de identificar qual dos padrões está no seu campo de visão.



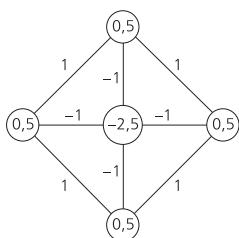
- 30.** Projete uma rede neural artificial capaz de identificar qual dos padrões a seguir está no seu campo de visão.



- 31.** O diagrama abaixo representa uma rede neural artificial para uma memória associativa, como discutido no fim da Seção 10.4. Que padrão ela associa com qualquer padrão no qual apenas duas unidades separadas por uma única estejam excitadas? O que acontecerá se a rede for iniciada com todas as suas unidades inibidas?



- 32.** O diagrama abaixo representa uma rede neural artificial para uma memória associativa, como discutido no fim da Seção 10.4. Que configuração estável ela associa com qualquer padrão inicial no qual ao menos três unidades no perímetro estejam excitadas e a unidade central, inibida? O que acontecerá se lhe for dado um padrão inicial no qual apenas duas unidades opostas no perímetro estejam excitadas?



- 33.** Projete uma rede neural artificial para uma memória associativa (como discutido no fim da Seção 10.4) que consista em um arranjo retangular de unidades de processamento que tente se mover para padrões estáveis nos quais uma única coluna vertical de unidades fica excitada.

- 34.** Em que os resultados da análise sintática das duas orações abaixo diferem? E os resultados da sua análise semântica?

Theodore rode the zebra.

*The zebra was ridden by Theodore.**

- 35.** Em que os resultados da análise sintática das duas orações seguintes diferem? E os resultados da sua análise semântica?

Se $X = 5$ então some 1 a X senão subtraia 1 de X.

Se $X \neq 5$ então subtraia 1 de X senão some 1 a X.

- 36.** Dê um exemplo em que a hipótese de universo fechado conduza a uma contradição.

- 37.** Dê dois exemplos em que a hipótese de universo fechado seja comumente empregada.

- 38.** Ajuste os pesos e os valores de referência na rede neural artificial da Figura 10.17, de forma que sua saída seja 1 quando as duas entradas forem iguais (ambas forem 0 ou 1) e 0 quando diferentes (uma delas for 0 e a outra, 1).

- 39.** Faça um diagrama semelhante ao da Figura 10.4, representando o processo de simplificação da expressão algébrica $7x + 3 = 3x - 5$ para a expressão $x = -2$.

- 40.** Estenda a sua resposta do problema anterior, mostrando outras trajetórias que um sistema de controle poderia seguir ao tentar resolver o problema.

- 41.** Faça um diagrama semelhante ao da Figura 10.4 para representar o processo de raciocínio envolvido na conclusão *Polly can fly*** a partir dos seguintes fatos iniciais: *Polly is a parrot*, *A parrot is a bird* e *All birds can fly****.

- 42.** Em oposição à proposição do problema anterior, alguns pássaros não podem voar, como é o caso

*N. de T. Em português, *Theodore montou na zebra*, e *A zebra foi montada por Theodore*.

**N. de T. Em português, *Polly pode voar*.

***N. de T. Em português, *Polly é um papagaio*, *Um papagaio é um pássaro*, *Todos os pássaros podem voar*.

do avestruz, ou de um pisco-de-peito-ruivo com uma asa quebrada. De fato, não seria razoável construir um sistema de raciocínio dedutivo em que sejam explicitamente listadas todas as exceções à proposição *All birds can fly*. Assim, como os seres humanos decidem se um dado pássaro pode ou não voar?

- 43.** Explique como a semântica da oração *I read the new refrigerator warranty** depende do contexto.
- 44.** Descreva como o problema de viajar de uma cidade para outra pode ser enquadrado como um sistema de produções. Quais são os estados? Quais são as produções?
- 45.** Suponha que você deva executar três tarefas, A, B e C, que podem ser realizadas em qualquer ordem (mas não ao mesmo tempo). Formule seu problema por meio de um sistema de produções e faça o seu grafo de estados.
- 46.** Em que o grafo de estados do problema anterior se alteraria se a tarefa C desse ser executada antes da A?
- 47.** Descreva como uma estratégia para o jogo da velha poderia ser desenvolvida usando a abordagem dos algoritmos genéticos.
- 48.** O método de cruzar estratégias descrito na Seção 10.5 é conhecido como cruzamento em um único ponto. Uma alternativa é o cruzamento em dois pontos, na qual os segmentos do “meio” de duas estratégias são trocados. Mostre que qualquer estratégia construída usando o cruzamento em um único

ponto também pode ser construída usando cruzamento em dois pontos e vice-versa.

- 49.**
 - a. Se a notação (i, j) , onde i e j são inteiros positivos, é usada para significar “se o elemento na posição I na lista for maior que o elemento na posição J, então troque esses dois elementos”, qual das seguintes sequências faz o melhor trabalho de ordenar uma lista com três elementos?
(1, 3) (3, 2) (1, 2) (2, 3) (1, 2)
 - b. Use esse sistema notacional para o desenvolvimento de um algoritmo genético visando a construção de um programa de ordenação de listas com dez elementos.
- 50.** Discutimos no texto como os algoritmos genéticos poderiam ser aplicados a populações representadas por listas. Descreva como seriam aplicados a populações representadas por estruturas de árvore.
- 51.** Mudanças no contexto de uma oração podem afetar a importância da mesma, assim como seu significado. No contexto da Figura 10.27, como a importância da oração *Mary hit John* seria alterada se suas datas de nascimento fossem nos anos 1960? E como seria se um deles tivesse nascido nos anos 1960 e o outro nos anos 1990?
- 52.** Faça uma rede semântica que represente a informação contida no seguinte parágrafo:
*Donna threw the ball to Jack, who hit it into center field. The center fielder tried to catch it, but it bounced off the wall instead***.

*N. de T. Em português, *Eu leio a garantia da nova geladeira*.

**N. de T. Em português, *Donna atirou a bola para Jack, que a rebateu para o meio do campo. O jogador do meio do campo tentou pegá-la, mas ela acabou saltando por cima do muro*.

Questões sociais

As seguintes questões procuram auxiliar o leitor no entendimento de alguns assuntos éticos, sociais e legais no campo da computação. O objetivo não é meramente o de fornecer respostas a tais questões. O leitor também deve justificá-las e verificar se as justificativas apresentadas preservam sua consistência de uma questão para outra.

1. Até que ponto os pesquisadores de energia nuclear, engenharia genética e inteligência artificial devem ser considerados responsáveis pela forma de utilização dos resultados dos seus trabalhos? Um cientista é responsável pelo conhecimento revelado por sua pesquisa? O que acontece se o conhecimento resultante tem consequência inesperada?
2. Como você distinguiria entre inteligência e inteligência simulada? Você acredita que haja diferença?
3. Suponha que um sistema especialista médico computadorizado ganhe uma reputação na comunidade médica por dar conselhos certos. Até que ponto um médico deveria permitir que tal sistema venha a alterar as suas decisões relativas ao tratamento de pacientes? Se o médico receitar um tratamento diferente do proposto pelo sistema especialista e os conselhos do sistema estiverem corretos, o médico será culpado de má conduta profissional? Em geral, se um sistema especialista se torna bem conhecido em um certo campo, até que nível ele pode obstruir, em vez de aumentar, a capacidade dos especialistas humanos de fazer os seus próprios julgamentos?
4. Muitos argumentariam que as ações de um computador são somente consequências do modo como foram programados e que, portanto, um computador não pode ter livre arbítrio. Como consequência, não pode ser responsabilizado por suas ações. A mente humana é um computador? Os seres humanos nascem pré-programados? Os seres humanos são programados pelo ambiente em que vivem? Os seres humanos são responsáveis por suas ações?
5. Há caminhos que a ciência não deveria percorrer, mesmo que tenha capacidade para fazê-lo? Por exemplo, se for possível construir uma máquina com recursos de percepção e raciocínio comparáveis aos dos seres humanos, a construção seria apropriada? Que questões a existência dessa máquina poderia levantar? Que questões têm sido levantadas atualmente pelos avanços em outras áreas científicas?
6. A História da humanidade tem sido recheada de casos nos quais os trabalhos de cientistas e artistas foram afetados por influências políticas, religiosas ou outras de cunho social, relativas às suas respectivas épocas. De que maneira tais fatores estão afetando os esforços científicos recentes? E quanto à Ciência da Computação em particular?
7. Muitas culturas atualmente assumem ao menos alguma responsabilidade no sentido de ajudar a reciclar as pessoas cujos trabalhos ficaram redundantes com o avanço da tecnologia. O que a sociedade deve/pode fazer, uma vez que a tecnologia vem tornando as nossas capacidades cada vez mais redundantes?
8. Suponha que você receba uma conta de \$0,00 gerada por um computador. O que deve fazer? Suponha que você nada faça e 30 dias após receba uma segunda notificação de \$0,00 em débito e, junto com ela, uma advertência de que se o pagamento não for feito prontamente, ações legais serão tomadas. Quem deve agir?
9. Existem momentos em que você associa personalidades com o seu computador pessoal? Há momentos em que ele parece vingativo ou teimoso? Você já ficou com raiva do seu computador? Qual é a diferença entre ficar com raiva do computador e ficar enlouquecido *como resultado de*

usar o computador? Seu computador já ficou com raiva de você? Você tem relação similar com outros objetos, como carros, televisões ou canetas esferográficas?

10. Baseado em suas respostas da Questão 9, até que ponto os seres humanos são inclinados a associar o comportamento de uma entidade com a presença de inteligência e consciência? Até que ponto os seres humanos devem fazer tais associações? É possível, para uma entidade inteligente, revelar a sua inteligência por outro meio que não seja o seu comportamento?
11. Muitos acham que a capacidade de passar no teste de Turing não implica que uma máquina seja inteligente. Um argumento é que o comportamento inteligente por si só não implica inteligência. Entretanto, a teoria da evolução se baseia na sobrevivência dos mais capazes, ou seja, em um teste baseado no comportamento. A teoria da evolução implica que o comportamento inteligente precede a inteligência? A capacidade de passar no teste de Turing implica que as máquinas, a seu modo, estariam se tornando inteligentes?

Leituras adicionais

- Banzhaf, W., P. Nordin, R. E. Deller, and E. D. Francone. *Genetic Programming: An Introduction*. San Francisco, CA: Morgan Kaufmann, 1998.
- Luger, G. F. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, 4th ed. Boston: Addison-Wesley, 2002.
- Mitchell, M. *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press, 1998.
- Mitchell, T. M. *Machine Learning*. New York: McGraw-Hill, 1997.
- Nilsson, N. *Artificial Intelligence: A New Synthesis*. San Francisco, CA: Morgan Kaufmann, 1998.
- Rumelhart, D. E. and J. L. McClelland. *Parallel Distributed Processing*. Cambridge, MA: MIT Press, 1986.
- Russell, S. and P. Norvig. *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice-Hall, 1995.
- Shapiro, L. G. and G. C. Stockman. *Computer Vision*. Englewood Cliffs, NJ: Prentice-Hall, 2001.
- Tanimoto, S. L. *The Elements of Artificial Intelligence Using Common Lisp*, 2nd ed. New York: Computer Science Press, 1995.
- Weizenbaum, J. *Computer Power and Human Reason*. New York: W. H. Freeman, 1979.

Teoria da computação

Neste capítulo, consideraremos algumas questões relativas ao quê os computadores podem ou não fazer. Veremos como máquinas simples, conhecidas como máquinas de Turing, são usadas para identificar a fronteira entre os problemas que podem ser resolvidos por máquinas e os que não podem. Identificaremos um problema específico, conhecido como o problema da parada, cuja solução fica além das potencialidades dos sistemas algorítmicos, portanto, além das capacidades dos computadores atuais e dos futuros. Além disso, veremos que mesmo entre os problemas passíveis de solução por máquina, existem aqueles cujas soluções são tão complexas que são considerados insolúveis sob qualquer ponto de vista prático. Encerramos considerando como esse entendimento da complexidade pode ser usado para construir um sistema de criptografia de chave pública.

11.1 Funções e sua computação

11.2 Máquinas de Turing

Fundamentos da máquina de Turing
A tese de Church-Turing

11.3 Linguagens de programação universais

A linguagem Bare Bones
Programação em Bare Bones
A universalidade da Bare Bones

11.4 Uma função incomputável

O problema da parada
A insolubilidade do problema da parada

11.5 Complexidade de problemas

Medição da complexidade de um problema
Problemas polinomiais *versus* não-polinomiais
Problemas NP

11.6 Criptografia de chave pública

Criptografia via problemas da mochila
Aritmética modular
De volta à criptografia

11.1 Funções e sua computação

O objetivo deste capítulo é investigar as capacidades dos computadores. Queremos entender o que as máquinas podem ou não fazer e as características exigidas para que elas alcancem o seu potencial pleno. Iniciamos com o conceito de funções computáveis.

Uma **função**, no sentido matemático, é uma correspondência entre uma coleção de valores possíveis de entrada e uma coleção de valores de saída, de forma que a cada entrada possível é atribuída uma única saída. Um exemplo é a função que converte medidas em jardas para metros. A cada medida em jardas, a função atribui o valor que resultaria se a mesma distância fosse medida em metros. Outro exemplo, que poderíamos chamar de função ordenação, atribui a cada lista de entrada uma lista de saída cujos elementos são os mesmos da lista de entrada, mas estão dispostos de acordo com uma regra pré-estabelecida. Um outro exemplo é a função de adição, cujas entradas são pares de valores e cujas saídas são valores que representam a soma de cada par de entrada.

O processo de determinar a saída de uma função a partir de sua entrada é chamado *computação da função*. A capacidade da computação de funções é importante porque é assim que podemos resolver problemas. Para resolver um problema de adição, devemos realizar a computação da função de adição; para ordenar uma lista, a da função ordenação. Logo, uma tarefa fundamental da ciência da computação é encontrar técnicas para realizar a computação das funções subjacentes aos problemas que queremos resolver.

Por exemplo, suponha um sistema no qual as entradas e saídas da função sejam predeterminadas e armazenadas em uma tabela. Todas as vezes que a saída da função for solicitada, simplesmente procuraremos na tabela a entrada fornecida, e aí encontraremos a saída desejada. Assim, a computação da função se resume ao processo de procura na tabela. Tais sistemas são convenientes, mas limitados, pois muitas funções não podem ser representadas na forma de tabelas. Um exemplo está registrado na Figura 11.1, na qual tentamos mostrar a função que converte medidas em jardas para metros. Uma vez que não há limite na lista de pares de entrada/saída possíveis, a tabela está fadada a permanecer incompleta.

Uma abordagem mais eficaz para a computação de uma função seria seguir as diretrizes proporcionadas em uma fórmula algébrica, em vez de tentar mostrar todas as possíveis combinações de entrada/saída em uma tabela. Por exemplo, para descrever a função

$$V = P(1+r)n$$

cuja saída é o valor obtido de um investimento original de P com uma taxa de juros anual r durante n anos.

Todavia, a capacidade de expressão das fórmulas algébricas tem suas limitações. Há funções cujas relações de entrada/saída são excessivamente complexas para serem descritas através da manipulação algébrica do valor da entrada da função. Por exemplo, podem ser citadas as funções trigonométricas, como seno e co-seno. Se for solicitado o cálculo do seno de 38 graus, você poderá desenhar o triângulo apropriado, medir os seus lados e calcular a relação desejada — um processo que não tem como ser expresso em termos de manipulações algébricas do valor 38. Sua calculadora de bolso também tem dificuldades para calcular o seno de 38 graus. Na realidade, para isso ela é obrigada a aplicar técnicas matemáticas sofisticadas para que seja obtida uma boa aproximação de seno de 38 graus, que lhe é retornada como resposta.

Vimos então que, à medida que nos deparamos com funções cujas relações de entrada/saída são mais e mais complexas, somos forçados a encontrar técnicas mais complexas para computar tais relações. A questão é se sempre encontraremos um sistema para computar

Jardas (entrada)	Metros (saída)
1	0,9144
2	1,8288
3	2,7432
4	3,6576
5	4,5720
.	.
.	.
.	.

Figura 11.1 Uma tentativa de mostrar a função que converte medidas em jardas para metros.

funções independentemente de suas complexidades. A resposta é não. Um resultado notável da matemática é que existem funções tão complexas a ponto de não existir processo bem definido, passo a passo, para determinar as suas saídas a partir de seus valores de entrada. Assim, a computação dessas funções fica além da capacidade de qualquer sistema algorítmico. Essas funções são denominadas não-computáveis, enquanto as funções cujos valores de saída podem ser determinados algorítmicamente a partir de seus valores de entrada são chamadas *funções computáveis*.

O estudo das funções computáveis e não-computáveis é um empreendimento importante na Ciência da Computação. Uma vez que as máquinas podem apenas realizar tarefas descritas por algoritmos, o estudo das funções computáveis é o estudo da capacidade das máquinas. Se identificarmos as capacidades que nos permitem computar o conjunto completo das funções computáveis e então construirmos máquinas com essas capacidades, teremos certeza que as máquinas construídas serão tão capazes quanto possível. Do mesmo modo, se descobrirmos que a solução para um problema exige a computação de uma função não-computável, concluiremos que a solução desse problema fica além da capacidade das máquinas.

Teoria das funções recursivas

Parece que nada atormenta mais a natureza humana do que a informação de que algo não pode ser feito. Logo que os pesquisadores começaram a identificar problemas insolúveis, no sentido de não possuírem soluções algorítmicas, outros iniciaram o estudo desses problemas para tentar entender a sua complexidade. Atualmente, esse campo de pesquisa é uma parte importante do assunto conhecido como teoria das funções recursivas, e tem-se aprendido muito sobre esses problemas tão difíceis. De fato, do mesmo modo como os matemáticos desenvolveram sistemas de numeração que revelam níveis “quantitativos” além do infinito, os teóricos das funções recursivas vêm descobrindo múltiplos níveis de complexidade dentro dos problemas que permanecem além das capacidades dos algoritmos.



QUESTÕES/EXERCÍCIOS

1. Identifique outras funções cuja saída possa ser descrita por meio de uma expressão algébrica que envolva seus valores de entrada.
2. Identifique uma função que não possa ser descrita por meio de uma fórmula algébrica. Apesar disso, essa função é computável?

11.2 Máquinas de Turing

Lembre-se de que para uma máquina realizar uma tarefa, precisamos primeiro encontrar um algoritmo que a realize. Assim, uma vez que não existe algoritmo para funções não-computáveis, essas funções permanecem além das potencialidades dos computadores atuais e futuros. Em um esforço para entender tais limitações das máquinas, muitos pesquisadores propuseram e estudaram diversos dispositivos computacionais. Um deles é a máquina de Turing, proposta por Alan M. Turing em 1936 e ainda hoje utilizada como ferramenta para estudar o potencial dos processos algorítmicos.

Fundamentos da máquina de Turing

Uma **máquina de Turing** consiste em uma unidade de controle, que pode ler e escrever símbolos em uma fita por meio de um cabeçote de leitura e gravação (Figura 11.2). A fita estende-se indefinidamente nas duas extremidades e é dividida em células, sendo que cada uma pode conter qualquer elemento de um conjunto finito de símbolos. Este conjunto é chamado alfabeto da máquina.

A qualquer momento durante o seu processamento, uma máquina de Turing deve estar em uma das condições pertencentes a um conjunto finito chamadas estados. O processamento, em uma máquina

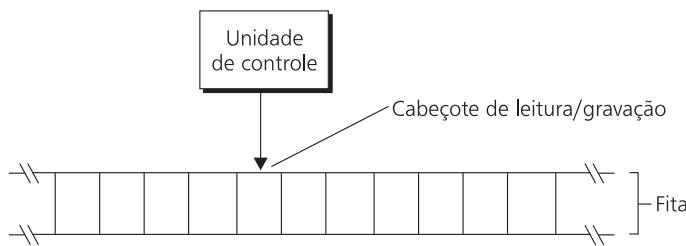
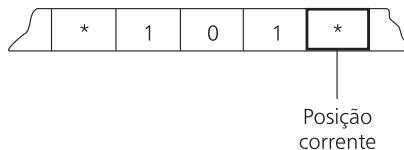


Figura 11.2 Os componentes de uma máquina de Turing.

bolo nesta célula, provavelmente movendo o cabeçote de leitura e gravação para uma célula à esquerda ou à direita, e em seguida mudar de estado. A ação exata a ser executada é determinada por um programa que comunica à unidade de controle o que deve ser feito, com base no estado da máquina e no conteúdo da célula corrente.

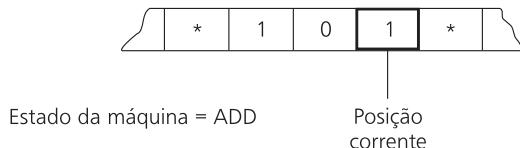
Consideremos agora um exemplo de uma máquina de Turing específica. Para este propósito, representamos a fita da máquina como uma tira horizontal, dividida em células nas quais podemos registrar símbolos do alfabeto da máquina. Indicamos a posição corrente do cabeçote de leitura/gravação colocando um ponteiro apontado para a célula apropriada. O alfabeto para o nosso exemplo consiste nos símbolos 0, 1 e *. A fita da nossa máquina poderia ser da seguinte forma:



Ao interpretar na fita uma cadeia de símbolos que representam números binários separados por asteriscos, reconhecemos que a fita contém o valor 5. A nossa máquina de Turing foi projetada para incrementar tal valor em uma unidade. Mais precisamente, assume-se a posição inicial sobre o asterisco que marca a extremidade direita de uma cadeia de 0s e 1s e efetua-se a alteração do padrão de bits à sua esquerda, de forma que passe a representar o inteiro imediatamente superior.

Os estados da nossa máquina são START, ADD, CARRY, OVERFLOW, RETURN e HALT. As ações correspondentes a cada estado e o conteúdo da célula corrente estão descritos na tabela da Figura 11.3. Pressupõe-se que a máquina comece sempre sua operação no estado START.

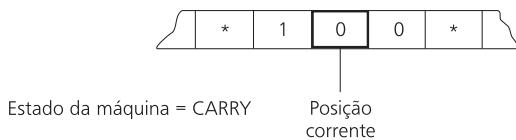
Aplicemos esta máquina à fita anteriormente descrita, que continha o valor 5. Observe que, no estado START, em que a célula corrente contém * (como é o nosso caso), somos instruídos pela tabela a reescrever o *, a mover o cabeçote de leitura/gravação para uma célula à esquerda e a entrar no estado ADD. Após isso ter sido feito, a máquina pode ser descrita como segue:



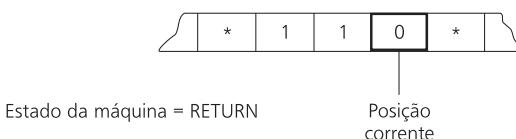
Em seguida, como estamos no estado ADD, em que a célula corrente contém 1, segundo as instruções da tabela, devemos substituir o 1 por 0, mover o cabeçote de leitura/gravação uma célula à esquerda e entrar no estado CARRY. Assim, a configuração da máquina se torna:

de Turing, começa em um estado especial chamado estado inicial e cessa quando a máquina atinge outro estado especial, o estado de parada.

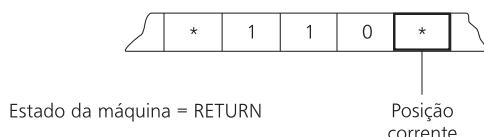
O processamento de uma máquina de Turing consiste em uma seqüência de passos executada pela unidade de controle da máquina. Cada passo consiste em observar o símbolo da célula corrente da fita (aquele que se encontra logo abaixo do cabeçote de leitura e gravação), escrever um sím-



Novamente consultamos a tabela para verificar o que fazer em seguida e encontramos a instrução para, como estamos no estado CARRY, com a célula corrente que contém 0, substituir o 0 por 1, mover o cabeçote de leitura/gravação uma célula à esquerda e entrar no estado RETURN. Feito isso, a configuração da máquina será a seguinte:



A partir desta situação, a tabela nos instrui a substituir o 0 da célula corrente por um outro 0, mover o cabeçote de leitura/gravação uma célula à esquerda e permanecer no estado RETURN. Conseqüentemente, a nossa máquina se encontrará agora na seguinte condição:



Neste ponto, observamos que a tabela nos instrui a reescrever o asterisco na célula corrente e entrar no estado de parada HALT. Desta forma, a máquina pára na seguinte configuração (agora os símbolos na fita representam o valor 6, conforme desejado):



Figura 11.3
Uma máquina de Turing para incrementar um valor.

Estado corrente	Conteúdo da célula corrente	Valor a ser registrado	Direção do movimento	Próximo estado
START	*	*	Esquerda	ADD
ADD	0	1	Direita	RETURN
ADD	1	0	Esquerda	CARRY
ADD	*	*	Direita	HALT
CARRY	0	1	Direita	RETURN
CARRY	1	0	Esquerda	CARRY
CARRY	*	1	Esquerda	OVERFLOW
OVERFLOW	*	*	Direita	RETURN
RETURN	0	0	Direita	RETURN
RETURN	1	1	Direita	RETURN
RETURN	*	*	Sem movimento	HALT

As origens das máquinas de Turing

Alan Turing desenvolveu o conceito da máquina de Turing nos anos 1930, bem antes de a tecnologia ser capaz de prover as máquinas que conhecemos atualmente. De fato, a motivação de Turing era o conceito de um ser humano durante a realização de computações com lápis e papel. O objetivo era fornecer um modelo pelo qual os limites dos “processos computacionais” pudessem ser estudados. Isso ocorreu pouco após a publicação, em 1931, de um famoso artigo de Gödel, expondo as limitações dos sistemas computacionais, e um grande esforço de pesquisa estava sendo dirigido ao entendimento dessas limitações. No mesmo ano em que Turing apresentou o seu modelo (1936), Emil Post apresentou outro (agora conhecido como sistema de produções de Post) com a mesma capacidade do modelo de Turing. Como confirmação das descobertas desses antigos pesquisadores, seus modelos de sistemas computacionais são utilizados ainda hoje como ferramentas valiosas em pesquisa na Ciência da Computação.

A tese de Church-Turing

A máquina de Turing no exemplo anterior pode ser utilizada para calcular os valores da função sucessor, que atribui a cada valor de entrada n , inteiro não-negativo, o valor de saída $n + 1$. Precisamos simplesmente colocar o valor de entrada na sua forma binária na fita, deixar que a máquina execute suas operações até parar e ler o valor de saída na fita. Uma função que pode ser computada desta forma por uma máquina de Turing é denominada **Turing-computável**.

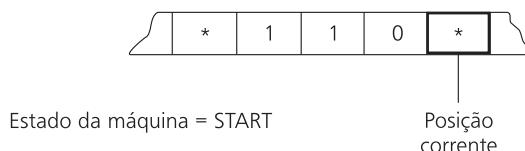
A conjectura de Turing era que o conjunto das funções Turing-computáveis seria igual ao das funções computáveis. Em outras palavras, ele conjecturou que a capacidade computacional das máquinas de Turing abrangeia a de qualquer sistema algorítmico, ou seja, que (diferentemente das abordagens como tabelas ou fórmulas algébricas) o conceito de máquina de Turing proporcionaria um contexto no qual qualquer função computável pudesse ser descrita. Atualmente, esta conjectura freqüentemente é chamada de **tese de Church-Turing**, em referência às contribuições de Alan Turing e Alonzo Church. Desde o trabalho inicial de Turing, muitas evidências foram coletadas para apoiar essa conjectura, e atualmente a tese de Church-Turing é amplamente aceita, ou seja, considera-se verdade atualmente que funções computáveis e funções Turing-computáveis formem um único conjunto.

A importância desta conjectura é que ela proporciona um critério para a avaliação das capacidades e limitações de máquinas computacionais. Mais precisamente, ela estabelece a capacidade das máquinas de Turing como padrão de comparação para a potencialidade de outros sistemas computacionais. Se um sistema computacional for capaz de computar todas as funções Turing-computáveis, será considerado tão potente quanto um sistema computacional pode ser.



QUESTÕES/EXERCÍCIOS

1. Aplique a máquina de Turing descrita nesta seção a partir da seguinte configuração inicial:



2. Descreva uma máquina de Turing que substitua uma cadeia de 0s e 1s por um único 0.
3. Descreva uma máquina de Turing que decremente o valor que se encontra na fita se ele for maior que zero, ou o mantenha inalterado se for zero.
4. Identifique uma situação cotidiana que envolva cálculos. Em que tal situação se assemelha a uma máquina de Turing?
5. Descreva uma máquina de Turing que finalmente pare para algumas entradas, mas não para outras.

11.3 Linguagens de programação universais

No Capítulo 5, estudamos diversas características encontradas nas linguagens de programação de alto nível. Nesta seção, aplicamos o nosso conhecimento de computabilidade para determinar quais dasquelas características são realmente necessárias. Veremos que a maioria das características das linguagens de programação atuais simplesmente melhora a conveniência, em vez de contribuir para o poder fundamental da linguagem.

Nossa abordagem é descrever uma linguagem imperativa simples que seja rica o bastante para permitir a expressão de programas para todas as funções computáveis. Por isso, se um futuro programador constatar que um problema não pode ser resolvido usando a nossa linguagem, então o motivo não será falha na mesma. Uma linguagem de programação com esta propriedade é chamada **linguagem de programação universal**.

Você pode se surpreender com o fato de uma linguagem universal não ser complexa. Sem dúvida, a que vamos apresentar é bem simples. Assim, vamos chamá-la de *Bare Bones*^{*}, no sentido em que ela isola um conjunto mínimo de exigências de uma linguagem de programação universal.

A linguagem Bare Bones

Vamos iniciar a nossa apresentação da Bare Bones considerando as instruções declarativas encontradas em outras linguagens. Essas instruções dão ao programador o luxo de pensar em termos de estruturas de dados e tipos de dados (como matrizes de valores numéricos e cadeias de caracteres alfabéticos), muito embora a máquina, por sua conta, manipule apenas padrões de *bits* sem conhecimento algum do que tais padrões representam. Antes de ser apresentada a uma máquina para execução, uma instrução de alto nível que trabalhe com estruturas e tipos de dados elaborados deve ser traduzida em instruções de máquina que manipulem padrões de *bits* para simular as ações pretendidas. Assim, o projeto de uma linguagem de programação pode ser simplificado, bastando, em primeiro lugar, forçar o programador a expressar todas as operações em termos de padrões de *bits*. Tal linguagem tem um tipo único de dados e de estrutura, de modo que não necessitaria de instruções para descrever os dados.

A título de simplificação, a nossa linguagem Bare Bones adota essa abordagem. Todas as variáveis são do tipo “padrão de *bits* de qualquer tamanho”. Assim, em um programa na linguagem Bare Bones, não precisamos de uma parte declarativa; um programador simplesmente começa a usar um novo nome de variável quando necessário, tendo em mente que ele se refere a um padrão de *bits* de tamanho arbitrário.

Naturalmente, um tradutor da nossa linguagem Bare Bones deve ser capaz de distinguir, de outros elementos, os nomes das variáveis. Isto é realizado projetando a sintaxe de Bare Bones com capacidade de identificar pelo contexto a função de qualquer elemento. Para este propósito, especificamos que os nomes das variáveis sejam compostos somente de letras do alfabeto inglês, que possam ser seguidas por qualquer combinação de letras e dígitos (0 a 9). Assim, as cadeias XYZ, B747, abcdefghi e X5Y podem ser usadas como nomes de variáveis, enquanto 2G5, %6 ou x.y, não.

Como instruções imperativas, a Bare Bones contém três instruções de atribuição e uma estrutura de laço. É uma linguagem de formato livre, de modo que cada

Os alienígenas existem?

Os estudantes raramente se surpreendem ao aprender que existem problemas que não podem ser resolvidos por processos algorítmicos. Contudo, quando solicitados a sugerir candidatos, propõem questões como “A raça humana sobreviverá outros mil anos?” e “Os alienígenas existem?”. A surpresa, contudo, é que estes são exemplos de questões que *podem* ser respondidas de modo algorítmico – e, de fato, por algoritmos muito simples. Cada questão anterior é respondida por um algoritmo que consiste em um único passo: “Producir a resposta sim” ou por outro que consiste em “Producir a resposta não”. Apenas não sabemos atualmente qual dos algoritmos é o correto.

Para esclarecer, considere todos os possíveis programas em Bare Bones dispostos em uma longa lista, começando pelos menores e progredindo com os programas cada vez maiores. Indagar se um problema é passível de solução algorítmica é indagar se existe um programa na lista que o resolva. Isto não é o mesmo que identificar qual programa da lista resolve o problema. Existem muitos problemas que são resolvidos por programas cuja posição na lista ainda não foi determinada. Isto é, existem muitos problemas passíveis de solução algorítmica cujas soluções algorítmicas ainda não foram identificadas. Um exemplo é se os alienígenas existem.

*N. de T. Em português, ao pé da letra, ossos nus, ou seja, esqueleto, em alusão à ausência de supérfluos.

instrução termina com um ponto-e-vírgula, facilitando ao tradutor a separação de instruções que possam aparecer na mesma linha. Todavia, adotaremos a regra de escrever apenas uma instrução em cada linha para melhorar a legibilidade.

Cada uma das três instruções de atribuição faz com que o conteúdo da variável identificada na instrução seja modificado. A primeira nos permite associar uma cadeia de zeros a uma variável. Sua sintaxe é:

```
clear nome;
```

onde *nome* pode ser qualquer nome permitido de variável.

As outras instruções de atribuição são essencialmente inversas uma da outra:

```
incr nome;
```

e

```
decr nome;
```

Novamente, *nome* representa qualquer nome de variável legal. A primeira instrução incrementa o valor associado à variável identificada. Aqui o termo *incrementar* se refere a interpretar os padrões de bits como representações de valores numéricos na notação binária e alterar esse padrão para representar o próximo inteiro imediatamente superior. A título de ilustração, suponha que o padrão 101 esteja associado à variável Y, antes de a instrução

```
incr Y;
```

ser executada. Então, o padrão 110 estará associado a Y imediatamente após a execução. Isto é, foi somado 1 ao valor associado a Y.

A instrução decr, por sua vez, é utilizada para decrementar o valor associado à variável identificada ou, em outras palavras, reduzir em uma unidade o valor representado. Ocorrerá uma exceção quando a variável identificada já estiver associada ao valor zero, caso em que a instrução manterá inalterado o valor. Portanto, se o valor associado a Y for 101, antes de a instrução

```
decr Y;
```

ser executada, o padrão 100 estará associado a Y depois da execução. Entretanto, se o valor de Y fosse zero antes da execução da instrução, esse valor permaneceria zero após a mesma.

A linguagem Bare Bones contém somente uma estrutura de controle, representada pelo par de instruções while-end. A seqüência de instruções

```
while nome not 0 do;
```

.

```
end;
```

(em que *nome* representa qualquer nome de variável) faz com que qualquer instrução ou seqüência de instruções posicionadas entre as instruções while e end sejam repetidas enquanto o valor da variável *nome* não for zero. Para sermos mais precisos, quando uma estrutura while-end é encontrada durante a execução do programa, o valor da variável é primeiramente comparado a zero. Se for zero, a estrutura será ignorada, e a execução continuará com a instrução que seguir o end. Se o valor da variável não for zero, a seqüência de instruções internas da estrutura while-end será executada e o controle, devolvido à instrução while, quando a comparação for feita novamente. Observe que o controle cansativo do processo iterativo é parcialmente realizado pelo programador, que é forçado a pedir explicitamente que o valor da variável seja alterado internamente no corpo de iteração para evitar uma repetição infinida. Por exemplo, a seqüência

```

incr X;
while X not 0 do;
    incr Z;
end;

```

resulta em um processo infinito, porque uma vez que a instrução while seja atingida, o valor associado a X nunca será zero, enquanto a seqüência

```

clear Z;
while X not 0 do;
    incr Z;
    decr X;
end;

```

em última instância termina por transferir o valor inicialmente associado a X para a variável Z.

Observe que as instruções while e end devem aparecer aos pares, iniciando com while. Entretanto, o par while-end pode aparecer dentro da seqüência de instruções que está sendo repetida por outro par while-end. Neste caso, para encontrar as instruções while e end que formam um par, esquadrinha-se o programa, em seu formato escrito, do princípio ao final, associando cada end com a instrução while anterior mais próxima que ainda esteja sem par. Embora não seja sintaticamente necessária, freqüentemente usamos a endentação para facilitar a leitura de tais estruturas.

Como um exemplo final, a seqüência de instruções da Figura 11.4 resulta na associação à variável Z do produto dos valores associados a X e Y, embora tenha o efeito colateral de destruir qualquer valor diferente de zero que possa ter sido anteriormente associado a X. (A estrutura while-end controlada pela variável W tem como função restabelecer o valor original de Y.)

Programação em Bare Bones

Convém lembrar que nosso objetivo na apresentação da linguagem Bare Bones é investigar o que é possível, e não o que é prático. A linguagem Bare Bones provavelmente se mostraria mais estranha do que a maioria das linguagens de programação se fosse aplicada em alguma situação prática. Não obstante, veremos brevemente que esta linguagem simples cumpre o nosso objetivo de apresentar uma linguagem de programação universal isenta de ornamentos. Por ora, simplesmente demonstraremos como a linguagem Bare Bones pode ser usada para expressar algumas operações elementares.

Primeiro notemos que, com uma combinação das instruções de atribuição, qualquer valor (qualquer padrão de bits) pode ser associado a uma determinada variável. Por exemplo, a seguinte seqüência atribui o padrão de bits 11 (a representação binária para 3) à variável X, limpando primeiro qualquer associação anterior e, em seguida, incrementando o seu valor três vezes:

```

clear X;
incr X;
incr X;
incr X;

```

Outra atividade comum em programas é mover dados de uma localização para outra. Em termos de Bare Bones, isto significa atribuir a uma variável o padrão de bits anteriormente associado a outra variável. Isso pode

```

clear Z;
while X not 0 do;
    clear W;
    while Y not 0 do;
        incr Z;
        incr W;
        decr Y;
    end;
    while W not 0 do;
        incr Y;
        decr W;
    end;
    decr X;
end;

```

Figura 11.4 Um programa da linguagem Bare Bones para calcular o produto $X \times Y$.

ser realizado limpando primeiro a variável-destino e, em seguida, incrementando-a um número apropriado de vezes. De fato, já tivemos a oportunidade de observar que a seqüência

```
clear Z;
while X not 0 do;
    incr Z;
    decr X;
end;
```

transfere para Z o valor associado a X. No entanto, esta seqüência tem o efeito colateral de destruir o valor original de X. Para corrigir isso, podemos introduzir uma variável auxiliar, para a qual transferimos primeiramente o valor em questão a partir de sua posição inicial. Utilizamos, então, esta variável auxiliar para restabelecer a original, ao mesmo tempo em que depositamos no destino desejado o valor em questão. Desta maneira, o movimento de Hoje para Amanhã pode ser realizado pela seqüência descrita na Figura 11.5.

Adotamos a sintaxe

```
copy nome1 to nome2;
```

(onde *nome1* e *nome2* representam nomes de variáveis) como uma notação abreviada para uma estrutura de instruções da forma mostrada na Figura 11.5. Assim, embora Bare Bones não tenha uma instrução explícita de cópia de dados (*copy*), freqüentemente escrevemos programas desta forma, sempre levando em conta que, para converter tais programas informais em programas escritos em Bare Bones, é preciso substituir os comandos *copy* pelas suas estruturas while-end equivalentes, utilizando uma variável auxiliar, cujo nome deverá ser diferente de qualquer outro já utilizado no programa.

A universalidade da linguagem Bare Bones

Vamos agora aplicar a tese de Church-Turing para confirmar a nossa afirmação de que Bare Bones é uma linguagem de programação universal. Primeiro, observamos que qualquer programa escrito em Bare Bones pode ser entendido como especificação da computação de uma função. A entrada da função consiste nos valores atribuídos às variáveis antes da execução do programa, e a saída, nos valores das variáveis quando o programa termina. Para computar a função, simplesmente executamos o programa, atribuindo inicialmente os valores apropriados às variáveis, e então observamos os valores destas quando o programa termina.

Sob estas condições, o programa

```
incr X;
```

```
clear Aux;
clear Amanha;
while Hoje not 0 do;
    incr Aux;
    decr Hoje;
end;
while Aux not 0 do;
    incr Hoje;
    incr Amanha;
    decr Aux;
end;
```

descreve a mesma função (a função sucessor) descrita pelo exemplo da máquina de Turing da Seção 11.2. Na verdade, ela aumenta em uma unidade o valor associado a X. Do mesmo modo, considerando as variáveis X e Y como entradas e a variável Z como a saída, o programa abaixo descreve a função adição:

```
copy Y to Z;
while X not 0 do,
    incr Z;
    decr X;
end;
```

Figura 11.5 Uma implementação da instrução “*copy Hoje to Amanha*” na linguagem Bare Bones.

Os pesquisadores mostraram que a linguagem de programação Bare Bones pode ser utilizada para expressar algoritmos que computam todas as funções Turing-

computáveis. Combinando isto com a tese de Church-Turing, temos que qualquer função computável pode sê-lo por um programa escrito em Bare Bones. Ou seja, Bare Bones é uma linguagem de programação universal, no sentido em que, se existir um algoritmo para resolver um problema, então esse problema poderá ser resolvido por algum programa escrito em Bare Bones. Por sua vez, a Bare Bones teoricamente pode servir como uma linguagem de programação de propósito geral.

Dizemos *teoricamente* porque esta linguagem certamente não é tão conveniente quanto as linguagens de alto nível apresentadas no Capítulo 5. No entanto, em sua essência, todas aquelas linguagens contêm, como núcleo, os mesmos recursos disponíveis em Bare Bones. É esse núcleo que, de fato, assegura a universalidade das linguagens, e é por mera conveniência que todos os demais recursos estão presentes nas várias linguagens.

Embora não sejam práticas em um ambiente de programação de aplicações, as linguagens como a Bare Bones são usadas em estudos teóricos da Ciência da Computação. Por exemplo, no Apêndice E, usamos a Bare Bones como ferramenta para resolver a questão da equivalência entre as estruturas iterativas e recursivas, levantada no Capítulo 4. Ali descobrimos que a nossa suspeita de equivalência era, de fato, justificável.



QUESTÕES/EXERCÍCIOS

1. Mostre que o comando `invert X;` (cuja ação é converter o valor de `X` para zero se o seu valor inicial for diferente de zero e para 1 se for zero) pode ser simulado por um trecho de programa escrito na linguagem Bare Bones.
2. Mostre que mesmo a nossa linguagem simples Bare Bones contém mais instruções do que o necessário, mediante a demonstração de que a instrução `clear` pode ser substituída pela combinação de outras instruções da linguagem.
3. Mostre que uma estrutura `if-then-else` pode ser simulada utilizando Bare Bones. Para isso, escreva um trecho de programa em Bare Bones que simule a ação da instrução `if X not 0 then S1 else S2;` onde `S1` e `S2` representam seqüências arbitrárias de instruções.
4. Mostre que todas as instruções de Bare Bones podem ser expressas em termos da linguagem de máquina do Apêndice C (ou seja, que Bare Bones pode ser utilizada como linguagem de programação para tal máquina).
5. Como os números negativos podem ser tratados na linguagem Bare Bones?
6. Descreva a função computada pela seguinte programa em Bare Bones, pressupondo que a entrada da função seja representada por `X` e a saída, por `Z`:
`clear Z;`
`while X not 0 do;`
`incr Z;`
`incr Z;`
`decr X;`
`end;`

11.4 Uma função incomputável

Agora, vamos identificar uma função que seja Turing-incomputável e, portanto, em decorrência da tese de Church-Turing, amplamente aceita como incomputável de um modo geral. Assim, ela é uma função cuja computação permanece além da capacidade dos computadores atuais.

O problema da parada

A função incomputável que iremos revelar está associada com um problema conhecido como o **problema da parada**, o qual (de uma maneira informal) é o problema de tentar prever se um programa terminará se for iniciado sob certas condições. Por exemplo, considere o programa simples escrito em Bare Bones

```
while X not 0 do;
    incr X;
end;
```

Se executarmos esse programa enquanto o valor da variável X for 0, o laço não será executado e a execução do programa terminará rapidamente. Entretanto, se executarmos o programa com qualquer outro valor inicial de X, o laço será executado eternamente, levando a um processo que não termina.

Nesse caso, então, é fácil concluir que a execução do programa só terminará quando ele for iniciado com 0 atribuído à variável X. Contudo, quando passamos a exemplos mais complexos, a tarefa de prever o comportamento do programa vai se tornando mais complicada. De fato, em alguns casos, a tarefa é impossível, como veremos. No entanto, primeiro precisamos formalizar a nossa terminologia e focalizar mais precisamente nossos pensamentos.

Nosso exemplo mostrou que o término de um programa pode depender do valor inicial de suas variáveis. Assim, se quisermos prever se a execução de um programa terminará, deveremos ser precisos quanto a esses valores iniciais. A escolha que fazemos desses valores pode parecer estranha à primeira vista, mas não desanime. Nossa objetivo é tirar vantagem de uma técnica chamada auto-referência — a idéia de um objeto referir-se a si próprio. Muitas situações como essa têm conduzido a resultados surpreendentes em matemática, desde curiosidades informais, como a afirmação *Esta sentença é falsa* até o paradoxo mais sério, representado pela pergunta *o conjunto de todos os conjuntos contém a si próprio?* O que pretendemos fazer então é definir o ambiente para uma linha de raciocínio semelhante a *Se isto faz, então não faz, mas, se isto não faz, então faz*, como veremos brevemente.

Em nosso caso, a auto-referência será alcançada atribuindo-se às variáveis de um programa um valor inicial que represente o próprio programa. Para isso, observe que cada programa em Bare Bones pode ser codificado como um padrão de bits único e extenso, em um formato de um caractere por byte, usando ASCII. Além disso, cada variável em um programa Bare Bones é do tipo “padrão de bits de comprimento arbitrário”, de modo que podemos atribuir essa versão codificada de um programa como valor de suas variáveis.

Vamos considerar o que aconteceria se isso fosse feito no caso do programa simples

```
while X not 0 do;
    incr X;
end;
```

Queremos saber o que acontecerá se iniciarmos esse programa com a sua versão codificada atribuída à variável X (Figura 11.6). Nesse caso, a resposta é imediata. Uma vez que X terá um valor diferente de zero, o programa ficará no laço e não terminará. No entanto, se realizarmos uma experiência similar com o programa

```
clear X;
while X not 0 do;
    incr X;
end;
```

este terminará, uma vez que a variável X teria o valor 0 quando a estrutura while-end fosse alcançada, independentemente de seu valor inicial.

Façamos então a seguinte definição: Um programa em Bare Bones **termina por si mesmo** a execução do programa com todas as suas variáveis iniciadas com a representação codificada de si mesmo

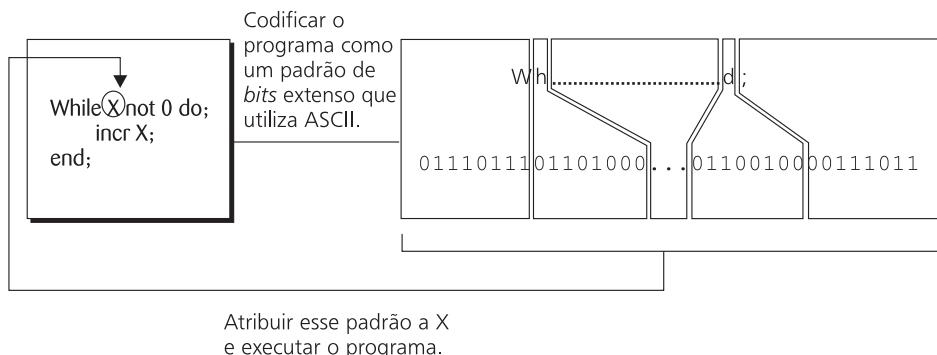


Figura 11.6 Teste para verificar se um programa termina por si mesmo.

levar a um processo que termina. Informalmente, um programa termina por si mesmo se a sua execução termina quando iniciada com ele próprio como entrada. Aqui então está a auto-referência prometida.

Note que o fato de um programa terminar por si mesmo provavelmente não terá nada a ver com o objetivo para o qual ele foi escrito. É meramente uma propriedade que cada programa em Bare Bones possui ou não. Isto é, cada programa em Bare Bones termina ou não por si mesmo.

Agora podemos descrever o problema da parada de uma maneira precisa. É o problema de se determinar se os programas em Bare Bones terminam ou não por si mesmos. Estamos prestes a ver que não existe algoritmo que responda essa questão em geral. Isto é, não existe um algoritmo que, dado qualquer programa em Bare Bones, seja capaz de determinar se ele termina ou não por si mesmo. Assim, a solução do problema da parada permanece além da capacidade dos computadores.

O fato de termos aparentemente resolvido o problema da parada em nossos exemplos anteriores e agora afirmarmos que o problema da parada é insolúvel pode soar contraditório, por isso, façamos uma pausa para esclarecer. As observações que usamos em nossos exemplos seriam únicas àqueles casos particulares, e não aplicáveis a todas as situações. O que o problema da parada precisa é de um algoritmo genérico que possa ser aplicado a qualquer programa em Bare Bones para determinar se ele termina por si mesmo. A nossa habilidade de aplicar certas percepções para determinar se um programa específico termina por si mesmo de maneira alguma implica a existência de uma abordagem genérica que possa ser aplicada em todos os casos.

A insolubilidade do problema da parada

Agora queremos mostrar que a resolução do problema da parada está além da capacidade das máquinas. Nossa abordagem é mostrar que ela exige um algoritmo para computar uma função incomputável. As entradas da função em questão são versões codificadas de programas em Bare Bones; suas saídas são limitadas aos valores 0 e 1. Mais precisamente, definimos a função de tal forma que a representação de um programa que termina por si mesmo produz o valor de saída 1, e a representação de um programa que não termina por si mesmo, o valor de saída 0. Para sermos concisos, nos referiremos a essa função como a *função de parada*.

Nossa tarefa é mostrar que a função de parada é incomputável. Nossa abordagem é a técnica conhecida como “prova por contradição”. Em síntese, provamos que uma declaração é falsa mostrando que ela não pode ser verdadeira. Vamos, então, mostrar que a declaração “a função de parada é computável” não pode ser verdadeira. Nosso argumento completo está mostrado na Figura 11.7.

Se a função de parada é computável, então (uma vez que a Bare Bones é uma linguagem de programação universal) deve existir um programa em Bare Bones que a compute. Em outras palavras,

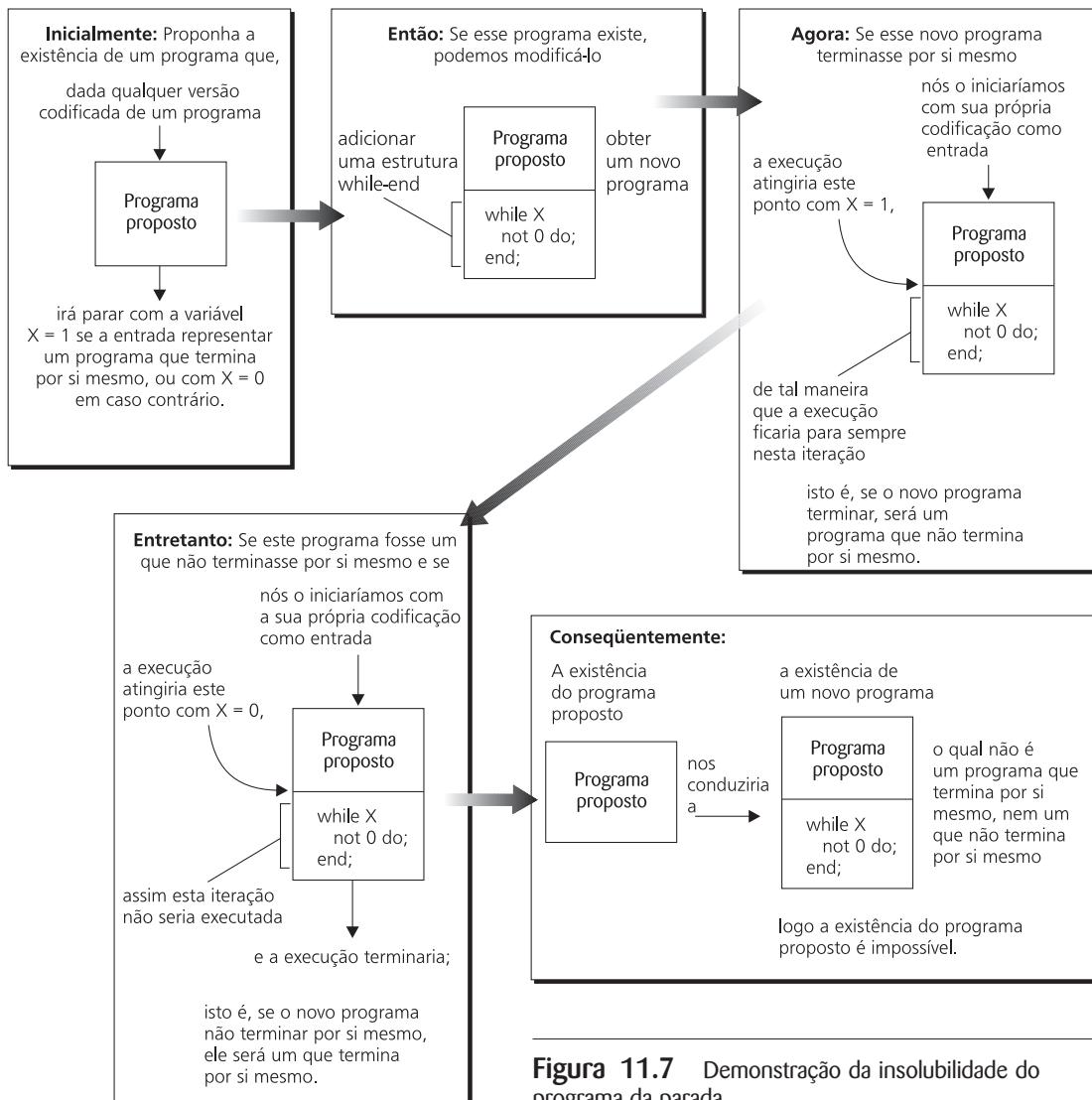


Figura 11.7 Demonstração da insolubilidade do programa da parada.

existe um programa em Bare Bones que terminará com sua saída igual a 1 se a entrada for a versão codificada de um programa que termina por si mesmo e igual a 0 em caso contrário.

Para aplicar esse programa, não precisamos identificar qual variável é de entrada; simplesmente iniciamos todas as variáveis do programa com a representação codificada do programa a ser testado. Isto porque uma variável que não seja de entrada é inherentemente uma variável cujo valor inicial não afeta a saída final do programa. Concluímos que se a função de parada é computável, então existe um programa em Bare Bones que termina com a sua saída igual a 1 se todas as suas variáveis são iniciadas com a versão codificada de um programa que termina por si mesmo, igual a 0 em caso contrário.

Assumindo que o nome da variável de saída do programa seja X (se não for, poderemos simplesmente renomear as variáveis), modificamos o programa adicionando ao seu final as instruções

```
while X not 0 do;
end;
```

produzindo assim um novo programa. Dessa maneira, o programa deverá obrigatoriamente terminar ou não por si mesmo. Estamos, porém, prestes a verificar que não acontecerá uma nem outra coisa.

Se este fosse um programa que terminasse por si mesmo, e se o processássemos com a entrada igual à sua própria versão codificada, então quando sua execução alcançasse a instrução while que acabamos de adicionar, a variável X conteria o valor 1. (Neste ponto, o novo programa seria idêntico ao original, que produzia o valor 1 se sua entrada fosse a representação de um programa que terminasse por si mesmo.) Aqui, a execução do programa estaria presa para sempre na estrutura while-end porque não tomamos qualquer providência para decrementar a variável X no interior desta iteração. Entretanto, isso contradiz a nossa hipótese de que o novo programa termina por si mesmo. Então devemos concluir que ele não termina por si mesmo.

Entretanto, se admitirmos que este novo programa não termina por si mesmo e o executarmos com suas variáveis iniciadas com a sua própria representação codificada, ele alcançará a instrução while adicionada, com o valor 0 na sua variável X. (Isto acontece porque as instruções que precedem a instrução while constituem o programa original que produz uma saída 0 quando sua entrada representa um programa que não termina por si mesmo.) Neste caso, a iteração representada pela estrutura while-end não será ativada e, portanto, o programa deverá parar. No entanto, esta é a propriedade dos programas que terminam por si mesmos, portanto, seríamos forçados a concluir que o novo programa é um programa que termina por si mesmo, contradizendo a nossa hipótese, da mesma forma que ocorreu no caso anterior.

Em resumo, estamos às voltas com uma situação impossível de um programa que por um lado deve simultaneamente terminar e não terminar e, por outro, não deve terminar nem não terminar. Consequentemente, a hipótese que conduziu a esse dilema tem de ser falsa.

Concluímos que a função de parada é incomputável, e uma vez que a solução do problema da parada depende da computação dessa função, devemos concluir que a resolução do problema da parada permanece além da capacidade de qualquer sistema algorítmico. Tais problemas são chamados problemas insolúveis.

Para terminar, devemos relacionar com as idéias do Capítulo 10 aquilo que acabamos de discutir. Uma importante questão subjacente é se as capacidades das máquinas de computação incluem ou não aquelas exigidas pela própria inteligência. Lembre-se de que as máquinas podem resolver apenas os problemas com solução algorítmica, e agora vimos que existem problemas sem solução algorítmica. Portanto, a questão é se a inteligência natural incorpora algo mais que a execução de processos algorítmicos. Se não incorpora, então os limites que identificamos aqui também são os do pensamento humano. É desnecessário dizer que esta é uma questão altamente polêmica e, às vezes, emocional. Se, por exemplo, a mente humana nada mais é do que uma máquina programada, poderíamos concluir que os homens não possuem livre arbítrio.



QUESTÕES/EXERCÍCIOS

- O programa abaixo, em Bare Bones, termina por si mesmo? Justifique a sua resposta
incr X;
decr Y;
- O programa abaixo, em Bare Bones, termina por si mesmo? Justifique a sua resposta.
copy X to Y;
incr Y;
incr Y;
while X not 0 do;
decr X;

```

decr X;
decr Y;
decr Y;
end;
decr Y;
while Y not 0 do;
end;

```

3. Aponte o erro encontrado no seguinte cenário:

Em uma certa comunidade, todos possuem sua própria casa. O pintor de casas da comunidade reivindica pintar todas as casas não pintadas pelos seus donos, e somente elas.

(Sugestão: Quem pinta a casa do pintor?)

11.5 Complexidade de problemas

Na Seção 11.4, investigamos problemas quanto às possibilidades de serem ou não resolvidos. Nesta, estamos interessados na questão de um problema passível de solução apresentar ou não uma solução prática. Veremos que alguns problemas teoricamente passíveis de solução possuem tal complexidade que se tornaram insolúveis de um ponto de vista prático.

Medição da complexidade de um problema

Começamos retomando o nosso estudo de eficiência de algoritmos desenvolvido na Seção 4.6. Ali, usamos a notação Θ para classificar os algoritmos de acordo com o tempo exigido para executá-los. Vimos que o algoritmo de ordenação por inserção pertence à classe $\Theta(n^2)$; o de busca seqüencial, à classe $\Theta(n)$ e o de busca binária, à classe $\Theta(\lg n)$. Usaremos agora esse sistema de classificação para nos ajudar a identificar a complexidade dos problemas. Nosso objetivo é desenvolver um sistema de classificação que nos diga quais problemas são mais complexos e, em última instância, quais são tão complexos que suas soluções se tornam impraticáveis.

A razão de o nosso estudo se basear no conhecimento da eficiência dos algoritmos é que desejamos medir a complexidade de um problema em termos da complexidade de suas soluções. Consideramos um problema simples como aquele que possui uma solução simples; um problema complexo é o que não a possui. Note que o fato de um problema possuir uma solução difícil não significa necessariamente que seja complexo. Afinal de contas, um problema possui muitas soluções, e uma delas pode ser difícil. Assim, para concluir que um problema é de fato, complexo, é necessário mostrar que nenhuma de suas soluções é simples.

Na Ciência da Computação, os problemas de interesse são aqueles passíveis de solução por máquinas. As soluções desses problemas são formuladas como algoritmos. Assim, a complexidade de um problema é determinada pelas propriedades dos algoritmos que o resolvem. Mais precisamente, a complexidade do algoritmo mais simples que resolve um problema é considerada como a complexidade do próprio problema.

Entretanto, como medimos a complexidade de um algoritmo? Infelizmente, o termo *complexidade* tem diferentes interpretações. Uma delas se refere à quantidade de ramificações e decisões envolvida no algoritmo. Nesse enfoque, um algoritmo complexo seria o que envolve um conjunto entrelaçado de direções. Esta é a complexidade segundo a perspectiva de um engenheiro de software, que está interessado em tópicos relacionados à descoberta e à representação de algoritmos, mas ela não captura o conceito de complexidade analisado do ponto de vista de uma máquina. As máquinas realmente não tomam qualquer decisão ao selecionar a próxima instrução para ser executada, seguindo apenas o seu ciclo repetidamente, sempre executando a instrução indicada pelo contador de instruções. Conseqüentemente, uma máquina pode executar um conjunto de instruções emaranhadas tão facilmente quanto executaria uma

lista de instruções dada em uma certa ordem. Portanto, esta interpretação da complexidade tende a medir a dificuldade encontrada na representação do algoritmo, e não no algoritmo propriamente dito.

Uma interpretação que reflete com maior precisão a complexidade de um algoritmo é obtida considerando os algoritmos a partir de um ponto de vista da máquina. Nesse contexto, medimos a complexidade de um algoritmo em termos do tempo necessário para a sua execução, que é proporcional ao número de passos que devem ser dados. Note-se que isso não é o mesmo que o número de instruções presentes no programa escrito. Por exemplo, uma instrução iterativa cujo corpo consiste em uma única instrução de impressão, mas cujo controle comanda por 100 vezes a execução do corpo, é equivalente a 100 instruções de impressão na fase de execução. Tal rotina é considerada mais complexa do que uma lista de 50 instruções similares de impressão, embora na forma escrita a última pareça ser mais longa. O ponto a ser lembrado é que o nosso significado de *complexidade* está, em última instância, relacionado com o tempo necessário para uma máquina executar um algoritmo, e não com o tamanho da representação escrita do algoritmo.

Portanto, consideraremos um problema complexo se todas as suas soluções exigem muito tempo. Essa definição de complexidade se refere à **complexidade de tempo**. Já mostramos o conceito de complexidade de tempo indiretamente em nosso estudo de eficiência dos algoritmos, no Capítulo 4 (Seção 4.6). Sem dúvida, o estudo da eficiência de um algoritmo é o estudo da sua complexidade de tempo — os dois são simplesmente recíprocos. Isto é, “mais eficiente” se iguala a “menos complexo”. Assim, em termos da complexidade de tempo, o algoritmo de busca seqüencial (cuja eficiência vimos que é $\Theta(n)$) é uma solução mais complexa ao problema de consultar uma tabela do que o algoritmo de busca binária (cuja eficiência vimos que é $\Theta(\lg n)$).

Vamos então aplicar o nosso conhecimento da complexidade dos algoritmos visando obter os meios para identificar a complexidade dos problemas. Definimos a complexidade (de tempo) de um problema como $\Theta(f(n))$, onde $f(n)$ é alguma expressão matemática em n , se existir um algoritmo para resolver o problema cuja complexidade de tempo está em $\Theta(f(n))$ e nenhum outro algoritmo que resolve o problema possuir complexidade de tempo menor. Isto é, a complexidade (de tempo) de um problema é definida como a complexidade (de tempo) de sua melhor solução. Infelizmente, encontrar a solução mais simples para um problema e saber que esta é, de fato, a mais simples é, por si só, um problema muito difícil. Nessas situações, a notação O (uma variação da notação Θ) é usada para representar o que se sabe sobre a complexidade de um problema. Mais precisamente, se $f(n)$ é uma expressão matemática em n , e se um problema pode ser resolvido por um algoritmo em $\Theta(f(n))$, então dizemos que o problema está em $O(f(n))$, que é lido “O de $f(n)$ ”. Assim, dizer que um problema pertence a $O(f(n))$ significa que ele possui uma solução cuja complexidade está em $\Theta(f(n))$, mas provavelmente tem uma solução melhor.

Complexidade de espaço

Uma alternativa à medição da complexidade em termos de tempo é medir o espaço de armazenamento exigido — o que resulta na medida chamada **complexidade de espaço**. Isto é, a complexidade de espaço de um problema é determinada pela quantidade de espaço de armazenamento necessário para resolvê-lo. No texto, vimos que a complexidade de tempo para ordenar uma lista com n elementos é $O(n \lg n)$. A complexidade de espaço do mesmo problema é não mais que $O(n + 1) = O(n)$. De fato, ordenar uma lista com n elementos usando a ordenação por inserção exige espaço para guardar a lista propriamente dita mais o espaço para guardar um único elemento, em caráter temporário. Assim, se fôssemos solicitados a ordenar listas cada vez maiores, descobriríamos que o tempo necessário para cada ordenação cresceria mais rapidamente do que o espaço exigido. Este é, de fato, um fenômeno comum. Uma vez que usar o espaço toma tempo, a complexidade de espaço de um problema jamais cresce mais rapidamente do que a sua complexidade de tempo.

Freqüentemente há disputa entre a complexidade de tempo e a de espaço. Em algumas aplicações, pode ser vantajoso realizar algumas computações antecipadamente e armazenar os resultados em uma tabela, de onde podem ser rapidamente recuperados mais tarde, quando necessário. Essa técnica de “consulta à tabela” diminui o tempo necessário à execução do sistema final, às expensas do espaço adicional exigido para armazenar a tabela. Entretanto, a compressão de dados freqüentemente é usada para reduzir as necessidades de armazenamento, às custas de um tempo adicional exigido para compactar e descompactar os dados.

Nossa investigação de busca e ordenação diz-nos que o problema de consultar uma lista de tamanho n (quando tudo o que sabemos da lista é que ela está ordenada) está em $O(\lg n)$, uma vez que o algoritmo de busca binária resolve o problema. Além disso, pesquisadores mostraram que o problema da busca está, de fato, em $\Theta(\lg n)$, assim a busca binária representa uma solução ótima para o problema. Em contraste, sabemos que o problema de ordenar uma lista de comprimento n (quando nada sabemos sobre a distribuição original de seus valores) está em $O(n^2)$, uma vez que o algoritmo de ordenação por inserção resolve o problema. Contudo, sabe-se que o problema da ordenação está em $\Theta(n \lg n)$, o que nos diz que o algoritmo de ordenação por inserção não é uma solução ótima (no contexto da complexidade de tempo).

Um exemplo de uma solução melhor para o problema da ordenação é o algoritmo de ordenação por intercalação. Sua abordagem é intercalar pequenas partes ordenadas da lista para obter partes maiores ordenadas que possam então ser intercaladas para obter partes ordenadas ainda maiores. Cada processo de intercalação aplica o algoritmo de intercalação que vimos na discussão de arquivos seqüenciais (Figura 8.3). Por conveniência, o apresentamos novamente na Figura 11.8, desta vez no contexto de intercalação de duas listas. O algoritmo de ordenação por intercalação completo (recursivo) é apresentado no procedimento OrdenaIntercala na Figura 11.9. Quando solicitado a ordenar uma lista, esse procedimento verifica se a lista possui menos de dois elementos. Nesse caso, a tarefa do procedimento está completa. Caso contrário, o procedimento divide a lista em duas partes, pede a outras cópias do OrdenaIntercala para ordená-las e então intercala essas partes ordenadas para obter a versão final ordenada da lista.

Para analisar a complexidade desse algoritmo, primeiro consideramos o número de comparações entre os elementos da lista que devem ser feitas quando se intercala uma lista de comprimento r com outra de comprimento s . O processo de intercalação compara repetidamente um elemento de uma lista com um elemento da outra e coloca o menor deles na lista de saída. Assim, cada vez que uma comparação é feita, o número de elementos a ser considerado é reduzido em uma unidade. Uma vez que existem apenas $r + s$ elementos, concluímos que o processo de intercalação de duas listas envolverá não mais que $r + s$ comparações.

Consideremos agora o algoritmo completo de ordenação por intercalação. Ele ataca a tarefa de ordenar uma lista de comprimento n de forma que o problema inicial de ordenação seja reduzido a dois problemas menores, cada um dos quais se propõe a ordenar uma lista de comprimento aproximadamente $n/2$. Esses dois problemas, por sua vez, são reduzidos a um total de quatro problemas de ordenação de listas de comprimento aproximadamente igual a $n/4$. Esse processo de divisão pode ser resumido na

```

procedimento Intercala (ListaEntradaA, ListaEntradaB, ListaSaida)
  se (as duas listas estiverem vazias) então (Parar, com ListaSaida vazia)
  se (ListaEntradaA estiver vazia)
    então (Declarar que ela chegou ao fim)
    senão (Declarar seu primeiro elemento como elemento corrente)
  se (ListaEntradaB estiver vazia)
    então (Declarar que ela chegou ao fim)
    senão (Declarar seu primeiro elemento como elemento corrente)
  enquanto (nenhuma das listas chegar ao fim) faz
    (Colocar o “menor” elemento corrente na ListaSaida;
     se (esse elemento corrente for o último em sua lista)
       então (Declarar que essa lista chegou ao fim)
       senão (Declarar o próximo elemento nessa lista de entrada como elemento corrente)
    )
  
```

Iniciando com o elemento corrente na lista de entrada que não chegou ao fim, copiar os elementos restantes para a ListaSaida.

Figura 11.8 Procedimento Intercala para intercalar duas listas.

procedimento OrdenaIntercala(Lista)

se (Lista possuir mais de um elemento)

então (Aplicar o procedimento OrdenaIntercala para ordenar a primeira metade da Lista;

Aplicar o procedimento OrdenaIntercala para ordenar a segunda metade da Lista;

Aplicar o procedimento IntercalaListas para intercalar a primeira e segunda metades da Lista, produzindo a versão ordenada da Lista

)

Figura 11.9 Algoritmo de ordenação por intercalação implementado como procedimento OrdenaIntercala.

estrutura de árvore mostrada na Figura 11.10, onde cada nó da árvore apresenta um único problema no processo recursivo e os ramos abaixo dele representam os problemas menores derivados do ancestral. Por isso, encontramos o número total de comparações que ocorrem no processo completo de ordenação somando os números de comparações que ocorrem em cada nó da árvore.

Vamos inicialmente determinar o número de comparações feitas em cada nível da árvore. Observe que cada nó que aparece em qualquer nível da árvore possui a tarefa de ordenar um único segmento da lista original. Isso é feito pelo processo de intercalação, portanto requer um número de comparações igual ao número de elementos do segmento, como já demonstramos. Por isso, cada nível da árvore requer um número de comparações igual ao número total de elementos do segmento, e como os segmentos em um mesmo nível da árvore representam partes disjuntas da lista original, esse total não é maior do que o comprimento da lista original. Conseqüentemente, cada nível da árvore envolve não mais que n comparações. (Obviamente, o nível mais baixo envolve a ordenação de listas com um único elemento, que não necessita comparação alguma.)

Agora vamos determinar o número de níveis na árvore. Para isso, observe que o processo de dividir problemas em problemas menores continua até que as listas de comprimento menor que dois sejam obtidas. Assim, o número de níveis na árvore é determinado pelo número de vezes que, iniciado com o valor n , podemos repetidamente dividir por dois até que o resultado não seja maior que um, que é $\lg n$. Mais precisamente, existem não mais do que $\lceil \lg n \rceil$ níveis na árvore que envolvam comparações, onde $\lceil \lg n \rceil$ representa o valor de $\lg n$ arredondado para cima até o próximo inteiro.

Finalmente, o número total de comparações feitas pelo algoritmo de ordenação por intercalação quando ordena uma lista de comprimento n é obtido multiplicando-se o número de comparações feitas em cada nível da árvore pelo número de níveis nos quais as comparações são feitas. Concluímos que ele não será maior que $n \lceil \lg n \rceil$. Uma vez que o gráfico de $n \lceil \lg n \rceil$ possui a mesma forma geral do gráfico de $n \lg n$, concluímos que o algoritmo de ordenação por intercalação pertence à classe $O(n \lg n)$. Combinando isto com o fato de os pesquisadores garantirem que o problema da ordenação possui complexidade $\Theta(n \lg n)$, podemos afirmar que o algoritmo de ordenação por intercalação representa uma solução ótima para o problema da ordenação.

Problemas polinomiais versus não-polinomiais

Suponha que $f(n)$ e $g(n)$ sejam expressões matemáticas. Dizer que $g(n)$ é limitada por $f(n)$ significa que, conforme aplicamos essas expres-

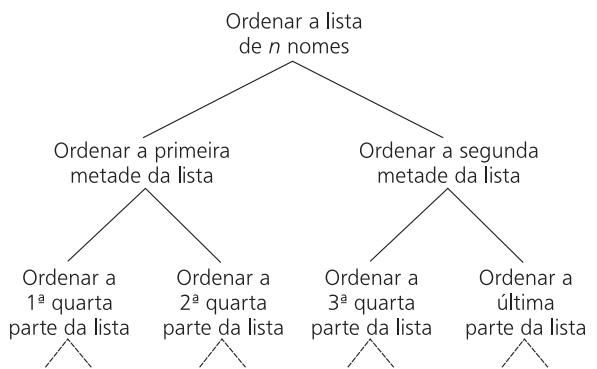


Figura 11.10 A hierarquia dos problemas gerados pelo algoritmo de ordenação por intercalação.

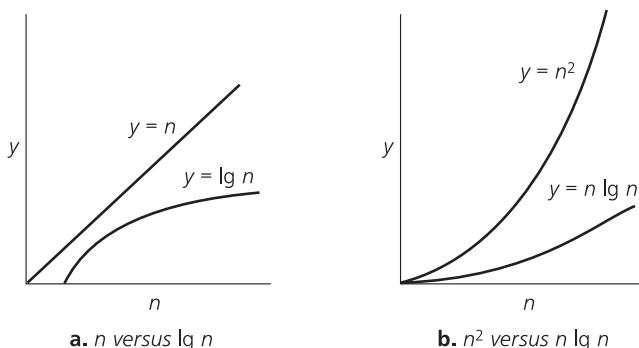


Figura 11.11 Gráfico das expressões matemáticas n , $\lg n$, $n \lg n$ e n^2 .

rior nos diz que os problemas de consultar e ordenar uma lista pertencem a P.

Dizer que um problema é polinomial é declarar algo a respeito do tempo necessário para resolvê-lo. Frequentemente dizemos que um problema em P pode ser resolvido em um tempo polinomial, ou que o problema tem uma solução em tempo polinomial.

Identificar os problemas que pertencem a P é de fundamental importância na Ciência da Computação porque se relaciona intimamente com a questão sobre se os problemas possuem soluções práticas. Sem dúvida, os problemas que estão fora da classe P são caracterizados por possuir tempos de execução extremamente longos, mesmo para entradas de tamanho moderado. Considere, por exemplo, um problema cuja solução requeira 2^n passos. A expressão exponencial 2^n não é limitada por polinômio — se $f(n)$ é um polinômio, à medida que aumentarmos o valor de n , veremos que os valores de 2^n serão, em última instância, maiores do que os de $f(n)$. Isso significa que um algoritmo com complexidade $\Theta(2^n)$ geralmente é menos eficiente, e assim exige mais tempo que um algoritmo com complexidade $\Theta(f(n))$. Diz-se que um algoritmo cuja complexidade é identificada por uma exponencial requer tempo exponencial.

Como exemplo, suponha o problema de listar todos os possíveis grupos que podem ser formados com os elementos de um conjunto de n pessoas. Dado que há $2^n - 1$ grupos (consideraremos também como um grupo aquele formado por todos os elementos, mas não o conjunto vazio como um grupo), qualquer algoritmo capaz de resolver este problema terá pelo menos $2^n - 1$ passos e, portanto, uma complexidade mínima de tal ordem. Entretanto, a expressão $2^n - 1$, por ser exponencial, não é limitada por polinômio. Então, qualquer solução para esse problema consome muito tempo à medida que se aumenta o tamanho do conjunto inicial de onde são formados os grupos.

Em contraste com o nosso problema dos grupos, cuja complexidade é grande apenas devido ao tamanho de sua saída, existem problemas cujas complexidades são grandes apesar da sua saída final ser uma resposta simples, do tipo sim ou não. Um exemplo envolve a capacidade de responder a perguntas sobre a veracidade de declarações que tratam de adição de números reais. Por exemplo, podemos facilmente reconhecer que a resposta para a pergunta “é verdade que existe um número real que quando somado consigo mesmo produz o valor 6?” é sim, enquanto a resposta para “é verdade que há um real diferente de zero que quando adicionado consigo mesmo resulta 0?” é não. Contudo, à medida que tais perguntas se tornam mais complicadas, a nossa capacidade de respondê-las diminui. Se nos deparamos com muitas destas perguntas, poderemos ser tentados a solicitar auxílio a um programa computacional. Infelizmente, foi demonstrado que a capacidade de responder a tais perguntas exige tempo exponencial e, desta forma, até mesmo um computador falha, em última instância, em fornecer respostas em um intervalo de tempo razoável, à medida que as perguntas se tornam mais complexas.

O fato de os problemas teoricamente passíveis de solução que não pertencem a P possuírem essas enormes complexidades de tempo nos leva a concluir que esses problemas são essencialmente insolúveis na prática. Os cientistas da computação chamam esses problemas de intratáveis. Por sua vez, os proble-

sões a valores cada vez maiores de n , o valor de $f(n)$ irá tornar-se maior que o de $g(n)$ e permanecer maior para todos os valores maiores que n . Em outras palavras, $g(n)$ é limitada por $f(n)$ quando o gráfico de $f(n)$ está acima do gráfico de $g(n)$ para “grandes” valores de n . Por exemplo, a expressão $\lg n$ é limitada pela expressão n (Figura 11.11a) e $n \lg n$, por n^2 (Figura 11.11b).

Dizemos que um problema é **polinomial** se ele pertence à classe $O(f(n))$, onde a expressão $f(n)$ é polinomial ou limitada por um polinômio. A coleção de todos os problemas polinomiais é representada por **P**. Note que a nossa investigação anterior nos diz que os problemas de consultar e ordenar uma lista pertencem a P.

mas que possuem solução prática estão contidos em P. Assim, um entendimento das fronteiras da classe P tem se tornado um objetivo importante para os cientistas da computação.

Problemas NP

Vamos agora considerar o **problema do caixeleiro viajante**: um caixeleiro viajante deve visitar os seus clientes em diferentes cidades sem exceder o seu orçamento de viagem. Seu problema, então, consiste em determinar uma trajetória (a partir da sua casa, conectando todas as cidades envolvidas e retornando à sua casa) cujo comprimento total não exceda a quilometragem máxima permitida.

A solução tradicional para este problema é considerar, de maneira sistemática, cada trajetória possível e comparar os seus comprimentos com o limite de quilometragem até que seja encontrado algum caminho aceitável ou que todas as possibilidades tenham sido consideradas. Essa abordagem, porém, não produz uma solução em tempo polinomial. À medida que o número de cidades aumenta, o número de trajetórias que devem ser testadas cresce mais rapidamente que qualquer polinômio. Assim, não é prático resolver o problema do caixeleiro viajante desta maneira nos casos em que o número de cidades for grande.

Concluímos que, para resolver este problema em um tempo razoável, devemos encontrar um algoritmo mais rápido. Essa busca pode ser estimulada pelo fato de que, se existir um caminho satisfatório e se este for selecionado em primeiro lugar, o algoritmo assim proposto terminará rapidamente. Por exemplo, a seguinte seqüência de instruções pode ser executada rapidamente e tem potencial para resolver o problema:

```
Escolher uma das trajetórias possíveis e calcular sua quilometragem total.  
Se (esta quilometragem não excede a permitida)  
    então (considerar encontrada uma solução)  
    senão (nada pode ser concluído)
```

No entanto, esta seqüência de instruções não é um algoritmo no sentido técnico. Sua primeira instrução é ambígua, pois não especifica qual a trajetória a ser selecionada nem como a decisão deve ser tomada. Em vez disto, ela se apóia na criatividade do mecanismo que executa o programa para tomar decisões a seu modo. Dizemos que instruções dessa natureza são não-determinísticas, e chamamos um “algoritmo” que as contenha de **algoritmo não-determinístico**.

Note-se que, à medida que o número de cidades cresce, o tempo necessário para executar o algoritmo não-determinístico cresce de forma relativamente lenta. O processo de selecionar uma trajetória consiste apenas em produzir uma lista de cidades, o que pode ser feito em um tempo proporcional ao número de cidades. Além disso, o tempo necessário para calcular a distância total da trajetória escolhida também é proporcional ao número de cidades a serem visitadas, e o tempo necessário para comparar este total com o limite de quilometragem independe do número de cidades. Assim, o tempo necessário para executar este algoritmo não-determinístico é limitado por um polinômio. Portanto, é possível resolver o problema do caixeleiro viajante com um algoritmo não-determinístico processado em tempo polinomial.

Naturalmente, a nossa solução não-determinística não é totalmente satisfatória. Ela se baseia em um lance de sorte. Entretanto, a sua existência é suficiente para levantar uma dúvida sobre a possível existência de uma solução determinística para o problema do caixeleiro viajante, processada em tempo polinomial. Se tal solução existe ou não, ainda é uma questão em aberto. De fato, o problema do caixeleiro viajante é apenas um de uma longa série de problemas que apresentam soluções não-determinísticas executáveis em tempo polinomial, mas para os quais ainda não foi encontrada uma solução determinística polinomial. A eficiência exasperante das soluções não-determinísticas para estes problemas faz com que se espere ansiosamente que soluções determinísticas eficientes sejam descobertas algum dia, embora a maioria creia que tais problemas são demasiadamente complexos para as capacidades dos algoritmos determinísticos eficientes.

Um problema que pode ser resolvido em tempo polinomial por um algoritmo não-determinístico é chamado de **problema polinomial não-determinístico** ou, de modo abreviado, **problema NP**. É comum denotar a classe de problemas NP por **NP**. Note-se que todos os problemas em P também estão em

Determinístico versus não-determinístico

Em muitos casos, existe uma linha tênue que separa um algoritmo determinístico de um não-determinístico. Ainda assim, a distinção é clara e significativa. Um algoritmo determinístico não se apóia na capacidade criativa do mecanismo que o executa, enquanto o “algoritmo” não-determinístico, sim. Por exemplo, compare a instrução

Vá para o próximo cruzamento e vire à direita ou à esquerda
com a instrução

Vá para o próximo cruzamento e vire à direita ou à esquerda, dependendo do que dirá a pessoa que está na esquina.

Nos dois casos, a ação tomada pela pessoa que segue as instruções não é determinada antes da execução. Contudo, a primeira instrução exige que a pessoa tome uma decisão baseada em seu próprio julgamento e é, portanto, não-determinística. A segunda instrução não faz tal exigência à pessoa – em cada estágio, ela é informada sobre como agir. Se a primeira instrução for seguida por várias pessoas diferentes, algumas poderão virar à direita e outras à esquerda. Se várias pessoas seguirem a segunda instrução e receberem a mesma informação, todas virarão para a mesma direção. Aqui, então, está uma distinção importante entre algoritmos determinísticos e “algoritmos” não-determinísticos. Se um algoritmo determinístico for executado repetidamente com os mesmos dados de entrada, as mesmas ações serão realizadas. Contudo, um “algoritmo” não-determinístico pode produzir diferentes ações quando repetido sob condições idênticas.

NP dado que a qualquer algoritmo (determinístico) pode ser acrescentada alguma instrução não-determinística, sem que o seu desempenho seja afetado.

Entretanto, saber se os problemas NP também estão em P é uma questão em aberto, o que já observamos no nosso problema do caixeiro viajante. Trata-se do problema não solucionado mais amplamente conhecido da Ciência da Computação. Sua resolução poderia trazer consequências significativas. Por exemplo, na próxima seção, veremos que têm sido desenvolvidos sistemas de criptografia cuja integridade se apóia no enorme tempo necessário para resolver problemas, similar ao problema do caixeiro viajante. Se for descoberta a existência de soluções eficientes para tais problemas, os sistemas de segurança estarão comprometidos.

Os esforços para solucionar a questão quanto ao fato de a classe NP ser, na verdade, igual à classe P conduziram à descoberta de toda uma classe de problemas, dentro da classe NP, conhecida como a classe dos **problemas NP-completos**, os quais têm a seguinte propriedade: a descoberta de uma solução polinomial para qualquer um deles fornecerá soluções polinomiais para todos os demais problemas em NP. Isto é, se conseguirmos achar um algoritmo (determinístico) capaz de resolver um dos problemas NP-completos em tempo polinomial, então aquele algoritmo poderá ser estendido para resolver qualquer outro problema NP em tempo polinomial. Com isso, a classe NP coincidiria com a classe P. O problema do caixeiro viajante é um exemplo de problema NP-completo.

Em resumo, concluímos que os problemas podem ser classificados como solúveis (têm uma solução algorítmica) ou insolúveis (não têm solução algorítmica), como mostra a Figura 11.12. Além disso, a

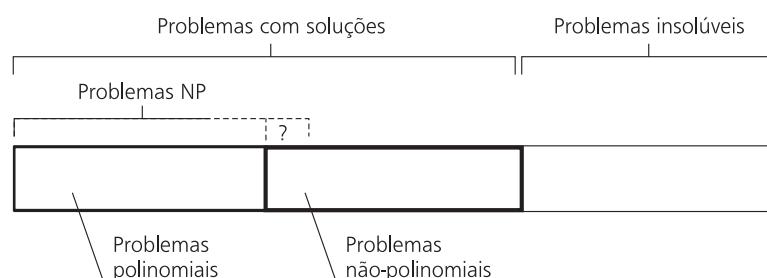


Figura 11.12 Um gráfico unificado da classificação de problemas.

classe de problemas solúveis apresenta duas subclasses. Uma delas compreende a coleção dos problemas polinomiais que se considera terem soluções práticas. A outra compreende os problemas não-polinomiais, cujas soluções são consideradas práticas apenas para dados de entrada relativamente pequenos, ou cuidadosamente selecionados. Finalmente, há aqueles problemas NP misteriosos que até agora fugiram de uma classificação precisa.



QUESTÕES/EXERCÍCIOS

1. Suponha que um problema possa ser resolvido por um algoritmo em $\Theta(2^n)$. O que podemos concluir quanto à sua complexidade?
2. Suponha que um problema possa ser resolvido por um algoritmo em $\Theta(n^2)$ e por outro em $\Theta(2^n)$. Um dos algoritmos será sempre superior ao outro?
3. Liste todos os subgrupos que podem ser formados com os elementos de um grupo de dois sócios, Alice e Bill. Liste todos os subgrupos que podem ser formados a partir do grupo composto por Alice, Bill e Carol, bem como a partir de um composto por Alice, Bill, Carol e David.
4. Dê um exemplo de um problema polinomial e um de um problema não-polinomial. Apresente um exemplo de um problema NP que ainda não tenha sido demonstrado como um problema polinomial.
5. Se a complexidade de um algoritmo X é maior que a de um algoritmo Y, o algoritmo X é necessariamente mais difícil de entender do que o algoritmo Y? Justifique a sua resposta.

11.6 Criptografia de chave pública

O fato de não haver uma solução eficiente conhecida para um problema NP-completo leva a muitas aplicações interessantes, e uma delas trata da criptografia de informação preciosa. Nesta seção, investigaremos essa técnica de criptografia. Ela envolve valores chamados chaves, que são usados para criptografar os dados e decifrar dados criptografados. Contudo, as chaves usadas para a criptografia de dados não são as mesmas que as chaves usadas para decifrar a informação criptografada. Quem conhece a chave para criptografar mensagens pode fazê-lo, mas não decifrar as mensagens que foram criptografadas por outros, mesmo que tenham usado a mesma chave. Assim, as chaves para criptografar podem ser amplamente divulgadas sem comprometer a segurança do sistema. Com tal sistema de criptografia, muitas pessoas diferentes enviam mensagens seguras para um mesmo destinatário — e esse destinatário comum é a única pessoa a ter as chaves para decifrar. Essas técnicas de criptografia compõem o campo de estudo conhecido como **criptografia de chave pública**, termos que refletem o fato de que as chaves usadas para criptografar as mensagens podem ser de domínio público.

Criptografia via problemas da mochila

Para descrever um sistema de criptografia de chave pública específico, começamos com o problema NP-completo conhecido como o **problema da mochila**^{*}. É o problema de selecionar números de uma coleção de tal maneira que a soma dos números selecionados seja um valor específico. Ele é chamado assim porque é análogo ao problema de selecionar uma coleção de itens que caibam exata-

*N. de T. Em inglês, *knapsack problems*.

Sistemas de criptografia populares

Um dos sistemas de criptografia mais populares usado por indivíduos para comunicação segura pela Internet é o PGP (Pretty Good Privacy), desenvolvido por Philip Zimmermann em 1991. O PGP é um sistema de criptografia de chave pública de propósito geral, fácil de usar, que está disponível

em diversas fontes na Internet, sem custo, desde que para uso não-comercial. O algoritmo no qual a criptografia de chave pública do PGP se baseia é o RSA, assim chamado em homenagem a seus criadores – Ron Rivest, Adi Shamir e Leonard Adleman. Enquanto o sistema de criptografia discutido no texto é baseado na dificuldade de resolver grandes problemas da mochila, o RSA é baseado na dificuldade de encontrar os fatores de grandes números inteiros.

O RSA é um exemplo de como as leis de patentes e as regulamentações governamentais afetam a distribuição de software. De fato, muitas batalhas judiciais por patentes foram travadas em relação ao RSA, e o governo dos EUA continua a limitar a sua distribuição. Você pode procurar na Web para aprender mais sobre essa epopeia dinâmica. Uma boa maneira de começar é procurar nos sítios que tratam de PGP e RSA.

mente em uma mochila. Um exemplo de problema da mochila é selecionar a coleção de valores da lista

191 691 573 337 365 730 651 493 177 354
cuja soma seja 2063.

O melhor método conhecido para resolver os problemas da mochila em geral é tentar todas as combinações possíveis até que uma solução seja encontrada. Entretanto, se existirem n valores onde deve ser feita a seleção, existirão 2^n diferentes combinações a testar.

Podemos usar os problemas da mochila baseados na lista acima para criptografar mensagens como segue: inicialmente representamos a mensagem como uma cadeia de bits, usando ASCII ou Unicode. Quebramos, então, essa cadeia em segmentos de 10 bits e representamos cada segmento por um único número, o qual é obtido somando-se os valores da lista que ocupam as posições correspondentes às posições dos 1s no segmento de dez bits. Por exemplo, o segmento de 10 bits 1001100001 seria representado pelo número 1247 (Figura 11.13). Isso porque os 1s no segmento são encontrados na primeira, quarta, quinta e décima posições e a soma dos valores correspondentes na lista (191, 337, 365 e 354) é 1247. Do mesmo modo, o padrão 0010011010 seria representado por 2131 (que é 573 + 730 + 651 + 177). Por sua vez, a mensagem 10011000010010011010 seria criptografada como a sequência dos dois números 1247 e 2131.

Suponha que alguém intercepte a mensagem criptografada e que essa pessoa conheça a lista de valores que foi usada para criptografá-la. Essa pessoa ainda teria de resolver dois problemas da mochila para decifrar a mensagem, o que seria um processo demorado. Além disso, se o tamanho da lista usada para criptografar as mensagens fosse significativamente maior do que 10, a tarefa de decifrar as mensagens interceptadas seria totalmente impraticável — ou seja, o conteúdo da mensagem estaria seguro.

O problema desse sistema simples é que ninguém estará habilitado a decifrar a mensagem rapidamente — nem mesmo o destinatário. O que precisamos é de um truque que permita a este

resolver os problemas da mochila rapidamente, restando a todos os demais uma tarefa demorada impraticável.

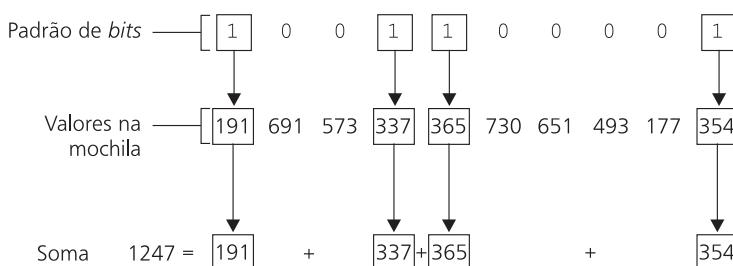


Figura 11.13 Criptografia de um padrão de bits como um problema da mochila.

Para obter tal truque, primeiro observamos que alguns problemas da mochila são fáceis de resolver. Suponha que os valores que devem ser selecionados sejam

1 4 6 12 24 51 105 210 421 850

Cada número nessa lista é maior do que a soma dos precedentes. Então, se precisarmos selecionar uma coleção de valores cuja soma seja 995, saberemos imediatamente que 850 deve ser um dos valores selecionados, porque a soma de todos os outros valores seria muito pequena. Após esta seleção, nosso problema fica reduzido a selecionar números cuja soma seria 995 – 850, ou 145. Entretanto, isso significa que devemos selecionar o 105, porque a soma dos outros valores possíveis seria menor que 145. Continuando dessa maneira, poderíamos concluir rapidamente que os valores a serem selecionados são 850, 105, 24, 12 e 4.

E agora, o truque — existe uma maneira de converter esses problemas da mochila fáceis em problemas da mochila difíceis e vice-versa. Por ora, consideremos esse processo de conversão em termos de três números “mágicos”. Mais tarde, veremos a origem desses números mágicos e como você pode construir o seu próprio sistema de criptografia. Os números mágicos que usaremos são 642, 2311 e 18.

Nosso primeiro passo é converter a lista

1 4 6 12 25 51 105 210 421 850

com a qual os problemas da mochila fáceis são construídos, em outra lista, cujos problemas da mochila são mais difíceis. Fazemos isso multiplicando cada elemento da lista por 642 (o primeiro número mágico), dividindo esse produto por 2311 (o segundo número mágico) e guardando os restos dessas divisões. Isso produz a lista

642 257 1541 771 2184 388 391 782 2206 304

O valor 4 na lista original é substituído por 257 na nova lista porque $4 \times 642 = 2568$ e $2568 \div 2311$ produz um resto de 257.

Observe que um problema da mochila posto em termos dessa nova lista seria difícil, uma vez que o nosso processo de conversão destruiu a relação que existia entre os valores da lista original. Entretanto, conhecendo os números mágicos, podemos resolver esses problemas rapidamente. Nossa abordagem é multiplicar a soma dada por 18 (o terceiro número mágico), dividir o produto por 2311 (o segundo número mágico) e guardar o resto dessa divisão. Usamos então esse resto como a soma para o problema da mochila posto em termos do sistema original fácil. Uma vez resolvido esse problema fácil, os valores da lista original que resolvem o problema da mochila original são os que se encontram nas posições correspondentes à seleção do problema da mochila fácil.

Por exemplo, suponha que o problema seja selecionar os valores da lista

642 257 1541 771 2184 388 391 782 2206 304

cuja soma seja 4895. Inicialmente, calculamos $4895 \times 18 = 88.110$, então dividimos 88.110 por 2311, obtendo o resto 292. Em seguida, determinamos que os valores 6, 25, 51 e 210 são os valores da lista

1 4 6 12 25 51 105 210 421 850

cuja soma é 292. Uma vez que eles são o terceiro, o quinto, o sexto e o oitavo valores em sua respectiva lista, concluímos que o terceiro, quinto, sexto e oitavo valores na lista

642 257 1541 771 2184 388 391 782 2206 304

são aqueles cuja soma é 4895. De fato, $1541 + 2184 + 388 + 782 = 4895$, como desejado.

O sistema completo de criptografia de chave pública funciona assim: distribuímos abertamente a lista

642 257 1541 771 2184 388 391 782 2206 304

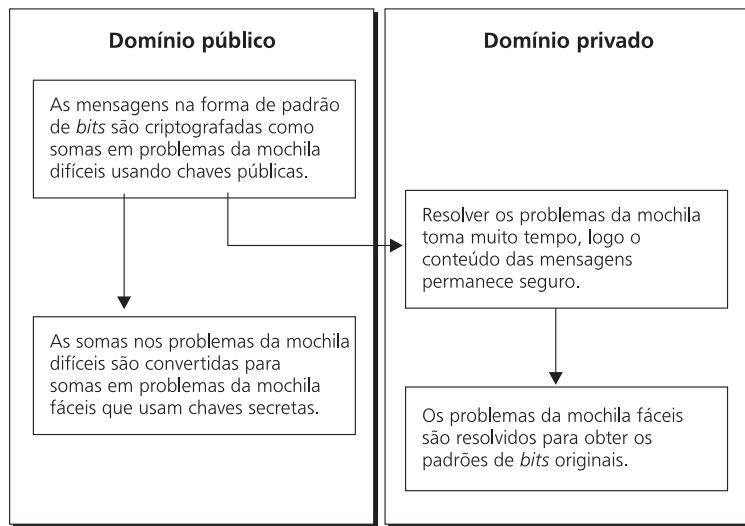


Figura 11.14 Criptografia de chave pública que usa problemas da mochila.

simplesmente um sistema obtido pela substituição de cada inteiro no sistema aritmético tradicional pelo resto que é obtido da divisão do inteiro por um valor predeterminado, chamado *módulo*. Por exemplo, se tomarmos 7 como módulo, então os valores inteiros

$$0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad \dots$$

seriam traduzidos para valores

$$0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 0 \quad 1 \quad 2 \quad \dots$$

É costume usar a notação $x \pmod{m}$, que é lido “ x módulo m ” ou algumas vezes apenas “ $x \text{ mod } m$ ” para representar o resto obtido quando o valor x é dividido por m . Assim, $9 \pmod{7}$ é 2, porque $9 \div 7$ produz resto 2. Do mesmo modo, $24 \pmod{7}$ é 3 porque $24 \div 7$ produz resto 3 e $5 \pmod{7}$ é 5 porque $5 \div 7$ produz resto 5.

Dois inteiros que produzem o mesmo resto quando divididos por m são denominados equivalentes módulo m . Assim, 16 e 23 são equivalentes módulo 7 porque $16 \pmod{7}$ é o mesmo que $23 \pmod{7}$. De fato, 16 e 23 produzem resto 2 quando divididos por 7. Costuma-se usar a notação $x \equiv y \pmod{m}$ — leia-se “ x é equivalente a y mod m ” — que significa que x é equivalente a y quando se usa o módulo m . Assim, $16 \equiv 23 \pmod{7}$.

Após a tradução dos valores inteiros tradicionais para um sistema modular usando m como módulo, ficamos apenas com os valores 0, 1, 2, 3, ..., $m - 1$. Podemos realizar operações aritméticas com esse conjunto restrito de valores primeiramente realizando a operação na aritmética tradicional e então traduzindo a resposta, digamos x , de volta à faixa restrita substituindo-o pelo valor $x \pmod{m}$. Assim, em um sistema modular baseado no módulo 7, estamos restritos aos valores 0, 1, 2, 3, 4, 5 e 6. A soma $2 + 6$ seria o valor 1, visto que $2 + 6 = 8$, que produz resto 1 quando dividido por 7. Além disso, o produto 2×6 seria 5, já que $2 \times 6 = 12$, que produz resto 5 quando dividido por 7.

A aritmética em um sistema modular é, portanto, uma reflexão distorcida da aritmética no sistema tradicional. É uma reflexão, no sentido em que, se $x \equiv a \pmod{m}$ e $y \equiv b \pmod{m}$, então $x + y \equiv a + b \pmod{m}$. Entretanto, também é uma distorção, no sentido em que as somas e os produtos não são os mesmos nos dois sistemas. Mais especificamente, o produto de dois números distintos pode ser 1 em um

e com isso permitimos às pessoas criptografar suas mensagens em termos dos problemas da mochila baseados nessa lista. Entretanto, mantemos consigo a lista original, bem como os três números secretos. Quando recebermos mensagens criptografadas, as decifraremos rapidamente pela conversão em problemas da mochila fáceis, mas ninguém mais poderá fazê-lo. Assim, as mensagens que nos forem enviadas estarão seguras (Figura 11.14).

Aritmética modular

O sistema de criptografia de chave pública acima descrito se apoia em um conceito matemático conhecido como aritmética modular, que é sim-

sistema modular, fenômeno que não ocorre no sistema tradicional de números inteiros. Por exemplo, no sistema modular baseado no módulo 7, temos $3 \times 5 = 1$ (uma vez que $3 \times 5 = 15$ e $15 \div 7$ produz resto 1).

Dois números que resultam no produto 1 são chamados inversos multiplicativos um do outro. No sistema tradicional, o número 3 não possui inverso multiplicativo. Em vez disso, o seu inverso multiplicativo tradicional, que é $1/3$, está fora do sistema de valores inteiros. No entanto, no sistema de inteiros módulo 7, o valor 3 possui um inverso multiplicativo que, como já vimos, é 5.

Quando um valor x tem um inverso multiplicativo no sistema modular baseado no módulo m ? Os matemáticos nos dizem que se x e m são dois inteiros positivos tais que $x < m$ e x e m são relativamente primos (ou seja, o único inteiro que divide ambos exatamente é 1), então o valor x terá um inverso multiplicativo no sistema modular baseado no módulo m . Por exemplo, 6 é menor do que 13, e os dois valores não possuem divisor comum além do 1. Assim, o 6 terá um inverso multiplicativo no sistema de módulo 13. De fato, seu inverso é 11, pois $6 \times 11 = 66$, que quando dividido por 13 produz resto 1, isto é, $6 \times 11 \equiv 1 \pmod{13}$.

O fato de alguns inteiros terem outros inteiros como seus inversos multiplicativos em um sistema modular pode parecer um fenômeno estranho à primeira vista. Por exemplo, uma vez que 6 e 11 são inversos multiplicativos em um sistema módulo 13, vemos que $6 \times 11 \times x = x$, para qualquer valor de x . Afinal de contas, $6 \times 11 \times x = (6 \times 11) \times x = 1 \times x = x$.

De volta à criptografia

Note que se o valor x é não-negativo e menor que o módulo m , então $x \pmod{m}$ é o próprio x . Isso significa que se realizarmos operações aritméticas cujos resultados tradicionais estejam na faixa de 0 até $m - 1$, então esses resultados serão os mesmos obtidos no sistema modular. Assim, se tomarmos um módulo extremamente grande, poderemos realizar nossas computações aritméticas usuais sem saber sequer se estamos trabalhando no sistema aritmético tradicional ou em um sistema modular. Por exemplo, uma vez que a soma de todos os valores na lista

$$1 \quad 4 \quad 6 \quad 12 \quad 25 \quad 51 \quad 105 \quad 210 \quad 421 \quad 850$$

é 1685, as adições realizadas quando se tenta resolver um problema da mochila baseado nessa lista jamais produzirão resultados maiores que 1685. Quando resolvemos esses problemas, não precisamos nos preocupar se estamos trabalhando no sistema aritmético tradicional ou em um sistema modular cujo módulo é maior que 1685.

Entretanto, se pretendemos que o nosso problema da mochila fácil seja posto em um sistema modular com grande módulo, podemos obter um método de convertê-lo em um problema mais difícil e trazê-lo de volta novamente. Para esclarecer, suponha que tenhamos uma lista de números

$$a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6 \quad a_7 \quad a_8 \quad a_9 \quad a_{10}$$

tal que cada elemento na lista seja maior que a soma de seus antecedentes. Ela é, portanto, uma lista em cujos termos os problemas da mochila podem ser facilmente resolvidos. Tomemos um módulo m que seja maior que a soma de todos os valores na lista e dois outros valores x e y que sejam inversos multiplicativos no sistema modular baseado em m .

Se multiplicarmos cada elemento na lista original por x , obteremos a lista

$$a_1x \quad a_2x \quad a_3x \quad a_4x \quad a_5x \quad a_6x \quad a_7x \quad a_8x \quad a_9x \quad a_{10}x$$

em cujos termos os problemas da mochila novamente são resolvidos com facilidade. (Cada elemento continua sendo maior que a soma de seus antecedentes.) Entretanto, agora vamos substituir cada elemento em nossa nova lista por um valor equivalente a ele mod m . Especificamente, no lugar de a_1x , colocaremos o valor $a_1x \pmod{m}$; no lugar de a_2x , o valor $a_2x \pmod{m}$ e assim por diante. Isso produzirá uma nova lista

$$b_1 \quad b_2 \quad b_3 \quad b_4 \quad b_5 \quad b_6 \quad b_7 \quad b_8 \quad b_9 \quad b_{10}$$

onde cada elemento é equivalente mod m ao elemento correspondente na lista

$$a_1x \quad a_2x \quad a_3x \quad a_4x \quad a_5x \quad a_6x \quad a_7x \quad a_8x \quad a_9x \quad a_{10}x$$

Por sua vez, qualquer soma de valores dessa nova lista será equivalente módulo m à soma dos valores correspondentes na lista

$$a_1x \quad a_2x \quad a_3x \quad a_4x \quad a_5x \quad a_6x \quad a_7x \quad a_8x \quad a_9x \quad a_{10}x$$

Suponha então que nos seja dada uma soma tal como $b_1 + b_3 + b_5$ e pedido para selecionar os elementos da lista

$$b_1 \quad b_2 \quad b_3 \quad b_4 \quad b_5 \quad b_6 \quad b_7 \quad b_8 \quad b_9 \quad b_{10}$$

que produzem aquela soma. Uma vez que

$$b_1 + b_3 + b_5 \equiv (a_1x + a_3x + a_5x) \pmod{m}$$

e y é inverso multiplicativo de x , sabemos que

$$\begin{aligned} y(b_1 + b_3 + b_5) &\equiv y(a_1x + a_3x + a_5x) \pmod{m} \\ &\equiv yx(a_1 + a_3 + a_5) \pmod{m} \\ &\equiv (a_1 + a_3 + a_5) \pmod{m} \end{aligned}$$

Isso significa que se multiplicarmos a soma dos valores selecionados na lista

$$b_1 \quad b_2 \quad b_3 \quad b_4 \quad b_5 \quad b_6 \quad b_7 \quad b_8 \quad b_9 \quad b_{10}$$

por y , dividirmos o produto por m e guardarmos o resto, este será a soma dos elementos correspondentes na lista original

$$a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6 \quad a_7 \quad a_8 \quad a_9 \quad a_{10}$$

Contudo, uma vez que os problemas da mochila são facilmente resolvidos em termos desta lista, podemos rapidamente descobrir que elementos são esses. Por sua vez, podemos resolver o problema da mochila selecionando os elementos correspondentes na lista

$$b_1 \quad b_2 \quad b_3 \quad b_4 \quad b_5 \quad b_6 \quad b_7 \quad b_8 \quad b_9 \quad b_{10}$$

Em suma, para selecionar os valores da lista

$$b_1 \quad b_2 \quad b_3 \quad b_4 \quad b_5 \quad b_6 \quad b_7 \quad b_8 \quad b_9 \quad b_{10}$$

usados para formar a soma s , precisamos simplesmente calcular o valor $s \times y \pmod{m}$, encontrar os elementos na lista

$$a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6 \quad a_7 \quad a_8 \quad a_9 \quad a_{10}$$

cuja soma seja esse valor e então selecionar os valores correspondentes na lista

$$b_1 \quad b_2 \quad b_3 \quad b_4 \quad b_5 \quad b_6 \quad b_7 \quad b_8 \quad b_9 \quad b_{10}$$

Como exemplo, começemos com a lista

$$1 \quad 4 \quad 6 \quad 12 \quad 25 \quad 51 \quad 105 \quad 210 \quad 421 \quad 850$$

em cujos termos os problemas da mochila são facilmente resolvidos. Uma vez que a soma de todos os valores na lista é 1685, o valor 2311 é suficientemente grande para fazer o papel de m . Além disso, 642 e 18 são inversos multiplicativos no sistema modular que utiliza módulo 2311, então usamos 642 como x e 18 como y . Nossa primeiro passo é multiplicar cada elemento na lista acima por 642 e guardar o resto obtido na divisão desse produto por 2311. Isso produz a lista

$$642 \quad 257 \quad 1541 \quad 771 \quad 2184 \quad 388 \quad 391 \quad 782 \quad 2206 \quad 304$$

Suponha que agora nos seja dado o problema de selecionar os valores nessa lista cuja soma seja 4507. Multiplicamos 4507 por 18 para obter 81.126, dividimos esse valor por 2311 e guardamos o resto, que é 241. Então, vemos que os valores 6, 25 e 210 são os números na lista original cuja soma é 241. Uma vez que eles são o terceiro, quinto e oitavo elementos em sua respectiva lista, concluímos que o terceiro, quinto e oitavo elementos na lista

642 257 1541 771 2184 388 391 782 2206 304

são aqueles cuja soma é 4507. De fato, os valores 1541, 2184 e 782 resolvem o problema da mochila original.

Resumindo, podemos construir um sistema de criptografia de chave pública, como mostrado na Figura 11.15. Primeiro escrevemos uma lista de valores a partir da qual os problemas da mochila fáceis são construídos. Em seguida, tomamos os valores m , x e y de forma que m seja maior que a soma de todos os valores na lista e x seja inverso multiplicativo de y no sistema de módulo m . Multiplicamos então os valores na lista original por x , dividimos esses produtos por m e guardamos os restos. A lista desses restos é a chave pública de criptografia. Qualquer pessoa pode criptografar uma mensagem como uma seqüência de problemas da mochila baseados nessa lista, e seremos os únicos a conseguir decifrar essas mensagens facilmente. Precisamos simplesmente multiplicar cada soma dada por y , dividir o produto por m e guardar o resto. Então, encontraremos rapidamente os valores na lista original cuja soma é aquele resto e reconstruiremos os padrões de bits que formaram a mensagem.

Devemos fazer um comentário final. Um adversário poderia tentar quebrar o nosso sistema de criptografia adivinhando os valores m , x e y , em vez de resolver os problemas da mochila difíceis. Esse é o motivo pelo qual os números usados em um sistema de criptografia real são muito maiores que os valores usados em nossos exemplos. Selecionando-se grandes problemas da mochila e grandes valores para as chaves, o tempo necessário para resolver os problemas da mochila ou para adivinhar as chaves privadas pode se tornar significativo.

1. Selecionar os valores para os quais os problemas da mochila são facilmente resolvidos.

Exemplo: 2 5 8 17

2. Selecionar três números m , x e y , tais que m seja maior que a soma dos valores da mochila e x seja inverso multiplicativo de y no sistema de módulo m .

Exemplo: $m = 37$, $x = 25$, $y = 3$

3. Substituir cada valor a na lista original por $xa \pmod{m}$.

Exemplo:

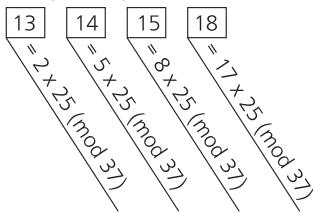


Figura 11.15 Construção de um sistema de criptografia de chave pública.



QUESTÕES/EXERCÍCIOS

1. Encontre os valores na lista abaixo que produzem a soma 2200.
191 691 573 337 365 730 651 493 177 354
(Não perca muito tempo nesse exercício, ele pode ser demorado.)
2. Encontre os valores na lista abaixo que produzem a soma 3223.
642 257 1541 771 2184 388 391 782 2206 304
Ao contrário da questão anterior, esta não deve levar muito tempo. Por quê?
3. Encontre o inverso multiplicativo de 5 no sistema modular cujo módulo é 23.
4. Projete um sistema de criptografia de chave pública baseado na lista
2 3 6 12 24
e no fato de 30 e 38 serem inversos multiplicativos em um sistema modular com módulo 67.

Problemas de revisão de capítulo

(Os problemas marcados com asterisco se referem às seções opcionais.)

- 1.** Mostre como uma estrutura da forma

```
while x equals 0 do;
```

```
.
```

```
.
```

```
end;
```

pode ser simulada com a linguagem Bare Bones.

- 2.** Escreva um programa em Bare Bones que coloque 1 na variável Z se a variável X for menor ou igual à variável Y e coloque 0 na variável Z se for maior.
- 3.** Escreva um programa em Bare Bones que coloque a potência de 2 elevado a X na variável Z.
- 4.** Nos seguintes casos, escreva uma sequência de instruções em Bare Bones que execute a ação indicada:
- Atribua 0 a Z se o valor de X for par, caso contrário atribua 1 a Z.
 - Calcule a soma dos inteiros de 0 a X.
- 5.** Escreva uma rotina, na linguagem Bare Bones, que divida o valor de X pelo de Y. Desconsidere qualquer resto, isto é, 1 dividido por 2 produz 0, e 5 dividido por 3 produz 1.
- 6.** Descreva a função computada pelo seguinte programa em Bare Bones, pressupondo que as suas entradas estejam representadas por X e Y e sua saída, por Z.

```
copy X to Z;
copy Y to Aux;
while Aux not 0 do;
    decr Z;
    decr Aux;
end;
```

- 7.** Descreva a função computada pelo seguinte programa em Bare Bones, pressupondo que as suas entradas estejam representadas por X e Y e sua saída, por Z.

```
clear Z;
copy X to Aux1;
copy Y to Aux2;
while Aux1 not 0 do;
    while Aux2 not 0 do;
        decr Z;
        decr Aux2;
```

```
end;
decr Aux1;
end;
```

- 8.** Escreva um programa em Bare Bones que compute o ou exclusivo das variáveis X e Y, colocando o resultado na variável Z. Você pode assumir que X e Y começem apenas com os valores inteiros 0 e 1.

- 9.** Mostre que se permitirmos que as instruções em um programa Bare Bones sejam rotuladas com valores inteiros e se substituirmos a estrutura de laço while por um desvio condicional representado pela forma

```
if nome not 0 goto rotulo;
```

onde nome é qualquer variável e rotulo é um valor inteiro usado em algum lugar para identificar uma instrução, então a nova linguagem continuará a ser uma linguagem de programação universal.

- 10.** Neste capítulo, vimos como a instrução

```
copy nome1 to nome2;
```

poderia ser simulada em Bare Bones. Mostre como essa instrução também poderia ser simulada se a estrutura de laço while fosse substituída por um laço pós-teste expresso na forma

```
repeat... until(nome equals 0)
```

- 11.** Mostre que a linguagem Bare Bones continuaria sendo universal se a instrução while fosse substituída por um laço pós-teste expresso na forma

```
repeat... until(nome equals 0)
```

- 12.** Projete uma máquina de Turing que, uma vez iniciada, não usará mais que uma célula em sua fita, mas nunca atingirá o seu estado de parada.

- 13.** Projete uma máquina de Turing que coloque 0s em todas as células à esquerda da célula corrente, até chegar a uma célula com um asterisco.

- 14.** Suponha que um padrão de 0s e 1s da fita de uma máquina de Turing seja delimitado por asteriscos em suas extremidades. Projete uma máquina de Turing que gire este padrão uma

célula para a esquerda, admitindo que a máquina inicie o seu processamento considerando como célula corrente aquela que contém o asterisco da extremidade direita do padrão.

15. Projete uma máquina de Turing que inverta o padrão de 0s e 1s encontrado entre a célula corrente (que contém um asterisco) e o primeiro asterisco à esquerda.
16. Resuma a tese de Church-Turing.
17. O seguinte programa em Bare Bones termina por si mesmo? Justifique a sua resposta.

```
copy X to Y;
incr Y;
incr Y;
while X not 0 do;
    decr X;
    decr X;
    decr Y;
    decr Y;
end;
decr Y;
while Y not 0 do;
    incr X;
    decr Y;
end;
while X not 0 do;
end;
```

18. O seguinte programa em linguagem Bare Bones termina por si mesmo? Justifique a sua resposta.

```
while X not 0 do;
end;
```

19. O seguinte programa em linguagem Bare Bones termina por si mesmo? Justifique a sua resposta.

```
while X not 0 do;
    decr X;
end;
```

20. Analise a validade das seguintes afirmações:

A próxima afirmação é verdadeira.
A afirmação acima é falsa.

21. Analise a validade da afirmação *O cozinheiro de um navio cozinha para todos aqueles que não cozinham para si mesmos, e somente para eles.* (Quem cozinha para o cozinheiro?)

22. Suponha que você esteja em um país onde cada pessoa ou diz a verdade ou é mentirosa. (No primeiro caso, ela sempre diz a verdade e no

segundo, sempre mente.) Que pergunta você faria para descobrir se uma pessoa diz a verdade ou se é mentirosa?

23. Resuma o significado das máquinas de Turing, no campo teórico da Ciência da Computação.
24. Resuma o significado do problema da parada, no campo teórico da Ciência da Computação.
25. Suponha que você queira saber se alguém em um grupo de pessoas faz aniversário em uma data específica. Uma abordagem seria perguntar aos membros do grupo, um de cada vez. Se você seguisse essa abordagem, a ocorrência de que evento garantiria a existência de tal pessoa? Que evento garantiria que essa pessoa não existe no grupo? Agora, suponha que queira descobrir se ao menos um número inteiro positivo possui uma propriedade particular e que você tenha aplicado a mesma abordagem de sistematicamente testar os inteiros um a um. Se, de fato, algum inteiro possuir tal propriedade, como você descobrirá? Se, porém, nenhum inteiro tiver a propriedade, como você ficará sabendo? A tarefa de testar para ver se uma conjectura é verdadeira é necessariamente simétrica em relação à tarefa de testar pra ver se ela é falsa?
26. O problema da busca de um elemento em uma lista é um problema polinomial? Justifique sua resposta.
27. Projete um algoritmo para decidir se um determinado inteiro positivo é primo. A sua solução é eficiente? Ela é polinomial ou não-polinomial?
28. Uma solução polinomial é sempre melhor do que uma exponencial para um problema? Explique a sua resposta.
29. O fato de um problema ter uma solução polinomial significa que sempre poderá ser resolvido em um intervalo de tempo tolerável? Explique a sua resposta.
30. Dado o problema de dividir um grupo (com um número par de pessoas) em dois subgrupos disjuntos, de tamanhos iguais, de forma que a diferença entre o total de idade de um subgrupo e o do outro seja o máximo possível, o programador Charlie propõe como solução formar todos os pares possíveis de subgrupos, calcular a diferença entre as somas das idades em cada par de subgrupos e selecionar como solução o par de subgrupos que apresentar a maior diferença. Entretanto, a programadora Mary propõe que o

grupo original seja primeiro ordenado por idade e depois dividido em dois subgrupos, um deles composto pela metade das pessoas mais jovens do grupo ordenado e o outro pelas mais idosas. Qual a complexidade de cada uma destas soluções? O problema é, em si, de complexidade polinomial, NP, ou de complexidade não-polinomial?

31. Por que a abordagem de gerar todos os possíveis arranjos de uma lista e então tomar aquele com a disposição adequada não é uma maneira satisfatória de ordenar a lista?
32. Suponha que uma loteria seja baseada na extração correta de quatro valores inteiros, cada um deles na faixa de 1 a 50. Além disso, suponha que o volume de apostas tenha ficado tão grande que se tornou vantajoso comprar um bilhete em separado para cada combinação possível. Se a compra de um único bilhete leva um segundo, quanto demorará para comprar um bilhete para cada combinação? Como a necessidade de tempo mudaria se a loteria resolvesse usar cinco números em vez de quatro no processo de extração? O que este problema tem a ver com o conteúdo aprendido neste capítulo?
33. O seguinte algoritmo é determinístico? Explique sua resposta.

```
procedure misterio (Numero)
if (Numero > 5)
    then (responder "sim")
    else (escolher um valor menor que 5 e
          fornecer este número como resposta)
```

34. O seguinte algoritmo é determinístico? Explique sua resposta.

Dirigir em linha reta, para a frente.
No terceiro cruzamento, perguntar à pessoa que se encontra na esquina se você deve fazer uma conversão à direita ou à esquerda.
Seguir as instruções dessa pessoa.
Dirigir por mais duas quadras e parar ali.

35. Identifique os pontos de não-determinismo no seguinte algoritmo:
Selecionar três números, entre 1 e 100.
If (a soma dos números selecionados for maior que 150)
then (responder "sim")

else (selecionar um dos números escolhidos e fornecê-lo como resposta)

36. O algoritmo abaixo tem uma complexidade polinomial ou não-polinomial? Explique sua resposta.
- ```
procedure misterio (ListaDeNumeros)
Escolher um conjunto de números de
ListaDeNumeros
if (a soma dos números deste conjunto for 125)
then (responder "sim")
else (não fornecer resposta)
```
37. Quais dos seguintes problemas estão na classe P?
    - a. Um problema com complexidade  $n^2$
    - b. Um problema com complexidade  $3n$
    - c. Um problema com complexidade  $n^2 + 2n$
    - d. Um problema com complexidade  $n!$  38. Resuma a diferença entre afirmar que um problema é polinomial e afirmar que é polinomial não-determinístico.
  39. Dê um exemplo de um problema que pertença tanto à classe P como à NP.
  40. Suponha que sejam dados dois algoritmos para solucionar um mesmo problema. Um deles tem complexidade de tempo igual a  $n^4$  e o outro, igual a  $4^n$ . Para quais tamanhos de entrada o primeiro algoritmo é mais eficiente que o segundo?
  41. Suponha que você deva resolver o problema do caixeiro viajante em um contexto que envolva 15 cidades, no qual quaisquer duas cidades estão ligadas por uma única estrada. Quantos caminhos diferentes que passam pelas cidades existem? Quanto tempo seria necessário para calcular o comprimento de todos caminhos assumindo que o comprimento de um possa ser calculado em um microsegundo.
  42. Quantas comparações de nomes serão feitas se o algoritmo de ordenação por intercalação (Figuras 11.8 e 11.9) for aplicado à lista Alice, Bob, Carol e David? E quantas comparações serão se a lista for Alice, Bob, Carol, David e Elaine?
  43. Dê um exemplo de um problema em cada categoria representada na Figura 11.12.
  44. Projete um algoritmo para encontrar soluções inteiras para equações da forma  $x^2 + y^2 = n$ ,

onde  $n$  é um inteiro positivo. Determine a complexidade de tempo do seu algoritmo.

45. Identifique as semelhanças entre o problema do caixeteiro viajante e os da mochila.
46. O seguinte algoritmo de ordenação de uma lista é chamado *Bubble Sort*. Utilizando este algoritmo para uma lista de  $n$  elementos, quantas comparações serão necessárias?

```
procedure BubbleSort (Lista)
Contador ← 1;
while (Contador < número de elementos em
Lista) do
[N ← número de elementos em Lista;
while (N > 1) do
(if (N-ésimo elemento de Lista é menor
que o elemento anterior)
then (intercambiar o N-ésimo elemento e
o anterior)
N ← N – 1
)
]
```

47. Quanto tempo levaria para testar todas as combinações possíveis quando se resolve um problema da mochila que envolve 40 valores, se cada teste leva um microsegundo?

48. Por que seria mais fácil resolver os problemas da mochila baseados nos valores  
1 2 4 8 16 32 64 128 256 512 1028  
do que os baseados nos valores  
191 691 573 337 365 730 651 493 177 354
49. Encontre os valores na lista abaixo que produzem a soma 3212.  
642 257 1541 771 2184 388 391 782 2206 304  
(Note que esses são os valores usados no sistema de criptografia de chave pública desenvolvida na Seção 11.6.)
50. Encontre o inverso multiplicativo de 5 no sistema modular cujo módulo é 8 e o inverso multiplicativo de 3 no sistema modular cujo módulo é 8.
51. Projete um sistema de criptografia de chave pública baseado na lista  
1 3 5 10 20  
e no fato de 30 e 38 serem inversos  
multiplicativos em um sistema modular com  
módulo 67.
52. No sistema modular de módulo 9, que valores  
possuem inversos multiplicativos e quais são  
esses inversos?

## Questões sociais

As seguintes questões procuram auxiliar o leitor no entendimento de alguns assuntos éticos, sociais e legais no campo da computação. O objetivo não é meramente o de fornecer respostas a tais questões. O leitor também deve justificá-las e verificar se as justificativas apresentadas preservam sua consistência de uma questão para outra.

1. Suponha que o melhor algoritmo para resolver um problema leve 100 anos de execução. Você consideraria tal problema tratável? Por quê?
2. Os cidadãos têm o direito de criptografar suas mensagens, de modo a impedir o monitoramento pelas agências governamentais? A sua resposta está de acordo com as exigências “apropriadas” da lei? Quem deve decidir o que é exigência “apropriada” da lei?
3. Se a mente humana fosse um dispositivo algorítmico, quais consequências a tese de Turing teria sobre a humanidade? Até que ponto você acredita que as máquinas de Turing abrangem as capacidades computacionais da mente humana?
4. Vimos que existem diferentes modelos computacionais (tabela finita, fórmulas algébricas, máquinas de Turing etc.) com capacidades computacionais distintas. Existem diferenças nas capacidades computacionais de organismos distintos? E de pessoas distintas? Em caso afirmativo, as pessoas com maior capacidade estão aptas a usar isso para obter melhor estilo de vida?

5. Atualmente, existem sítios na Web que fornecem mapas de estradas para a maioria das cidades. Esses sítios ajudam a encontrar endereços específicos e fornecem capacidade de zoom para ver os detalhes da vizinhança. Partindo dessa realidade, considere a seguinte sequência fictícia. Suponha que esses sítios tenham sido aperfeiçoados com fotografias de satélite com capacidades de zoom similares, as quais foram aumentadas para dar uma imagem mais detalhada das construções individuais e dos terrenos ao seu redor. Suponha também que as imagens tenham sido melhoradas para incluir vídeo em tempo real e que essas imagens foram aperfeiçoadas com a tecnologia de infravermelho. Nesse ponto, outras pessoas podem observá-lo em sua própria casa 24 horas por dia. Em que ponto dessa progressão os seus direitos de privacidade forem primeiramente violados? Em que ponto dessa progressão você acredita que fomos além das capacidades da tecnologia atual de espionagem por satélite? Até onde esse cenário é fictício?
6. Suponha que uma companhia desenvolva e obtenha patente de um sistema de criptografia. O governo nacional onde a companhia está instalada tem direito de usar o sistema se ele for considerado adequado, em nome da segurança nacional? Ele tem o direito de restringir o uso comercial do sistema em nome da segurança nacional? E se a companhia for uma organização multinacional?
7. Suponha que você compre um produto cuja estrutura interna seja codificada. Você tem o direito de decifrar a estrutura subjacente? Em caso afirmativo, você tem o direito de usar essa informação para fins comerciais? E fins não-comerciais? E se a codificação tiver sido feita usando um sistema secreto de criptografia e você descobrir o segredo? Você tem o direito de compartilhar o segredo?
8. Alguns anos atrás, o filósofo John Dewey (1859-1952) introduziu o termo *tecnologia responsável*. Dê alguns exemplos do que você consideraria ser tecnologia responsável. Baseado em seus exemplos, formule a sua própria definição da mesma. A sociedade tem praticado a tecnologia responsável ao longo dos últimos 100 anos? Ações devem ser tomadas para garantir que ela a pratique? Em caso afirmativo, quais ações? Senão, por quê?

## *Leituras adicionais*

- Garey, M. R. and D. S. Johnson. *Computers and Intractability*. New York: W. H. Freeman, 1979.
- Hofstadter, D. R. *Gödel, Escher, Bach: An Eternal Golden Braid*. St. Paul, MN: Vintage, 1980.
- Hopcroft, J. E., R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computations*, 2nd ed. Boston: Addison-Wesley, 2001.
- Lewis, H. R. and C. H. Papadimitriou. *Elements of the Theory of Computation*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1998.
- Sipser, M. *Introduction to the Theory of Computation*. Boston: PWS, 1996.
- Smith, C. and E. Kimber. *Theory of Computing: A Gentle Introduction*. Englewood Cliffs, NJ: Prentice-Hall, 2000.

## A PÊNDICES

---

- A ASCII
- B Circuitos para manipular representações em complemento de dois
- C Uma linguagem de máquina simples
- D Exemplos de programas em linguagens de alto nível
- E A equivalência entre estruturas iterativas e recursivas
- F Respostas das questões e dos exercícios



# A PÊNDICE A

---

## ASCII

A seguir, uma lista parcial do código ASCII, na qual cada padrão de *bits* foi estendido com um 0 à esquerda para apresentar o padrão de oito *bits* geralmente utilizado na atualidade.

| Símbolo          | ASCII    | Símbolo | ASCII    | Símbolo | ASCII    |
|------------------|----------|---------|----------|---------|----------|
| nova linha       | 00001010 | >       | 00111110 | ^       | 01011110 |
| retorno de carro | 00001101 | ?       | 00111111 | -       | 01011111 |
| espaço           | 00100000 | @       | 01000000 | a       | 01100001 |
| !                | 00100001 | A       | 01000001 | b       | 01100010 |
| "                | 00100010 | B       | 01000010 | c       | 01100011 |
| #                | 00100011 | C       | 01000011 | d       | 01100100 |
| \$               | 00100100 | D       | 01000100 | e       | 01100101 |
| %                | 00100101 | E       | 01000101 | f       | 01100110 |
| &                | 00100110 | F       | 01000110 | g       | 01100111 |
| '                | 00100111 | G       | 01000111 | h       | 01101000 |
| (                | 00101000 | H       | 01001000 | i       | 01101001 |
| )                | 00101001 | I       | 01001001 | j       | 01101010 |
| *                | 00101010 | J       | 01001010 | k       | 01101011 |
| +                | 00101011 | K       | 01001011 | l       | 01101100 |
| ,                | 00101100 | L       | 01001100 | m       | 01101101 |
| -                | 00101101 | M       | 01001101 | n       | 01101110 |
| .                | 00101110 | N       | 01001110 | o       | 01101111 |
| /                | 00101111 | O       | 01001111 | p       | 01110000 |
| 0                | 00110000 | P       | 01010000 | q       | 01110001 |
| 1                | 00110001 | Q       | 01010001 | r       | 01110010 |
| 2                | 00110010 | R       | 01010010 | s       | 01110011 |
| 3                | 00110011 | S       | 01010011 | t       | 01110100 |
| 4                | 00110100 | T       | 01010100 | u       | 01110101 |
| 5                | 00110101 | U       | 01010101 | v       | 01110110 |
| 6                | 00110110 | V       | 01010110 | w       | 01110111 |
| 7                | 00110111 | W       | 01010111 | x       | 01111000 |
| 8                | 00111000 | X       | 01011000 | y       | 01111001 |
| 9                | 00111001 | Y       | 01011001 | z       | 01111010 |
| :                | 00111010 | Z       | 01011010 | {       | 01111011 |
| ;                | 00111011 | [       | 01011011 | }       | 01111101 |
| <                | 00111100 | \       | 01011100 |         |          |
| =                | 00111101 | ]       | 01011101 |         |          |



# A PÊNDICE B

## *Circuitos para manipular representações em complemento de dois*

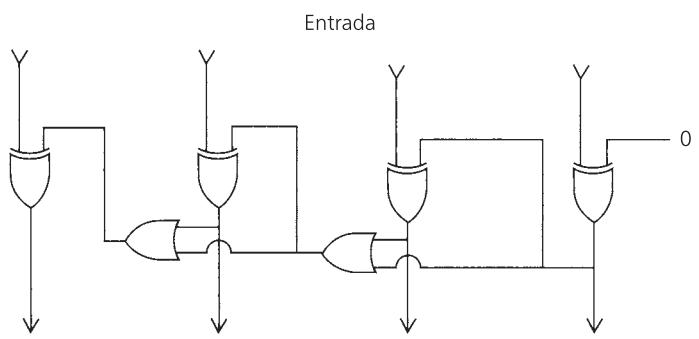
Este apêndice apresenta circuitos capazes de trocar o sinal e de somar valores representados na notação de complemento de dois. Iniciamos com o circuito da Figura B.1, que converte uma representação, em complemento de dois, de um padrão de 4 bits para a representação do negativo desse valor. Por exemplo, dada a representação em complemento de dois do número 3, o circuito produz a representação do número -3. Ele executa essa tarefa seguindo o mesmo algoritmo apresentado no texto, isto é, copia o padrão da direita para a esquerda até que um bit 1 seja copiado, e então complementa cada bit restante à medida que o transfere da entrada para a saída. Dado que uma entrada da porta lógica XOR mais à direita está fixada em 0, esta porta simplesmente transferirá a sua entrada para a saída. Contudo, esta saída também é passada para a esquerda, como uma das entradas para a próxima porta XOR. Se esta saída for 1, a próxima porta XOR complementará seu bit de entrada quando este passar para a saída. Além disso, este 1 também será passado para a esquerda, através da porta OR, para influir na saída da próxima porta. Dessa maneira, o primeiro 1 que foi copiado para a saída também será passado para a esquerda, onde fará com que todos os bits restantes sejam complementados, à medida que forem sendo transferidos para a saída.

A seguir, vamos considerar um método para somar dois valores representados em complemento de dois. Por exemplo, ao fazer a conta

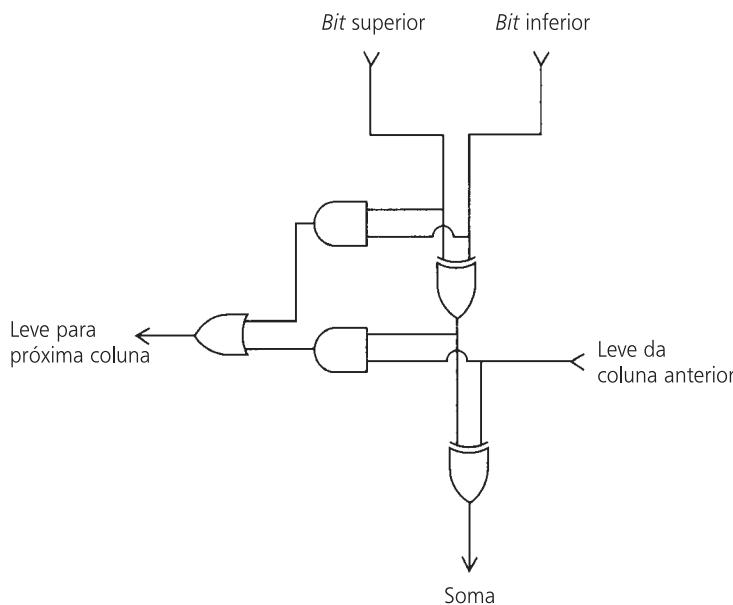
$$\begin{array}{r} 0110 \\ + 1011 \\ \hline \end{array}$$

procedemos da direita para a esquerda, coluna a coluna, executando o mesmo algoritmo para cada uma. Assim, obtido um circuito capaz de somar uma coluna neste problema, poderemos construir um circuito destinado a somar muitas, por meio da simples repetição daquele utilizado para somar uma única coluna.

O algoritmo para somar uma única coluna em um problema de adição de várias colunas consiste em so-



**FIGURA B.1** Circuito que troca o sinal de um padrão binário em complemento de dois.



**FIGURA B.2** Circuito de adição em uma única coluna em um problema de adição de várias colunas.

duzir um circuito que calcule a soma de dois valores representados em complemento de dois, com quatro bits. Cada retângulo representa uma cópia do circuito de adição de uma coluna. Note-se que o valor do transporte fornecido ao retângulo mais à direita é sempre 0, porque não há transporte proveniente de coluna anterior. Do mesmo modo, o transporte produzido pelo retângulo mais à esquerda é ignorado.

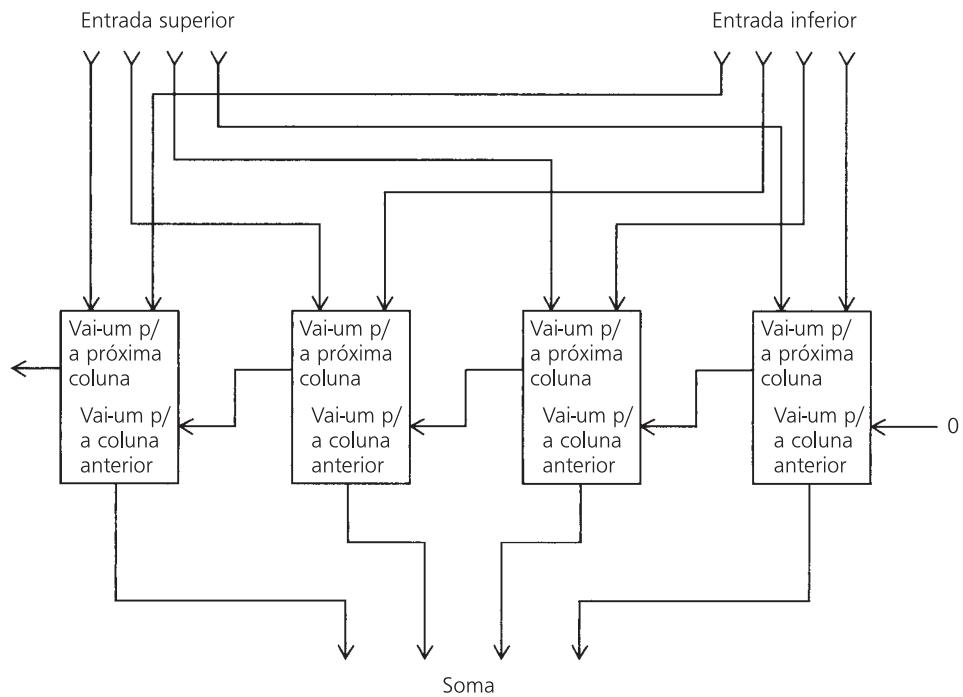
O circuito na Figura B.3 é conhecido como somador com ondulação\* porque a informação de transporte deve se propagar, ou ondular, da coluna mais à direita para a mais à esquerda. Embora sejam simples de compreender, tais circuitos são mais lentos na realização de suas tarefas do que as versões mais inteligentes, como o somador com previsão de transporte\*\*, que minimiza a propagação de coluna a coluna. Assim, o circuito na Figura B.3, embora suficiente para os nossos objetivos, não é usado nas máquinas atuais.

mar os dois valores contidos na coluna em questão, adicionar esta soma ao bit correspondente ao transporte (vai-um) proveniente da coluna anterior, escrever o bit menos significativo desta soma como resposta e transferir o transporte obtido para a próxima coluna. O circuito da Figura B.2 segue exatamente esse algoritmo. A porta XOR superior determina a soma dos dois bits de entrada e a inferior adiciona esta soma ao valor transportado da coluna anterior. As duas portas AND, juntamente com a porta OR, passam para a esquerda o transporte, se houver. Em particular, um vai-um será produzido se os dois bits originais nesta coluna forem 1, ou se a soma desses bits e o valor do transporte forem 1.

A Figura B.3 mostra como réplicas deste circuito de uma coluna podem ser utilizadas para pro-

\*N. de T. Em inglês, *ripple adder*.

\*\*N. de T. Em inglês, *lookahead carry adder*.



**FIGURA B.3** Um circuito para somar dois valores em complemento de dois, que utiliza quatro réplicas do circuito da Figura B.2.



## A PÊNDICE

# C

# *Uma linguagem de máquina simples*

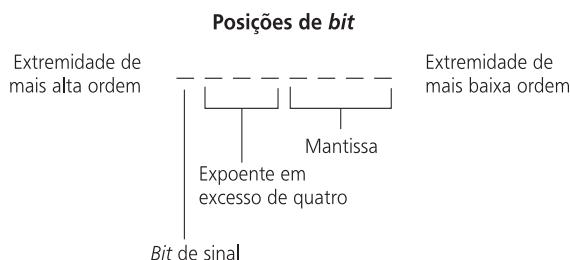
Neste apêndice, apresentamos uma linguagem de máquina simples, porém representativa. Iniciamos explicando a arquitetura da máquina.

## Arquitetura da máquina

A máquina tem 16 registradores de propósito geral, numerados de 0 a F (em hexadecimal). Cada registrador tem um *byte* de comprimento (oito bits). Para indicar os registradores nas instruções, cada registrador é associado univocamente a um padrão de quatro bits, que representa o número do registrador. Assim, o registrador 0 é identificado por 0000 (hexadecimal 0) e o 4, por 0100 (hexadecimal 4).

Existem 256 células na memória principal da máquina. A cada uma é atribuído um único endereço, que consiste em um número inteiro na faixa de 0 a 255. Um endereço pode ser, portanto, representado por um padrão de oito bits, que varia de 00000000 até 11111111 (ou um valor hexadecimal, no intervalo de 00 a FF).

Os valores de vírgula flutuante são armazenados no formato mostrado a seguir:



## Linguagem de máquina

Cada instrução de máquina tem dois *bytes* de comprimento. Os primeiros quatro *bits* constituem o código de operação; os últimos 12 *bits* compõem o campo de operando. A tabela seguinte contém a lista das instruções, em notação hexadecimal, juntamente com uma rápida descrição de cada uma. As letras R, S e T são usadas nesses campos no lugar de dígitos hexadecimais para identificar um registrador, que varia conforme cada aplicação específica da instrução. As letras X e Y são usadas no lugar de dígitos hexadecimais nos campos variáveis que não representam um registrador.

| <b>Código de operação</b> | <b>Operando</b> | <b>Descrição</b>                                                                                                                                                                                                                                                                                                   |
|---------------------------|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1                         | RXY             | LOAD ( <i>carrega</i> ) o registrador R com o padrão de <i>bits</i> encontrado na posição de memória de endereço XY.<br><i>Exemplo:</i> 14A3 carrega o conteúdo da posição de memória de endereço A3 no registrador 4.                                                                                             |
| 2                         | RXY             | LOAD ( <i>carrega</i> ) o registrador R com o padrão de <i>bits</i> XY.<br><i>Exemplo:</i> 20A3 coloca o valor A3 no registrador 0.                                                                                                                                                                                |
| 3                         | RXY             | STORE ( <i>armazena</i> ) o padrão de <i>bits</i> encontrado no registrador R na posição de memória de endereço XY.<br><i>Exemplo:</i> 35B1 armazena o conteúdo do registrador 5 na posição de memória de endereço B1.                                                                                             |
| 4                         | 0RS             | MOVE ( <i>copia</i> ) o padrão de <i>bits</i> encontrado no registrador R para o registrador S.<br><i>Exemplo:</i> 40A4 copia o conteúdo do registrador A no registrador 4.                                                                                                                                        |
| 5                         | RST             | ADD ( <i>soma</i> ) os padrões de <i>bits</i> dos registradores S e T, em complemento de dois, e coloca o resultado no registrador R.<br><i>Exemplo:</i> 5726 soma os valores binários dos registradores 2 e 6 e coloca esse resultado no registrador 7.                                                           |
| 6                         | RST             | ADD ( <i>soma</i> ) os padrões de <i>bits</i> dos registradores S e T em notação de vírgula flutuante e coloca o resultado em vírgula flutuante no registrador R.<br><i>Exemplo:</i> 634E soma os valores dos registradores 4 e E na notação em vírgula flutuante e coloca o resultado calculado no registrador 3. |
| 7                         | RST             | OR ( <i>ou</i> ) executa a operação lógica OR sobre os padrões de <i>bits</i> dos registradores S e T e coloca o resultado no registrador R.<br><i>Exemplo:</i> 7CB4 coloca no registrador C o resultado da operação OR com os conteúdos dos registradores B e 4.                                                  |
| 8                         | RST             | AND ( <i>e</i> ) executa a operação lógica AND sobre os padrões de <i>bits</i> dos registradores S e T e coloca o resultado no registrador R.<br><i>Exemplo:</i> 8045 coloca no registrador 0 o resultado da operação AND entre os conteúdos dos registradores 4 e 5.                                              |
| 9                         | RST             | EXCLUSIVE OR ( <i>ou exclusivo</i> ) executa a operação de OU EXCLUSIVO sobre os padrões de <i>bits</i> dos registradores S e T e coloca o resultado no registrador R.                                                                                                                                             |

| Código de operação | Operando | Descrição                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                    |          | <i>Exemplo:</i> 95F3 coloca no registrador 5 o resultado da operação EXCLUSIVE OR entre os conteúdos dos registradores F e 3.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| A                  | ROX      | ROTATE ( <i>gira</i> ) o padrão de <i>bits</i> do registrador R, de X <i>bits</i> para a direita. Sempre coloca o <i>bit</i> que está na extremidade de mais baixa ordem na de mais alta ordem.<br><i>Exemplo:</i> A403 gira em 3 bits para a direita o conteúdo do registrador 4, de forma circular.                                                                                                                                                                                                                                                                                                                                                                                                            |
| B                  | RXY      | JUMP ( <i>salta</i> ) para a instrução localizada na posição de memória de endereço XY se o padrão de <i>bits</i> do registrador R coincidir com o padrão de <i>bits</i> do registrador 0. Caso contrário, prossegue na seqüência normal de execução. (O salto é implementado copiando o valor de XY no contador de instruções durante a fase de execução.)<br><i>Exemplo:</i> B43C primeiro compara o conteúdo do registrador 4 com o do registrador 0. Se os dois forem iguais, o padrão 3C será colocado no contador de instruções, de modo que a próxima instrução a ser executada será a localizada naquele endereço de memória. Caso contrário, a execução do programa continuará em sua seqüência normal. |
| C                  | 000      | HALT ( <i>pára</i> ) a execução.<br><i>Exemplo:</i> C000 pára a execução do programa.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |



# A PÊNDICE D

---

## *Exemplos de programas em linguagens de alto nível*

Este apêndice apresenta exemplos de programas nas linguagens Ada, C, C++, C#, FORTRAN, Java e Pascal. Cada programa recebe uma lista de nomes pelo teclado, a ordena utilizando o algoritmo de ordenação por inserção e a apresenta ordenada na tela do monitor.

### **Ada**

A linguagem Ada tem este nome em homenagem a Augusta Ada Byron (1815-1851), que era assistente de Charles Babbage e filha do poeta Lord Byron, e foi desenvolvida por iniciativa do Departamento de Defesa norte-americano em uma tentativa de obter uma única linguagem de propósito geral para todas as suas necessidades em desenvolvimento de *software*. Uma ênfase importante durante o projeto da linguagem Ada foi incorporar recursos para a programação de sistemas computacionais em tempo real, os quais são utilizados como parte de máquinas maiores, como sistemas de direcionamento de mísseis, sistemas de controle ambiental de edifícios e sistemas de controle de automóveis e de pequenas aplicações domésticas. Assim, Ada contém recursos para expressar ações em ambientes de processamento paralelo, bem como técnicas adequadas para manipular casos especiais (chamados exceções) que possam surgir no ambiente da aplicação. A Ada é uma linguagem imperativa com muitas características orientadas a objeto, representando, portanto, um passo na evolução da programação imperativa para as modernas linguagens orientadas a objeto.

A Figura D.1 apresenta um exemplo de programa escrito na linguagem Ada.

### **C**

A linguagem C foi desenvolvida por Dennis Ritchie no Bell Laboratories no início dos anos 1970. Embora originalmente projetada como uma linguagem para desenvolver sistemas operacionais e compiladores, C ganhou popularidade na comunidade de programação e tem desfrutado das vantagens da padronização por meio dos esforços do American National Standards Institute.

O C foi inicialmente idealizado para ser um simples passo acima das linguagens de máquina. Por conseguinte, sua sintaxe se mostra concisa, quando comparada com as demais linguagens de alto nível, as quais se utilizam de palavras inglesas completas para expressar algumas primitivas que são representadas por símbolos especiais em C. Esta concisão é uma das razões da sua popularidade, pois permite representações compactas de algoritmos complexos. (Uma representação concisa geralmente é mais legível que outra mais longa.)

A Figura D.2 apresenta um exemplo de um programa escrito na linguagem C.

```
--Programa para manipular uma lista
with TEXT_IO;
use TEXT_IO;
procedure MAIN is
 subtype TIPO_NOME is STRING (1..8);
 COMPRIMENTO_DA_LISTA: constant := 10;
 NOMES: array (1..COMPRIMENTO_DA_LISTA) of TIPO_NOME;
 PIVOT: TIPO_NOME;
 LACUNA: INTEGER;
begin
 --Primeiro, obter os nomes a partir do terminal.
 for K in 1..COMPRIMENTO_DA_LISTA loop
 GET(NOMES(K));
 end loop;
 --Ordenar a lista (LACUNA contém a posição do espaço vago na lista
 -- desde o instante que o pivô é removido até ser
 -- re inserido.)
 for N in 2.. COMPRIMENTO_DA_LISTA loop
 PIVOT := NOMES(N);
 LACUNA := N;
 for M in reverse 1..N - 1 loop
 if NOMES(M) > PIVOT
 then NOMES(M + 1) := NOMES(M);
 else exit;
 end if;
 LACUNA := M;
 end loop;
 NOMES(LACUNA) := PIVOT;
 end loop;
 --Agora, imprimir a lista ordenada.
 for K in 1..COMPRIMENTO_DA_LISTA loop
 NEW_LINE;
 PUT(NOMES(K));
 end loop;
end MAIN;
```

---

**FIGURA D.1** Um exemplo de programa escrito na linguagem Ada.

## C++

A linguagem C++ foi desenvolvida por Bjarne Stroustrup no Bell Laboratories, como uma versão ampliada da linguagem C. O objetivo era produzir uma linguagem compatível com o paradigma orientado a objeto.

A Figura D.3 apresenta uma implementação em C++ do algoritmo de ordenação por inserção. As últimas quatro instruções deste programa exigem que um objeto chamado `listadenomes` seja definido como de “tipo” `lista` e que este novo objeto execute em si mesmo as operações `obtemnomes`, `ordena` e `imprimenomes`. A parte anterior do programa define as propriedades que qualquer objeto de “tipo” `lista` irá possuir. Cada objeto contém uma matriz interna de caracteres chamada `nomes` e três operações chamadas `obtemnomes`, `ordenalista` e `imprimenomes`. Note-se que as definições de tais operações são iguais às partes do programa em linguagem C da Figura D.2. A diferença é que, no programa C++, tais operações são consideradas partes das propriedades de um objeto, enquanto em C são consideradas como unidades dentro da parte de procedimentos do programa.

```
/* Programa para manipular uma lista */

#include <stdio.h>
#include <string.h>

int main ()
{
 char nomes[10][9],pivot[9];
 int i,j;
/* obter os nomes */
 for (i = 0; i < 10; ++i)
 scanf("%s",nomes[i]);
/*ordenar a lista */
 for (i = 1; i < 10; ++i)
 {
 strcpy(pivot,nomes[i]);
 j = i - 1;
 while ((j >= 0)&&(strcmp(pivot,nomes[j]) < 0))
 {strcpy(nomes[j+1],nomes[j]); --j;};
 strcpy(nomes[j+1], pivot);
 }

/*imprimir a lista ordenada */
 for (i = 0; i < 10; ++i)
 printf("%s\n",nomes[i]);
}
```

---

**FIGURA D.2** Exemplo de um programa escrito na linguagem C.

## C#

C# é um primo próximo do C++ e do Java. Foi desenvolvido pela Microsoft para ser uma ferramenta importante em .NET framework, que é um sistema compreensível de desenvolvimento de aplicações para as máquinas que executam *software* de sistema da Microsoft. Como mostra o exemplo da Figura D.4, um programa em C# é muito parecido com um programa em C++ ou em Java. De fato, a razão pela qual a Microsoft introduziu o C# como linguagem diferente não é que ele seja verdadeiramente novo, mas, como linguagem diferente, a Microsoft pode personalizar características específicas da linguagem sem se preocupar com os padrões já associados com outras linguagens, nem com os direitos autorais de propriedade de outras corporações. Com o respaldo da Microsoft, C# e .NET framework prometem influenciar o mundo do desenvolvimento de *software* nos anos vindouros.

## FORTRAN

FORTRAN é uma composição das palavras *FORmula TRANslator*. Foi uma das primeiras linguagens de alto nível a ser desenvolvida (anunciada em 1957) e a primeira a ganhar ampla aceitação dentro da comunidade da computação. Durante anos, a sua definição oficial sofreu numerosas extensões, incluindo o FORTRAN IV, o FORTRAN 77, o FORTRAN 90, o FORTRAN 95 e agora o FORTRAN 2000. Embora criticado por muitos, o FORTRAN continua sendo uma linguagem muito popular dentro da comunidade científica. Por exemplo, muitos pacotes de análise numérica e de estatística são e provavelmente

```
// Programa para manipular uma lista

#include <iostream.h>
#include <string.h>
const int ComprimentoDaLista = 10;

// Todos os objetos do tipo lista contêm uma lista de nomes e
// três métodos públicos chamados obtemnomes, ordenalista e
// imprimenomes.

class lista
{private:
 char nomes[ComprimentoDaLista][9];

public:

 void obtemnomes()
 {int i;
 for (i = 0; i < ComprimentoDaLista; ++i)
 cin >> nomes[i];
 }

 void ordenalista()
 {int i,j;
 char pivot[9];
 for (i = 1; i < ComprimentoDaLista; ++i)
 strcpy(pivot, nomes[i]);
 j = i - 1;
 while ((j >= 0) && (strcmp(pivot, nomes [j]) < 0))
 strcpy(nomes[j+1], nomes[j]);
 --j;
 }
 strcpy(nomes[j+1], pivot);
 }

 void imprimenomes()
 {int i;
 cout << endl;
 for (i = 0; i < ComprimentoDaLista; ++i)
 cout << nomes[i] << endl;
 }
};

// Definir um objeto chamado listadenomes e solicitar-lhe que
// colete alguns nomes, os ordene e imprima essa lista.

int main()
{lista listadenomes;
 listadenomes.obtemnomes();
 listadenomes.ordenalista();
 listadenomes.imprimenomes();
}
```

---

**FIGURA D.3** Exemplo de programa escrito na linguagem C++.

```
// Programa para manipular uma lista

// Todos os objetos do tipo lista contêm uma lista de nomes e
// três métodos públicos chamados ObtemNomes, OrdenaLista e
// ImprimeNomes.

class Lista {
 const int ComprimentoDaLista = 10;
 private String[] Nomes;

 List() {
 Nomes = new String[ComprimentoDaLista];
 }

 public void ObtemNomes() {
 for (int i = 0; i < ComprimentoDaLista; i++)
 Nomes[i] = System.Console.ReadLine();
 }

 public void OrdenaLista() {
 int j;
 String Pivot;
 for (int i = 1; i < ComprimentoDaLista; i++) {
 Pivot = Nomes[i];
 j = i - 1;
 while ((j >= 0) && String.Compare(Pivot, Nomes[j], true) < 0) {
 Nomes[j+1] = Nomes[j];
 j--;
 }
 }
 Nomes[j+1] = Pivot;
 }

 public void ImprimeNomes() {
 for (int i = 0; i < ComprimentoDaLista; i++)
 System.Console.WriteLine(Nomes[i]);
 }
}

// Definir um objeto chamado ListaDeNomes e solicitar-lhe que
// colete alguns nomes, os ordene e imprima essa lista.

class Ordena {
 public static void main() {
 Lista ListaDeNomes = new Lista();
 ListaDeNomes.ObtemNomes();
 ListaDeNomes.OredenaLista();
 ListaDeNomes.ImprimeNomes();
 Return 0;
 }
}
```

---

**FIGURA D.4** Exemplo de programa escrito na linguagem C#.

continuarão sendo escritos em FORTRAN. A Figura D.5 apresenta um exemplo de programa escrito em FORTRAN.

## Java

Java é uma linguagem orientada a objeto, desenvolvida pela Sun Microsystems no início dos anos 1990. Seus projetistas tomaram emprestadas muitas das características das linguagens C e C++. Por ser uma linguagem nova, Java não goza as vantagens da padronização. Na verdade, ela ainda se encontra no início de sua fase evolutiva.

O entusiasmo pelo Java não é tanto pela linguagem em si, mas por sua implementação universal e pela grande quantidade de gabaritos pré-fabricados disponíveis nos ambientes de programação Java. Essa implementação universal significa que um programa escrito em Java pode ser executado em várias máquinas e a disponibilidade de gabaritos, que softwares complexos podem ser desenvolvidos com relativa facilidade. Por exemplo, os gabaritos dos tipos *applet* e  *servlet* facilitam o desenvolvimento de software para a *World Wide Web*.

A Figura D.6 apresenta um exemplo de um programa escrito na linguagem Java. Observe a semelhança entre Java e C++.

## Pascal

O Pascal é uma linguagem imperativa cujo nome foi dado em homenagem ao matemático e inventor francês Blaise Pascal (1623-1662). Anunciada por Niklaus Wirth em 1971, incorpora muitos dos mais modernos recursos de projeto, como a ênfase em tipos e estrutura de dados, a sintaxe de formato livre e

```

! Programa para manipular uma lista
 INTEGER J,K
 CHARACTER(LEN=8) Pivot
 CHARACTER(LEN=8) DIMENSION(10) Nomes
! Primeiro, obter os nomes.
 READ(UNIT=5, FMT=100) (Nomes(K), K=1,10)
100 FORMAT(A8)
! Agora, ordenar a lista.
IteracaoExterna: DO J=2,10
 Pivot = Nomes(J)
IteracaoInterna: DO K=J-1, 1,-1
 IF (Nomes(K).GT. Pivot) THEN
 Nomes(K+1) = Nomes(K)
 ELSE
 EXIT IteracaoInterna
 ENDIF
 END DO IteracaoInterna
 Nomes(K+1) = Pivot
 END DO IteracaoExterna
! Agora, imprimir a lista ordenada.
 WRITE(UNIT=6,FMT=400) (Nomes(K),K=1,10)
400 FORMAT ('',A8)
 END

```

---

**FIGURA D.5** Exemplo de programa escrito na linguagem FORTRAN.

```
// Programa para manipular uma lista

import java.io.*;

// Todos os objetos do tipo lista contêm uma lista de nomes e
// três métodos públicos chamados obtemnomes, ordenalista e
// imprimenomes.

class lista {
 final int ComprimentoDaLista = 10;
 private String[] nomes;
 public lista() {
 nomes = new String[ComprimentoDaLista];
 }
 public void obtemnomes() {
 int i;
 DataInput dados = new DataInputStream(System.in);
 for (i=0; i < ComprimentoDaLista; i++)
 try {nomes[i] = dados.readLine();}
 }
 catch(IOException e) {};
 }
 public void ordenalista() {
 int i,j;
 String pivot;
 for (i=1; i < ComprimentoDaLista; i++) {
 pivot = nomes[i];
 j = i - 1;
 while ((j >= 0) && (pivot.compareTo(nomes[j]) < 0)) {
 nomes[j+1] = nomes[j];
 j--;
 }
 nomes[j+1] = pivot;
 }
 }
 public void imprimenomes() {
 int i;
 for (i=0; i < ComprimentoDaLista; i++)
 System.out.println(nomes[i]);
 }
}

// Definir um objeto chamado listadenomes e solicitar-lhe que
// colete alguns nomes, os ordene e imprima os resultados.
class ordena {
 public static void main(String args[]){
 lista listadenomes = new lista();
 listadenomes.obtemnomes();
 listadenomes.ordenalista();
 listadenomes.imprimenomes();
 }
}
```

---

**FIGURA D.6** Um exemplo de programa escrito na linguagem Java.

numerosas estruturas de controle. O Pascal foi usado intensamente no ensino da computação nos anos 1980 e início dos 1990, pois o seu projeto reforça uma abordagem organizada ao desenvolvimento de programas. Entretanto, com a atual ênfase no paradigma orientado a objeto, o Pascal tem dado lugar a linguagens como o C++ e o Java. A Figura D.7 apresenta um exemplo de programa escrito na linguagem Pascal.

```

{Programa para manipular uma lista}
program OrdenaPorInsercao(input, Output);
const Brancos = ' ';
 ComprimentoDaLista = 10;

type TipoNome = packed array [1 .. 8] of char;
var Nomes: Array[1 .. ComprimentoDaLista] of Tiponome;
 Pivot: Tiponome;
 PosicaoEncontrada: Boolean;
 J,M,N: Integer;
{ObtemNome é um procedimento para a leitura de um nome completo.}
procedure ObtemNome(var Nome: TipoNome);
var J: Integer;
begin J := 1;
repeat read(Nome[J]); J:= J + 1; until (J > 8) or eoln;
readln
end;
begin
{Antes de mais nada, obter os nomes pelo terminal.}
 for J := 1 to ComprimentoDaLista do
 begin Nomes[J] := Brancos; ObtemNome(Nomes[J]) end;
{Ordenar a lista.}
 N := 2;
 repeat
 Pivot := Nomes[N];
 M := N - 1;
 PosicaoEncontrada := false;
 while (not PosicaoEncontrada) do
 if Nomes[M] > Pivot
 then begin Nomes[M+1]:= Nomes[M];
 M := M - 1;
 if M = 0 then PosicaoEncontrada := true
 end
 else PosicaoEncontrada := true;
 Nomes[M+1] := Pivot;
 N := N + 1;
 until N > ComprimentoDaLista;
{Agora, imprimir a lista ordenada.}
 for J := 1 to ComprimentoDaLista do writeln (Nomes[J])
end.

```

---

**FIGURA D.7** Exemplo de programa escrito na linguagem Pascal.

# A PÊNDICE E

---

## *A equivalência entre estruturas iterativas e recursivas*

Utilizaremos, neste apêndice, a nossa linguagem Bare Bones do Capítulo 11 como uma ferramenta para responder à questão, colocada no Capítulo 4, sobre a capacidade relativa das estruturas iterativas e recursivas. Recordemo-nos de que Bare Bones contém somente três instruções de atribuição (`clear`, `incr` e `decr`) e uma estrutura de controle (construída com o par de instruções `while-end`). Além disto, esta linguagem simples tem a mesma capacidade computacional de uma máquina de Turing; portanto, se aceitarmos a tese de Church-Turing, concluiremos que qualquer problema com solução algorítmica tem uma solução que pode ser expressa em Bare Bones.

O primeiro passo na comparação entre estruturas iterativas e recursivas é substituir a estrutura iterativa de Bare Bones por uma recursiva. Isso é feito retirando as instruções `while` e `end` da linguagem, e, em troca, dando à linguagem a capacidade de dividir um programa escrito em Bare Bones em unidades e também de chamar tais unidades de alguma parte do programa. Mais precisamente, propomos que cada programa na linguagem modificada seja composto de várias unidades de programa, sintaticamente independentes. Cada programa deve conter exatamente uma unidade chamada `MAIN` com a estrutura sintática

`MAIN: begin;`

`.`

`.`

`.`

`end;`

(em que os pontos representam outras instruções da Bare Bones) e talvez outras unidades (semanticamente subordinadas a `MAIN`) que apresentam a seguinte estrutura:

*unidade*: `begin;`

`.`

`.`

`.`

`return;`

(em que *unidade* representa o nome da unidade, apresentando a mesma sintaxe dos nomes de variável). A semântica dessa estrutura particionada é que o programa sempre inicia sua execução a partir do início da unidade `MAIN` e pára ao atingir a sua instrução final. As unidades de programa, exceto a `MAIN` podem ser chamadas como procedimentos, por meio da instrução condicional

`if nome not 0 perform unidade;`

(onde *nome* representa qualquer nome de variável e *unidade*, qualquer nome de unidade do programa, diferente de MAIN). Além disso, permitimos que as unidades diferentes de MAIN chamem a si próprias, recursivamente.

Com esses recursos adicionais, podemos simular a estrutura while-end, encontrada na Bare Bones original. Por exemplo, um programa escrito em Bare Bones da forma

```
while X not 0 do;
 S;
end;
```

(em que S representa qualquer seqüência de instruções Bare Bones) pode ser substituído pela estrutura de unidades

```
MAIN: begin;
 if X not 0 perform unidadeA;
 end;
unidadeA: begin;
 S;
 if X not 0 perform unidadeA;
 return;
```

Como consequência, podemos concluir que a linguagem modificada preserva todas as capacidades da linguagem Bare Bones original.

Também é possível demonstrar que qualquer problema que admita solução com o uso da linguagem modificada também pode ser resolvido usando Bare Bones. Um método para provar isso é mostrar como qualquer algoritmo expresso na linguagem modificada pode ser escrito em Bare Bones original. Isso, porém, envolve uma descrição explícita de como as estruturas recursivas podem ser simuladas com a estrutura while-end de Bare Bones.

Para o nosso propósito, é mais simples confiar na tese de Church-Turing, apresentada no Capítulo 11. Essa tese e o fato de Bare Bones apresentar a mesma capacidade de uma máquina de Turing confirmam que nenhuma linguagem pode ser mais capaz do que a nossa Bare Bones original. Assim, concluímos, de imediato, que qualquer problema solúvel com a nossa linguagem modificada também pode ser resolvido utilizando Bare Bones.

Concluímos que o poder da linguagem modificada é igual ao da Bare Bones original. A única diferença entre ambas é que uma apresenta estrutura de controle iterativa e a outra fornece recursão. Devemos concluir, então, que as duas estruturas de controle são de fato equivalentes em termos de poder computacional.

# A PÊNDICE F

---

## *Respostas das questões e dos exercícios*

### **Parte Um**

#### **Capítulo 1**

##### ***Seção 1.1***

1. Uma e somente uma das duas entradas superiores deve ser 1, e a entrada inferior, 1.
2. O 1 na entrada inferior é “negado” para 0 pela porta lógica NOT, forçando a saída da porta lógica AND a ser 0. Assim, as duas entradas da porta lógica OR ficam 0 (convém lembrar que a entrada superior do flip-flop é mantida em 0), de modo que a saída da porta lógica OR se torna 0. Isso significa que a saída da porta lógica AND permanecerá 0 depois que a entrada inferior do flip-flop retornar a 0.
3. A saída da porta lógica OR superior se tornará 1 e fará a porta lógica NOT superior produzir uma saída 0. Isso fará a porta lógica OR inferior produzir um 0, o que levará a porta lógica NOT inferior a produzir um 1, o qual será visto como a saída do flip-flop e também como a realimentação da porta lógica OR superior, onde mantém a saída desta porta em 1, mesmo após a entrada do flip-flop ter retornado para 0.
4. O flip-flop está isolado dos valores de entrada do circuito quando o pulso do relógio for 0 e responde aos valores de entrada do circuito quando o pulso do relógio for 1.
5.
  - a. O circuito completo é equivalente a uma única porta lógica XOR.
  - b. O circuito completo também é equivalente a uma única porta lógica XOR.
6.
  - a. 6AF2
  - b. E85517
  - c. 48
7.
  - a. 0101111110110010111
  - b. 0110000100001010
  - c. 1010101111001101
  - d. 0000000100000000

##### ***Seção 1.2***

1. No primeiro caso, a célula 6 de memória termina contendo o valor 5. No segundo caso, termina com o valor 8.

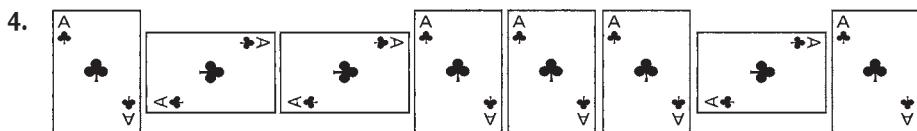
2. O passo 1 apaga o valor original da célula 3 de memória quando o novo valor é escrito nela. Por conseguinte, o passo 2 não coloca o valor original da célula 3 de memória na célula de memória 2. O resultado é que ambas terminam com o valor que estava originalmente na célula 2 de memória. Um procedimento correto seria o seguinte:  
*Passo 1.* Mover o conteúdo da célula de memória 2 para a célula 1.  
*Passo 2.* Mover o conteúdo da célula de memória 3 para a célula 2.  
*Passo 3.* Mover o conteúdo da célula de memória 1 para a célula 3.
3. 32768 bits.

### **Seção 1.3**

1. Recuperação mais rápida de dados e taxas de transferência mais altas.
2. O ponto a ser lembrado aqui é que a lentidão dos movimentos mecânicos, comparada com a velocidade do funcionamento interno dos computadores, pede que minimizemos o número de vezes que devemos mover cabeçotes de leitura e gravação. Se preenchermos completamente uma superfície antes de começar a seguir, moveremos o cabeçote todas as vezes que alcançarmos o final de uma trilha. Portanto, o número de movimentos é aproximadamente igual ao número total de trilhas existentes nas duas superfícies. Se alternarmos entre as superfícies, por meio de chaveamento eletrônico entre os cabeçotes nas duas superfícies, moveremos mecanicamente o cabeçote somente após cada cilindro ter sido gravado.
3. Nesta aplicação, ocorre uma constante expansão e retração dos dados. Se a informação for armazenada em fita, isso resultará em um processo infundável de reescrita para acomodar o turbilhão gerado nos dados. (O último bloco de dados fica balançando de um lado para o outro à medida que as primeiras reservas da fita vão sendo feitas, desfeitas ou ficando desatualizadas.) Se for utilizado armazenamento em disco, cada mudança afetará somente a parte dos dados armazenada no setor em questão. Por conseguinte, torna-se necessário reescrever bem menos dados durante as atualizações.
4. O espaço de armazenamento é alocado em unidades de setores físicos. Se o último não estiver cheio, o texto adicional poderá ser acrescentado sem aumentar o espaço de armazenamento alocado ao arquivo. Se o último setor estiver cheio, qualquer acréscimo ao documento exigirá a alocação de setores adicionais.

### **Seção 1.4**

1. *Computer science.*
2. Os dois padrões são iguais, exceto pelo fato de o sexto bit da extremidade de mais baixa ordem ser sempre 0 para maiúsculas e 1 para minúsculas.
3. a. 01010111      01101000      01100101      01110010  
   01100101      00100000      01100001      01110010  
   01100101      00100000      01111001      01101111  
   01110101      00111111  
 b. 00100010      01001000      01101111      01110111  
   00111111      00100010      00100000      01000011  
   01101000      01100101      01110010      01111001  
   01101100      00100000      01100001      01110011  
   01101011      01100101      01100100      00101110  
 c. 00110010      00101011      00110011      00111101  
   00110101      00101110



5. a. 5      b. 9      c. 11  
d. 6      e. 16      f. 18
6. a. 110      b. 1101      c. 1011  
d. 10010      e. 11011      f. 100
7. Em 24 bits, podemos armazenar três símbolos utilizando ASCII. Assim, podemos armazenar valores até 999. Entretanto, se utilizarmos os bits como dígitos binários, armazenaremos valores até 16.777.215.
8. a. 15.15      b. 51.0.128      c. 10.160
9. Como enfatizado no texto, as técnicas vetoriais são mais úteis nas mudanças de escala do que a codificação de imagens por mapas de bits. Para desenhos simples, elas também podem ser mais compactas. Por outro lado, não têm a mesma qualidade fotográfica dos mapas de bits.
10. Com uma taxa de amostragem de 44.100 amostras por segundo, uma hora de música estéreo necessita 635.040.000 bytes de armazenamento. Assim, ela enche um CD cuja capacidade é ligeiramente superior a 600 MB.

### **Seção 1.5**

1. a. 42      b. 33      c. 23  
d. 6      e. 31
2. a. 100000      b. 1000000      c. 1100000  
d. 1111      e. 11011
3. a.  $3\frac{1}{4}$       b.  $5\frac{7}{8}$       c.  $2\frac{1}{2}$   
d.  $6\frac{3}{8}$       e.  $\frac{5}{8}$
4. a. 100,1      b. 10,11      c. 1,001  
d. 0,0101      e. 101,101
5. a. 100111      b. 1011,110      c. 100000  
d. 1000,00

### **Seção 1.6**

1. a. 3      b. 15      c. -4  
d. -6      e. 0      f. -16
2. a. 00000110      b. 11111010      c. 11101111  
d. 00001101      e. 11111111      f. 00000000
3. a. 11111111      b. 10101011      c. 00000100  
d. 00000010      e. 00000000      f. 10000001
4. a. Com 4 bits, o maior valor é 7 e o menor, -8.  
b. Com 6 bits, o maior valor é 31 e o menor, -32.  
c. Com 8 bits, o maior valor é 127 e o menor, -128.

- 5.** a. 0111 ( $5 + 2 = 7$ )   b. 0100 ( $3 + 1 = 4$ )   c. 1111 ( $5 + (-6) = -1$ )  
d. 0001 ( $-2 + 3 = 1$ )   e. 1000 ( $-6 + (-2) = -8$ )
- 6.** a. 0111      b. 1011 (estouro)      c. 0100 (estouro)  
d. 0001      e. 1000 (estouro)
- 7.** a. 0110      b. 0011      c. 0100      d. 0010      e. 0001  

$$\begin{array}{r} + 0001 \\ \hline 0111 \end{array}$$

$$\begin{array}{r} + 1110 \\ \hline 0001 \end{array}$$

$$\begin{array}{r} + 1010 \\ \hline 1110 \end{array}$$

$$\begin{array}{r} + 0100 \\ \hline 0110 \end{array}$$

$$\begin{array}{r} + 1011 \\ \hline 1100 \end{array}$$
- 8.** Não. O estouro ocorre quando é feita uma tentativa de armazenar um número muito grande para o sistema em questão. Ao somar um valor positivo com um negativo, o valor do resultado deve estar compreendido entre esses dois valores. Assim, como os valores originais são pequenos o suficiente para ser representados, o resultado também será.
- 9.** a. 6 dado que  $1110 \rightarrow 14 - 8$   
b.  $-1$  dado que  $0111 \rightarrow 7 - 8$   
c. 0 dado que  $1000 \rightarrow 8 - 8$   
d.  $-6$  dado que  $0010 \rightarrow 2 - 8$   
e.  $-8$  dado que  $0000 \rightarrow 0 - 8$   
f. 1 dado que  $1001 \rightarrow 9 - 8$
- 10.** a. 1101 dado que  $5 + 8 = 13 \rightarrow 1101$   
b. 0011 dado que  $25 + 8 = 3 \rightarrow 0011$   
c. 1011 dado que  $3 + 8 = 11 \rightarrow 1011$   
d. 1000 dado que  $0 + 8 = 8 \rightarrow 1000$   
e. 1111 dado que  $7 + 8 = 15 \rightarrow 1111$   
f. 0000 dado que  $28 + 8 = 0 \rightarrow 0000$
- 11.** Não. O maior valor que pode ser armazenado na notação de excesso de oito é 7, representado por 1111. Para representar um valor maior, deve ser utilizado, no mínimo, excesso de 16 (que usa padrões de 5 bits). Do mesmo modo, 6 não pode ser representado na notação de excesso de quatro. (O maior valor que pode sê-lo é 3.)

### Seção 1.7

- 1.** a.  $\frac{5}{8}$       b.  $3\frac{1}{4}$       c.  $\frac{9}{32}$   
d.  $-1\frac{1}{2}$       e.  $-\frac{11}{64}$
- 2.** a. 01101011      b. 01111010 (erro de truncamento)  
c. 01001100      d. 11101110  
e. 11111000 (erro de truncamento)
- 3.** 01001001 ( $\frac{9}{16}$ ) é maior que 00111101 ( $\frac{13}{32}$ ). Segue abaixo um método simples de determinar qual dos dois padrões representa o maior valor:  
*Caso 1.* Se os bits de sinal forem diferentes, o maior é aquele com bit de sinal igual a 0.  
*Caso 2.* Se os dois bits de sinal forem 0, varrer o restante dos padrões, da esquerda para a direita, até que seja encontrada uma posição de bit em que os dois padrões diferem. O padrão que contiver um 1 nesta posição representa o maior valor.  
*Caso 3.* Se os dois bits de sinal forem 1, varrer o restante dos padrões, da esquerda para a direita, até que seja encontrada uma posição de bit em que os dois padrões diferem. O padrão que contiver um 0 nesta posição representa o maior valor.

A simplicidade deste processo de comparação é uma das razões da utilização da notação de excesso para representar o expoente em sistemas de vírgula flutuante, no lugar da notação de complemento de dois.

4. O maior valor seria  $7^{1/2}$ , que é representado pelo padrão 01111111. Como é para o menor valor positivo, você poderia argumentar que há duas respostas “corretas”. Primeiro, se você utilizar o processo de codificação descrito no texto, que exige que o bit mais significativo da mantissa seja 1 (chamada forma normalizada), a resposta será  $1/32$ , representada pelo padrão 00001000. A maioria das máquinas não impõe esta restrição para valores próximos de 0. Para tal máquina, a resposta correta seria  $1/256$ , que seria representado pelo padrão 00000001.

### Seção 1.8

1. Em notação hexadecimal, a mensagem é B5E95EFA56.
2. As respostas podem variar. Uma possibilidade é bbabbbaaba (5, 5, b) (10, 7, a).
3. Os desenhos coloridos consistem em blocos de uma cor com bordas finas. Além disso, o número de cores envolvidas é limitado.
4. Teríamos 1.049.576 pixels, e cada um precisaria de 1 byte para representar a sua cor. As técnicas de compressão adicional envolvidas no GIF reduziriam esse valor de modo significativo, embora a sua eficiência dependa da complexidade da imagem. Em geral, as imagens simples não necessitam mais do que alguns KB de armazenamento quando codificadas usando GIF. Se o padrão JPEG básico fosse usado, cada bloco de 2 por 2 pixels exigiria apenas seis componentes. Assumindo um byte por componente, a imagem de 1024 por 1024 pixels necessaria no máximo 1,5 MB, porém mais uma vez as técnicas de compressão adicional normalmente reduziriam esse gasto para menos de 100 KB.
5. O padrão JPEG básico se beneficia do fato de o olho humano não ser tão sensível a mudanças de cor quanto a mudanças de brilho. Assim, ele reduz o mínimo de bits usados para representar a informação de cor sem perda notável na qualidade da imagem.
6. Quando se codifica a informação, são feitas aproximações. No caso de dados numéricos, essas aproximações são combinadas quando as computações são realizadas, o que pode levar a resultados equivocados. As aproximações não são tão críticas nos casos de imagem e som porque os dados codificados normalmente são apenas armazenados, transferidos e reproduzidos. Se, porém, a imagem ou som fossem repetidamente reproduzidos e então recodificados, essas aproximações poderiam se combinar e, em última instância, levar a dados inúteis.

### Seção 1.9

1. b, c, e.
2. Sim. Se ocorrer um número par de erros em um byte, a técnica de paridade não os descobrirá.
3. Neste caso, erros ocorrem nos bytes a e d da pergunta número 1. A resposta à pergunta número 2 é a mesma.
4. a.

|           |           |           |
|-----------|-----------|-----------|
| 001010111 | 001101000 | 101100101 |
| 101110010 | 101100101 | 100100000 |
| 001100001 | 101110010 | 101100101 |
| 000100000 | 001111001 | 101101111 |
| 001110101 | 100111111 |           |

  
 b.
 

|           |           |           |
|-----------|-----------|-----------|
| 100100010 | 101001000 | 101101111 |
| 101110111 | 100111111 | 100100010 |
| 000100000 | 001000011 | 001101000 |
| 101100101 | 101110010 | 001111001 |
| 101101100 | 000100000 | 001100001 |

|    |           |           |           |
|----|-----------|-----------|-----------|
|    | 001110011 | 001101011 | 101100101 |
|    | 001100100 | 100101110 |           |
| c. | 000110010 | 100101011 | 100110011 |
|    | 000111101 | 100110101 | 100101110 |

5. a. BED                  b. CAB                  c. HEAD

6. Uma solução seria a seguinte:

|   |       |
|---|-------|
| A | 00000 |
| B | 11100 |
| C | 01111 |
| D | 10011 |

## Capítulo 2

### Seção 2.1

- Em algumas máquinas, muitas vezes isso é um processo de dois passos, que consiste, primeiramente, em copiar o conteúdo da primeira célula para um registrador e depois, escrevê-lo na célula-destino. Na maioria das máquinas, isso é feito como uma atividade, sem usar registrador intermediário.
- O valor a ser escrito, o endereço da célula de memória em que deve ser escrito e o comando para escrever.
- Os registradores de propósito geral são usados para manter os dados imediatamente aplicáveis à operação corrente; a memória principal é usada para manter os que serão necessários em futuro próximo; e o armazenamento em massa é usado para manter os que dificilmente serão necessários em um futuro próximo.

### Seção 2.2

- O termo *move* geralmente carrega a conotação de remover algo de uma posição e colocá-lo em outra, deixando, portanto, um vazio no local de origem. Na maioria dos casos, dentro de uma máquina, tal remoção não ocorre. Ao contrário, o objeto a ser movido geralmente é copiado (ou clonado) na nova célula.
- Uma técnica comum, chamada endereçamento relativo, é definir com que distância o salto deve ser realizado, em vez de indicar o endereço exato para onde deve se desviar. Por exemplo, uma instrução poderá saltar três instruções adiante ou saltar duas para trás. Deve-se, porém, notar que tais instruções exigem alteração sempre que houver alteração no número de instruções compreendidas entre a origem e o destino do salto.
- Isto poderia ser justificado das duas formas. A instrução é definida na forma de um salto condicional. Devido ao fato de que a condição de 0 ser igual a 0 é sempre satisfeita, o salto sempre ocorrerá, como se não houvesse condição. Você encontrará, com freqüência, máquinas com tais instruções em seu repertório, pois isso proporciona um projeto eficiente. Por exemplo, se uma máquina for projetada para executar uma instrução com uma estrutura da forma “If...jump to...”, tal instrução será utilizada tanto para expressar saltos condicionais como incondicionais.
- 156C = 0001010101101100  
166D = 0001011001101101  
5056 = 0101000001010110

306E = 0011000001101110  
 C000 = 1100000000000000

5. a. STORE (*armazene*) o conteúdo do registrador 6 na célula 8A de memória.  
 b. JUMP (*desvie*) para a célula DE se o conteúdo do registrador A for igual ao do registrador 0.  
 c. AND (*e lógico*) do conteúdo dos registradores 3 e C, deixando o resultado no registrador 0.  
 d. MOVE (*copie*) o conteúdo do registrador F para o 4.
6. A instrução 15AB exige que o processador consulte os circuitos de memória para obter o conteúdo da célula de memória de endereço AB. Esse valor, depois de ser obtido da memória, é guardado no registrador 5. A instrução 25AB não exige tal consulta. Pelo contrário, o valor AB é depositado no registrador 5.
7. a. 2356        b. A503        c. 80A5

### **Seção 2.3**

1. 34 hexadecimal em
2. a. 0F        b. C3
3. a. 00        b. 01        c. quatro vezes
4. Ele pára. Este é um exemplo do que geralmente é chamado código automodificável. Nele, o programa se automodifica. Note-se que as duas primeiras instruções colocam o código hexadecimal C0 na célula de memória F8, e as duas instruções seguintes colocam o padrão 00 na célula F9. Assim, antes que a máquina alcance a instrução da célula F8, lá terá sido depositada uma instrução de parada (C000).

### **Seção 2.4**

1. a. 00001011        b. 10000000        c. 00101101  
 d. 11101011        e. 11101111        f. 11111111  
 g. 11100000        h. 01101111        i. 11010010
2. 0011100 com a operação AND
3. 0011100 com a operação XOR
4. a. O valor final será 0 se a cadeia tiver um número par de 1s. Caso contrário, será 1.  
 b. O resultado é o valor do bit de paridade, para paridade par.
5. A operação lógica XOR é idêntica à adição, exceto quando os dois operandos forem 1, caso em que a operação XOR produzirá um 0, enquanto a soma será 10. (Assim, a operação XOR pode ser considerada uma operação de adição sem transporte.)
6. Usar AND com a máscara 1101111 para mudar de letra minúscula para maiúscula. Usar OR com a máscara 00100000 para mudar maiúsculas para minúsculas.
7. a. 01001101        b. 11100001        c. 11101111
8. a. 57        b. B8        c. 6F d. 6A
9. 5
10. 00110110 em complemento de dois, 01011110 em vírgula flutuante. O fato aqui é que o procedimento utilizado para somar os valores é diferente, dependendo da interpretação dada aos padrões de bits.

**11.** Uma solução seria:

- 12A7 (LOAD registrador 2 com o conteúdo da célula de memória A7.)  
 2380 (LOAD registrador 3 com o valor 80.)  
 7023 (OR registradores 2 e 3 colocam o resultado no registrador 0.)  
 30A7 (STORE o conteúdo do registrador 0 na célula de memória A7.)  
 C000 (HALT.)

**12.** Uma solução seria:

- 15E0 (LOAD registrador 5 com o conteúdo da célula de memória E0.)  
 A502 (ROTATE o conteúdo do registrador 5 em 2 bits, para a direita.)  
 260F (LOAD o registrador 6 com o valor 0F.)  
 8056 (AND registradores 5 e 6, deixando o resultado no registrador 0.)  
 30E1 (STORE o conteúdo do registrador 0 na célula de memória E1.)  
 C000 (HALT.)

**Seção 2.5**

- 1.**
  - a. 37B5
  - b. Um milhão de vezes.
  - c. Não. Uma página comum de texto contém menos de 4000 caracteres. Assim, a capacidade de imprimir cinco páginas por minuto indica uma taxa de impressão não maior que 20.000 caracteres por minuto, o que é muito menos de um milhão de caracteres por segundo. (O fato é que um computador pode enviar caracteres a uma impressora muito mais rápido do que a impressora poderia imprimi-los, assim ela precisa de algum meio para solicitar ao computador que a aguarde.)
- 2.** O disco fará 50 revoluções em um segundo, o que significa que 800 setores passarão embaixo do cabeçote de leitura/gravação em um segundo. Uma vez que cada setor contém 1024 bytes, os bits passarão embaixo do cabeçote a uma velocidade de aproximadamente 6,5 Mbps. Assim, a comunicação entre o controlador e o acionador do disco tem de ser nessa rapidez, para que o controlador acompanhe os dados que estão sendo lidos do disco.
- 3.** Um romance de 300 páginas representado em ASCII consiste em aproximadamente 1 MB, ou 8.000.000 bits. Assim, seriam necessários aproximadamente 139 segundos (ou  $2\frac{1}{3}$  minutos) para transferir o romance inteiro em 57.600 bps.

**Seção 2.6**

- 1.** O cano conteria as instruções B1B0 (em execução), 5002 (em decodificação) e B0AA (em busca). Se o valor contido no registrador 0 for igual ao contido no registrador 1, será executado o salto para a célula B0, e o esforço já dispensado às instruções no cano será desperdiçado. Entretanto, não se perderá tempo, pois o esforço dispensado a essas instruções não demanda tempo adicional.
- 2.** Se nenhuma precaução for tomada, a informação contida nas posições de memória F8 e F9 será lida como uma instrução, antes que a parte anterior do programa tenha tido a oportunidade de modificar seu conteúdo.
- 3.**
  - a. O processador que está tentando somar 1 à posição de memória pode primeiro ler o valor contido nessa posição. A seguir, o outro processador lê o valor da posição. (Note-se que, neste momento, os dois processadores terão recuperado o mesmo valor.) Se o primeiro terminar a sua adição e escrever o seu resultado nesta posição antes que o segundo termine

de escrever o resultado da sua subtração, o valor que será finalmente encontrado na posição de memória estará refletindo somente a ação do segundo processador.

- b. Os processadores leriam o dado desta posição da mesma forma como foi feito anteriormente, mas, desta vez, o segundo processador escreveriam o seu resultado antes do primeiro. Assim, somente a ação do primeiro processador teria efeito no valor final desta posição de memória.

## Parte II

### Capítulo 3

#### Seção 3.1

1. Um exemplo tradicional é a fila de pessoas que esperam para comprar ingressos para um evento. Neste caso, sempre aparece alguém que tenta “furar” a fila, violando assim a estrutura FIFO.
2. Opções (b) e (c).
3. Processamento em tempo real refere-se a sincronizar a execução de um programa com eventos originados no ambiente em que a máquina opera e processamento interativo, à interação de uma pessoa com um programa, durante sua execução. São necessárias boas características de tempo real para obter sucesso em um processamento interativo.
4. Tempo partilhado é a técnica que propicia a realização da multitarefa em uma máquina com um único processador.

#### Seção 3.2

1. Casca: Comunica com o ambiente da máquina.

*Gerente de Arquivos:* Coordena o uso das áreas de armazenamento em massa da máquina.

*Dirigentes de Dispositivo:* Manipulam a comunicação com os dispositivos periféricos da máquina.

*Gerente de Memória:* Coordena o uso da memória principal da máquina.

*Escalador:* Coordena os processos do sistema.

*Despachante:* Controla a atribuição de tempo de processador aos processos.

2. A linha divisória é vaga, e a diferença freqüentemente está na perspectiva do observador. Falando de modo geral, o software utilitário executa tarefas gerais básicas, enquanto o software de aplicação executa tarefas específicas, para uma aplicação particular da máquina.
3. Memória virtual é o espaço de memória imaginária cuja presença aparente é criada pelo processo de transferência de dados e programas de um lado para outro, entre a memória principal e o armazenamento em massa.
4. Quando a máquina é ligada, o processador começa a executar o *bootstrap* que reside em ROM. Este *bootstrap* faz com que o processador transfira o sistema operacional do disco para a área volátil da memória principal. Quando a transferência está completa, o *bootstrap* faz com que o processador transfira o controle para o sistema operacional.

#### Seção 3.3

1. Um programa é um conjunto de ordens ou instruções. Um processo é a ação de seguir tais ordens.

2. O processador completa o seu ciclo corrente, armazena o estado do processo atual e ajusta o seu contador de instruções com um valor predeterminado (que é a localização do programa de tratamento de interrupções). Assim, a instrução a ser executada em seguida será a primeira instrução do programa de tratamento de interrupções.
3. Podem ser-lhes atribuídas prioridades maiores, de forma que recebam preferência da parte do despachante. Outra opção seria dar ao processo de maior prioridade fatias de tempo mais longas.
4. Se cada processo consumir a sua fatia de tempo completa, a máquina proverá um quantum de tempo para no máximo 20 processos em um segundo. Se os processos não consumirem suas fatias completas, esse valor pode ser bem maior, mas então o tempo necessário para realizar o chaveamento de contexto talvez se torne mais significativo. (Veja o Problema 5.)
5. Na verdade, um total de  $5000/5001$  do tempo de máquina seria gasto executando processos. Contudo, quando um processo solicita uma atividade de entrada/saída, a sua fatia de tempo é terminada, enquanto o controlador executa a solicitação. Assim, se cada processo fizesse tal solicitação depois de apenas 1 microsegundo do início de seu quantum, a eficiência da máquina cairia para  $\frac{1}{2}$ . Assim, a máquina gastaria a mesma parcela de tempo fazendo chaveamentos de contexto e executando processos.
6. Que tal um negócio de vendas por correspondência e seus clientes, um corretor de títulos e seus clientes, ou um farmacêutico e seus clientes?

### **Seção 3.4**

1. Este sistema garante que o recurso não seja utilizado por mais de um processo de cada vez, porém determina que o recurso seja alocado apenas de forma alternada entre os processos. Uma vez que um processo tiver utilizado e devolvido o recurso, deverá esperar que o outro termine de utilizá-lo, antes que o original possa acessá-lo novamente. Isto é verdadeiro mesmo se o primeiro processo necessitar do recurso imediatamente, ainda que o outro não precise dele por algum tempo.
2. Se dois carros entrarem pelas extremidades opostas do túnel ao mesmo tempo, um não estará ciente da presença do outro. O processo de entrar e acender as luzes é um outro exemplo de uma região crítica. Neste caso, poderíamos chamá-lo de processo crítico. Nesta terminologia, resumiríamos a falha dizendo que os carros em extremidades opostas do túnel executariam um processo crítico ao mesmo tempo.
3. a. Isto garante que o recurso não-compartilhável não seja solicitado nem alocado parcialmente, ou seja, cede-se ao carro toda a ponte ou nada.  
b. Isto significa que o recurso não-compartilhável pode ser recuperado à força.  
c. Isto transforma o recurso não-compartilhável em compartilhável, o que elimina a competição.
4. Uma seqüência de setas que formam um caminho fechado no grafo orientado. Foi com base nesta observação que se desenvolveram técnicas para que alguns sistemas operacionais conseguissem reconhecer a existência de enlace mortal e, por conseguinte, ativar a ação corretiva apropriada.

### **Seção 3.5**

1. Uma rede aberta é aquela cujas especificações e protocolos são públicos, permitindo que diferentes vendedores possam produzir produtos compatíveis.

2. Um roteador é uma máquina que conecta duas redes para formar uma *internet*. O termo *gateway* é usado para referir-se a um roteador que conecta um domínio ao resto da *internet*.
3. O endereço completo de um hospedeiro da Internet consiste no identificador da rede e no endereço do hospedeiro.
4. Um URL é essencialmente o endereço de um documento na *World Wide Web*. Um navegador é um programa que ajuda um usuário a acessar hipertextos.
5. Qualquer rompimento no anel interromperia a comunicação. Se as mensagens pudessem ser transferidas em qualquer direção, um rompimento no anel não interromperia a comunicação.

### ***Seção 3.6***

1. A camada de ligação recebe a mensagem e a manda para a camada de rede. A camada de rede percebe que a mensagem é para um outro hospedeiro, coloca um outro endereço destinatário intermediário para a mensagem e a devolve para a camada de ligação.
2. Diferentemente do TCP, UDP é um protocolo sem conexão, que não confirma se a mensagem foi ou não recebida pelo destinatário.
3. A cada mensagem é associado um contador que determina o número máximo de vezes que a mensagem será retransmitida.
4. De fato, nada. Um programador em qualquer hospedeiro poderia modificar o *software* nele existente, com a finalidade de manter tais registros. É por isso que dados importantes devem ser criptografados.

### ***Seção 3.7***

1. O uso de senhas protege os dados (e, portanto, também a informação). O uso de criptografia protege a informação.
2. No contexto de nosso estudo, o ECPA define os direitos de privacidade envolvidos na comunicação eletrônica.
3. Se a aplicação da lei for tecnicamente inviável, essa lei não será (não poderá ser) obedecida. As exigências do CALEA são técnica e financeiramente dispendiosas – o que significa que a aplicação da lei é problemática.

## **Capítulo 4**

### ***Seção 4.1***

1. Um processo é a atividade de executar um algoritmo. Um programa é uma representação de um algoritmo.
2. No capítulo introdutório, citamos algoritmos para tocar música, operar lavadoras de roupa, construir modelos, executar truques de mágica e o algoritmo de Euclides. Muitos dos “algoritmos” que encontramos na vida cotidiana não o são, de acordo com nossa definição formal. O exemplo do algoritmo de divisão longa foi citado no texto. Outro é o executado por um relógio, que continua a avançar os seus ponteiros e a soar seu alarme dia após dia.

3. A definição informal não consegue exigir que os passos sejam ordenados e não-ambíguos. Ela somente sugere os requisitos necessários para que eles sejam executáveis e conduzam a um final.
4. Há dois pontos a serem analisados aqui. O primeiro é que as instruções definem um processo que não termina. Na realidade, porém, em última instância, o processo chegará ao estado em que não haverá mais moedas no seu bolso. De fato, este poderia ser o estado inicial. Nesse ponto, o problema é o da ambigüidade. O algoritmo, na forma como está representado, não fornece qualquer indicação sobre a conduta a adotar em tal situação.

### **Seção 4.2**

1. Um exemplo é encontrado na composição da matéria. Em um nível, as moléculas são consideradas primitivas, contudo, tais partículas são, na verdade, arranjos formados de átomos, que, por sua vez, são compostos de elétrons, prótons e nêutrons. Hoje, sabemos que até mesmo estas últimas “primitivas” são combinações de itens mais elementares ainda.
2. Uma vez que um procedimento esteja corretamente construído, poderá ser utilizado como bloco construtivo para estruturas maiores de programa, sem que para isso seja necessário reconsiderar a sua composição interna.
3.  $X \leftarrow$  maior entrada;  
 $Y \leftarrow$  menor entrada;  
Enquanto ( $Y \neq 0$ ) faça  
(Resto  $\leftarrow$  resto da divisão de  $X$  por  $Y$ ;  
 $X \leftarrow Y$ ;  
 $Y \leftarrow$  Resto);  
MDC  $\leftarrow X$
4. Todas as outras cores de luz podem ser reproduzidas combinando vermelho, azul e verde. Assim, um tubo de televisão é projetado para produzir estas três cores básicas.

### **Seção 4.3**

1. a. se ( $n = 1$  ou  $n = 2$ )  
então (a resposta é a lista que contém o único valor  $n$ )  
senão (Dividir  $n$  por 3, obtendo o quociente  $q$  e o resto  $r$ ).  
se ( $r = 0$ )  
então (a resposta é a lista que contém  $q$  números 3)  
se ( $r = 1$ )  
então (a resposta é a lista com ( $q - 1$ ) números 3 e dois números 2)  
se ( $r = 2$ )  
então (a resposta é a lista com  $q$  números 3 e um número 2)  
)  
b. O resultado seria a lista com 667 números 3.  
c. Você provavelmente fez uma série de experimentos com valores pequenos de entrada, até que começou a perceber um padrão.
2. a. Sim. Sugestão: Coloque a primeira peça no centro, para evitar o quadrante que contém o orifício, enquanto cobre um quadrado de cada um dos outros quadrantes. Assim, cada quadrante representará uma versão reduzida do problema original.  
b. Um tabuleiro com um único orifício contém  $2^{2n} - 1$  quadrados e cada peça cobre exatamente três quadrados.

- c. As partes (a) e (b) desta pergunta são um excelente exemplo de como saber a solução de um problema nos ajuda a resolver outro. Veja a quarta fase de Polya.
3. A mensagem diz: *This is the correct answer.*<sup>\*</sup>

#### Seção 4.4

1. Modifique o teste da instrução *while* para “valor desejado diferente da entrada corrente, e ainda há entradas a serem consideradas”.
2.  $Z \leftarrow 0;$   
 $X \leftarrow 1;$   
 Repete ( $Z \leftarrow Z + X;$   
 $X \leftarrow X + 1$ )  
 até ( $X = 6$ )
3. Isto é considerado um problema da linguagem C. Quando as palavras-chave *do* e *while* estão separadas por muitas linhas, os leitores do programa freqüentemente se confundem na interpretação correta da cláusula *while*. O *while* no fim de uma instrução *do* normalmente é interpretado como o início de uma instrução *while*. Assim, a experiência nos diz que é melhor usar diferentes palavras-chave para representar as estruturas *while* e *repeat*.
4. Cheryl            Alice            Alice  
 George            Cheryl            Bob  
 Alice              George            Cheryl  
 Bob                Bob                George
5. É desperdício de tempo insistir em colocar o pivô na posição acima de um elemento idêntico da lista. Por exemplo, faça a alteração proposta e, então, teste o novo programa com uma lista na qual todos os elementos sejam iguais.
6. procedimento ordena(Lista)  
 $N \leftarrow 1;$   
 enquanto ( $N$  for menor que o comprimento da Lista) faça  
 $(J \leftarrow N + 1;$  enquanto ( $J$  não for maior que o comprimento da Lista) faça  
 $(\text{se (o elemento na posição } J \text{ for menor que o elemento na posição } N\text{)} \\ \text{então (trocar os dois elementos)}}\\ J \leftarrow J + 1)\\ N \leftarrow N + 1)$
7. A solução a seguir é inefficiente. Você consegue torná-la mais eficiente?  
 procedimento ordena (Lista)  
 $N \leftarrow \text{comprimento da Lista};$   
 enquanto ( $N$  for maior que 1) faça  
 $(J \leftarrow \text{comprimento da Lista};$   
 $\text{enquanto (} J \text{ for maior que 1) faça}$   
 $(\text{Se (o elemento na posição } J \text{ for menor que o elemento na posição } J - 1)}$   
 $\text{então (trocar os dois elementos)}}\\ J \leftarrow J - 1)\\ N \leftarrow N - 1)$

---

<sup>\*</sup>N. de T. Em Português, *Esta é a resposta certa.*

**Seção 4.5**

1. O primeiro nome considerado seria Henry, o próximo, Larry e o último, Joe.
2. 8, 17

**Seção 4.6**

1. Se a máquina pode ordenar 100 nomes em um segundo, então ela realiza  $(10.000 - 100) / 4$  comparações em um segundo. Isso significa que cada comparação leva aproximadamente 0,0004 segundo. Em consequência, a ordenação de 1.000 nomes [que exige em média  $(1.000.000 - 100)/4$  comparações] levará 100 segundos ou 1 2/3 minutos.
2. A busca binária pertence a  $\Theta(\lg n)$ , a busca seqüencial, a  $\Theta(n)$ , e a ordenação por inserção pertence a  $\Theta(n^2)$ .
3. A classe  $\Theta(\lg n)$  é a mais eficiente, seguida pela  $\Theta(n)$ ,  $\Theta(n^2)$  e  $T(n^3)$ .
4. Não. A resposta não está correta, embora pareça. A verdade é que dois dos três cartões são iguais nos dois lados. Assim, a probabilidade de escolher tal cartão é dois terços.
5. Não. Se o dividendo for menor que o divisor, tal como em  $3/7$ , a resposta dada será 1, embora devesse ser 0.
6. Não. Se o valor de X for zero e o de Y, diferente de zero, a resposta dada não estará correta.
7. Todas as vezes que o teste de terminação for aplicado, a instrução “Soma = 1 + 2 +...+ I para I menor ou igual a N” será verdadeira. Combinando isto com a condição de terminação “I maior ou igual a N”, chega-se à conclusão desejada, “Soma = 1 + 2 +...+ N”. Dado que I é iniciado em zero e incrementado de uma unidade todas as vezes que executar o corpo da iteração, o seu valor deverá, em última instância, alcançar o valor N.
8. Infelizmente, não. Os problemas além do controle do *hardware* e do *software*, como falhas mecânicas e problemas elétricos, podem afetar a computação.

**Capítulo 5****Seção 5.1**

1. Um programa escrito em linguagem de terceira geração é independente de máquina no que se refere ao fato de seus passos não serem declarados em termos dos atributos da máquina, como registradores e endereços de células de memória. Entretanto, depende de máquina no que se refere à ocorrência de estouro aritmético e de erros de arredondamento, os quais continuarão acontecendo.
2. A principal diferença é que um montador traduz cada instrução do programa-fonte para uma única instrução de máquina, enquanto um compilador em geral produz diversas instruções em linguagem de máquina para obter o equivalente a uma única instrução de programa-fonte.
3. O paradigma declarativo se baseia no desenvolvimento de uma descrição do problema a ser solucionado, o paradigma funcional força o programador a descrever a solução dos problemas em termos das soluções de problemas menores e o orientado a objeto coloca ênfase na descrição dos componentes do ambiente do problema.

4. As linguagens de terceira geração permitem que o programa seja expresso mais nos termos do ambiente do problema e menos nos das ininteligibilidades do computador, como faziam as linguagens das primeiras gerações.

### Seção 5.2

1. Usar constantes descritivas pode tornar o programa mais acessível.
2. Uma instrução declarativa descreve uma terminologia, enquanto uma imperativa descreve os passos de um algoritmo.
3. Inteiro, real, caractere e booleano.
4. As estruturas *if-then-else* e *while* são muito comuns.
5. Todos os elementos de uma matriz homogênea têm o mesmo tipo.

### Seção 5.3

1. Uma variável local só é acessível dentro de uma unidade de programa, como um procedimento, enquanto uma variável global for acessível por toda a extensão do programa.
2. Uma função é um procedimento que retorna um valor associado ao nome da função.
3. Porque isto é o que elas são. Operações de entrada ou saída são, na realidade, chamadas das rotinas do sistema operacional da máquina.
4. Um parâmetro formal é um identificador interno de um procedimento. Serve como reserva de área para o valor (argumento) que é passado ao procedimento quando este for ativado.
5. Um procedimento é projetado para realizar uma ação, enquanto uma função, para produzir um valor. Assim, o programa ficará mais legível se o nome do procedimento refletir a ação que ele realiza, e o nome da função refletir o valor que ela retorna.

### Seção 5.4

1. *Análise lexical*: o processo de identificar símbolos.

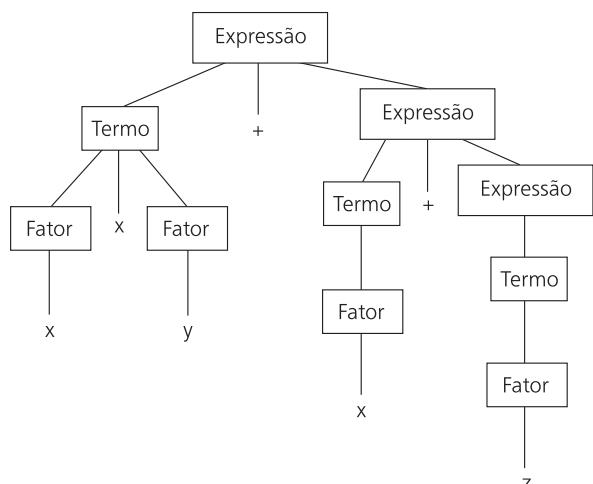
*Análise sintática*: o processo de reconhecer a estrutura gramatical do programa.

*Geração de código*: o processo de produzir as instruções do programa-objeto.

2. Uma tabela de símbolos é o registro da informação extraída das instruções declarativas do programa pelo analisador sintático.

3. Veja figura.

4. Eles são uma ou mais instâncias das subcadeias para a frente, para trás, cha, cha, cha, para trás, para a frente, cha, cha, cha, balançar para a direita, cha, cha, cha, balançar para a esquerda, cha, cha, cha,



**Seção 5.5**

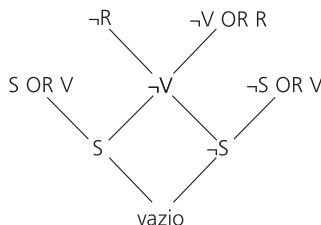
1. Uma classe é uma descrição de um objeto.
2. Uma provavelmente seria `ClasseMeteoro`, a partir da qual os vários meteoros seriam construídos. Na `ClasseLaser`, poderia existir uma variável de instância chamada `DirecaoAlvo`, indicando a direção em que o `laser` está apontando. Essa variável provavelmente seria usada pelos métodos `Disparar`, `VirarDireita` e `VirarEsquerda`.
3. A classe `Funcionario` pode conter as características de um empregado, tais como nome, endereço, anos de serviço, etc. A classe `FuncionarioTempoIntegral` pode conter características relativas a afastamentos remunerados e a classe `FuncionarioTempoParcial` pode conter características relativas a horas trabalhadas na semana, valor da hora etc.
4. Um construtor é um método especial em uma classe, que é executado sempre que uma instância da classe é criada.

**Seção 5.6**

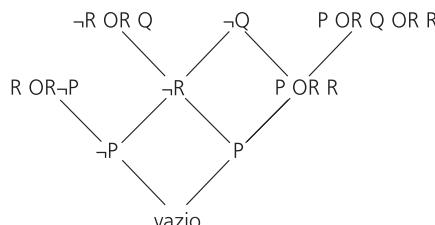
1. A lista incluiria técnicas para iniciar a execução de processos concorrentes e técnicas para implementar a comunicação entre processos.
2. Um deles coloca a responsabilidade nos processos, o outro coloca a responsabilidade nos dados. O último tem a vantagem de concentrar a tarefa em um único ponto do programa.

**Seção 5.7**

1. R, T e V. Por exemplo, mostramos que R é uma consequência, acrescentando ao conjunto a sua negação e mostrando que a resolução pode conduzir a uma declaração vazia, conforme mostrado na figura abaixo:



2. Não. O conjunto é inconsistente, dado que a resolução pode conduzir a uma declaração vazia, conforme ilustrado a seguir:



3. a. `maiseconomico (sue, carol)`  
`maiseconomico (sue, john)`

- b. maiseconomico (sue, carol)  
maiseconomico (bill, carol)
  - c. maiseconomico (carol, john)  
maiseconomico (bill, sue)  
maiseconomico (sue, carol)  
maiseconomico (bill, sue)  
maiseconomico (sue, john)
4. mãe(X, Y) :- progenitor(X, Y), fêmea (X).  
pai(X, Y) :- progenitor(X, Y). macho(X).

## Capítulo 6

### Seção 6.1

1. Uma seqüência longa de instruções de atribuição não é tão complexa, em termos de projeto de programa, quanto um pequeno conjunto de instruções *if* aninhadas.
2. Uma abordagem seria colocar intencionalmente alguns erros no *software* durante a sua fase de projeto. Então, depois que o *software* fosse supostamente bem depurado, conferir quantos dos erros originais ainda estariam presentes. Por exemplo, se 5 dos 7 erros originais tiverem sido removidos, então concluir-se-á que somente  $\frac{5}{7}$  dos erros totais do *software* foram removidos.
3. Que tal o número de erros encontrados depois de um período de uso? Um problema seria que tal valor não pode ser medido com antecedência.

### Seção 6.2

1. Os requisitos de um sistema são postos nos termos do ambiente da aplicação, enquanto as especificações o são em termos técnicos, e estas identificam de que maneira tais necessidades serão satisfeitas.
2. A fase de análise concentra-se em determinar o quê o sistema proposto deve realizar e a de projeto, em como o sistema realiza suas metas. A fase de implementação se concentra na construção propriamente dita do sistema e a de teste, em ter certeza de que o sistema faz o que dele se espera.
3. O tradicional modelo da cachoeira propõe que as fases de análise, projeto, implementação e teste sejam executadas de maneira seqüencial. O modelo de prototipação permite um método mais flexível, de tentativa e erro.

### Seção 6.3

1. Os capítulos de um romance são criados um a partir do outro, enquanto as seções de uma enciclopédia são bem independentes. Conseqüentemente, um romance apresenta mais correlação entre seus capítulos do que uma enciclopédia entre suas seções. Estas provavelmente têm um nível mais alto de coesão do que os capítulos de um romance.
2. O acoplamento explícito inclui o naipe de trunfo, a mão falsa, a ordem dos jogadores etc. Observações obtidas do processo de lanços, tais como quem possui quais cartas, podem ser consideradas como acoplamento implícito.

3. Isto é difícil. De um lado, poderíamos começar colocando tudo em um único módulo. Disso resultariam uma coesão pequena e nenhum acoplamento. Então, dividindo este único módulo em outros menores, o resultado seria um aumento do acoplamento. Logo, podemos concluir que quando cresce a coesão, o acoplamento tende a aumentar.

Entretanto, suponhamos que o problema em questão se divida, de modo natural, em três módulos bem coesos, que chamaremos A, B e C. Se o nosso projeto original não observar esta divisão natural (por exemplo, metade da tarefa A poderia ser colocada com metade da tarefa B e assim por diante), deveremos esperar uma baixa coesão e um alto acoplamento. Neste caso, reprojetar o sistema, isolando as tarefas A, B e C em módulos separados, provavelmente diminuiria o acoplamento entre os módulos, ao mesmo tempo em que aumentaria a coesão interna dos módulos.

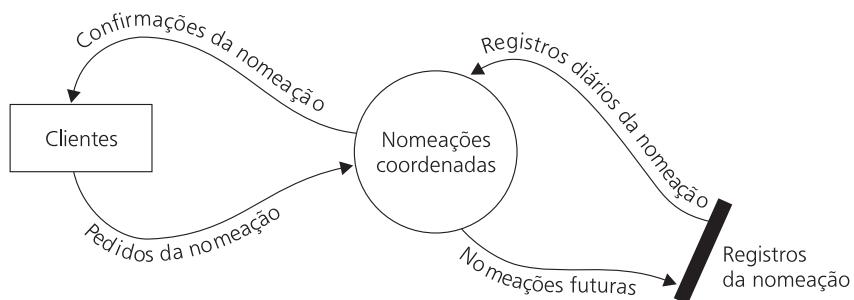
4. Para dar um toque pessoal, um objeto do tipo *Catálogo* poderia usar o nome do cliente quando se comunica com ele. Para isso, o objeto *Catálogo* precisaria enviar uma solicitação ao objeto *Cliente* para obter o seu nome.

#### **Seção 6.4**

1. O modelo cliente/servidor seria uma.
2. Os pesquisadores esperam que as armações sirvam como blocos construtores pré-fabricados, a partir dos quais grandes sistemas de *software* podem ser construídos, de maneira similar à construção de dispositivos complexos a partir de componentes de prateleira, como ocorre em outras disciplinas da engenharia.
3. A prototipação evolucionária tradicional é realizada dentro da organização que desenvolve o *software*, enquanto o desenvolvimento de código aberto não se restringe a uma organização. No caso do desenvolvimento de código aberto, a pessoa que supervisiona o desenvolvimento não necessariamente determina os aperfeiçoamentos a serem feitos, enquanto na prototipação evolucionária tradicional, a pessoa que gerencia o desenvolvimento do *software* atribui pessoal a tarefas específicas de aperfeiçoamento.

#### **Seção 6.5**

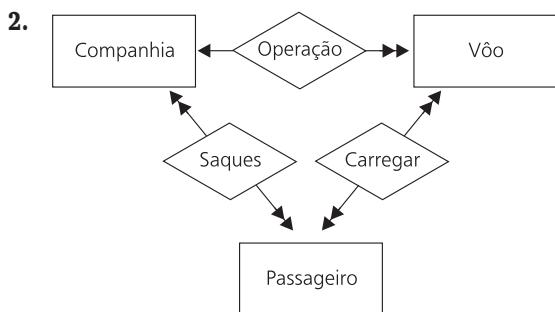
- 1.



2. Veja figura no alto da página seguinte.
3. Essa questão deve ser refeita após a leitura do Capítulo 7. A implementação de um relacionamento de um para muitos exige uma estrutura como uma lista ligada, onde todas as entidades relacionadas podem ser encontradas, enquanto um relacionamento de um para um pode ser implementado com um único ponteiro de uma entidade para a relacionada.

## Seção 6.6

1. O objetivo de testar *software* é encontrar erros. Portanto, os engenheiros de *software* geralmente consideram que um teste que não revela erros falhou.
2. Uma seria considerar a quantidade de desvios nos módulos. Por exemplo, um procedimento com numerosos laços e instruções *if-then-else* seria provavelmente mais suscetível a erros do que um módulo com uma estrutura lógica simples.
3. A análise de valores limites sugere que se teste o *software* tanto em uma lista com 100 elementos, como em uma vazia. Você também pode realizar um teste com uma lista que já se encontra em ordem.



## Seção 6.7

1. Nos manuais que acompanham o *software*, internamente no programa-fonte na forma de comentários e códigos bem redigidos, por meio de mensagens interativas que o próprio programa escreve no terminal, por meio de dicionários de dados e na forma de documentos de projeto, como diagramas estruturais, de classes, de fluxo de dados e diagramas completos das relações.
2. Tanto na fase de desenvolvimento como na de modificação. O ponto é que as modificações devem estar documentadas de forma tão completa quanto o programa original. (Também é verdade que o *software* é documentado durante a sua fase de uso. Por exemplo, um usuário do sistema pode descobrir problemas, os quais são publicados no manual do sistema do usuário. Além disso, são comuns livros sobre o uso e o projeto de sistemas populares de *software*. Eles geralmente são escritos por pessoas que não são os seus projetistas originais, depois de o *software* ter sido utilizado durante algum tempo e conquistado popularidade.)
3. Diferentes pessoas têm diferentes opiniões. Algumas argumentam que o programa é o ponto de todo um projeto e, portanto, é naturalmente o mais importante. Outros argumentam que um programa nada vale se não estiver documentado, pois se você não entender um programa, não poderá usá-lo nem modificá-lo. Além disso, com uma boa documentação, a tarefa de criar o programa pode ser “facilmente” reexecutada.

## Seção 6.8

1. Os tribunais têm interpretado a similaridade de uma maneira bem ampla, considerando pontos muito além dos envolvidos nos componentes literais do programa. Os componentes “não-literais” que têm sido considerados incluem a estrutura do programa, registros de projeto, a aparência e o jeito do programa.
2. Leis de direitos autorais e patentes beneficiam a sociedade, pois estimulam os criadores de novos produtos a torná-los disponíveis ao público. Leis de segredos industriais beneficiam a sociedade, pois permitem a uma empresa proteger de seus competidores as etapas do desenvolvimento de um produto. Sem tal proteção, as empresas hesitariam em fazer grandes investimentos em novos produtos.
3. Uma retratação não protege uma empresa contra negligência.

## Parte III

### Capítulo 7

#### Seção 7.1

1. Se você fosse escrever um programa para jogar damas, a estrutura de dados para representar o tabuleiro provavelmente seria estática, uma vez que o seu tamanho não muda durante o jogo. Entretanto, se fosse escrever um programa para jogar dominó, a estrutura de dados que representa o padrão de dominós construído sobre a mesa provavelmente seria dinâmica, uma vez que esse padrão varia em tamanho e não pode ser predeterminado.
2. Um catálogo telefônico é essencialmente uma coleção de ponteiros para pessoas. Os vestígios encontrados na cena do crime são ponteiros para o criminoso (talvez codificados).

#### Seção 7.2

1. 5 3 7 4 2 8 1 9 6
2. Se R for o número de linhas da matriz, a fórmula será  $R(J - 1) + (I - 1)$ .
3. A partir do endereço inicial 25, devemos saltar  $11(3 - 1) + (6 - 1) = 27$  elementos da matriz, sendo que cada um ocupa duas células de memória. Assim, devemos saltar 54 células de memória. Portanto, o endereço final pode ser encontrado somando-se 54 ao endereço do primeiro elemento, o que nos fornecerá o endereço 79.
4.  $(C - I) + J$

#### Seção 7.3

1. Como exemplo, para encontrar o quinto elemento de uma lista densa, multiplique o número de células de cada elemento por 4 e some o resultado ao endereço do primeiro elemento. A situação é bem diferente no caso da lista ligada, pois o endereço do quinto elemento não tem relação alguma com o do primeiro. Assim, para encontrar o quinto elemento, deve-se, de fato, percorrer cada um dos elementos que o precedem.
2. O ponteiro para o início da lista contém o valor NIL.
3. Ultimo ← Último nome a ser impresso  
Terminado ← Falso  
Ponteiro Corrente ← ponteiro para o início da lista;  
enquanto (Ponteiro Corrente não for NIL e Terminado = falso) faça  
(imprimir o elemento apontado pelo Ponteiro Corrente,  
se (o nome que acabou de ser impresso = Ultimo)  
então      (Terminado ← verdadeiro)  
Ponteiro Corrente ← o valor contido na célula de ponteiro do  
elemento apontado pelo Ponteiro Corrente)
4. procedimento apaga (Elemento)  
Corrente ← InícioDaLista  
Anterior ← NIL  
Encontrado ← falso  
enquanto (Corrente não é NIL e Encontrado = falso) faça  
se (elemento apontado por Corrente for o elemento desejado)

```

então (Encontrado ← verdadeiro)
senão (Anterior ← Corrente;
Corrente ← o valor contido na célula de ponteiro do
elemento apontado por Corrente)
se (Encontrado for verdadeiro)
então (se (Anterior = NIL)
então (InícioDaLista ← o valor contido na célula de ponteiro do
elemento apontado por Corrente)
senão (Valor contido na célula de ponteiro
apontado por Anterior ← o valor contido na célula de
ponteiro do elemento apontado por Corrente))

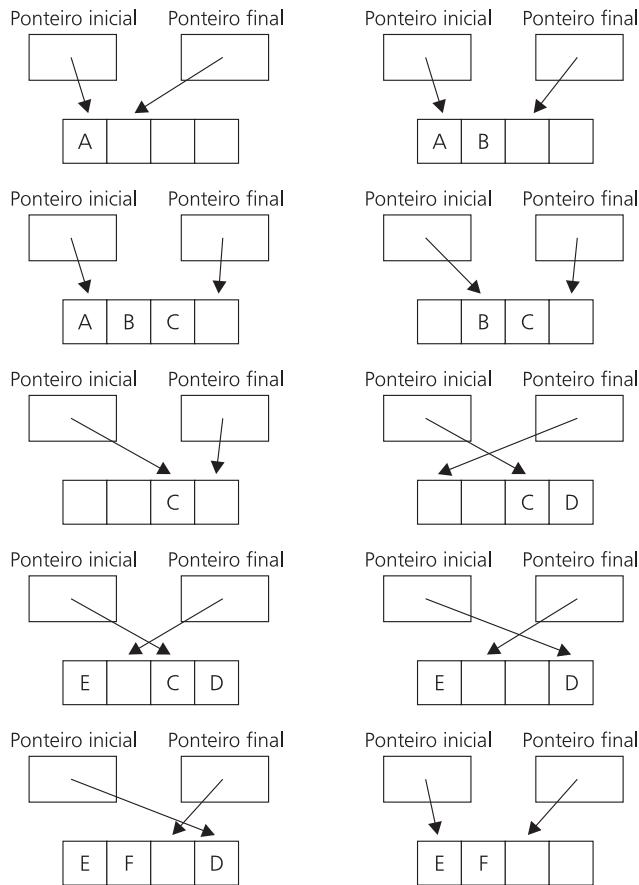
```

## Seção 7.4

1. Um exemplo tradicional é a pilha de bandejas em uma lanchonete. Muitos destes aparelhos são equipados com molas para manter a bandeja do topo em um nível conveniente. Neste caso, o termo empilhar significa realmente acrescentar mais um elemento à pilha.
2. Quando o programa principal chamar o subprograma A, a pilha conterá a “posição de retorno” na unidade do programa principal.  
Quando a sub-rotina A chamar a sub-rotina B, a pilha conterá a “posição de retorno” em A em cima da “posição de retorno” da unidade do programa principal.  
Após o término da sub-rotina B, a pilha conterá apenas a “posição de retorno” da unidade do programa principal.  
Quando a sub-rotina A chamar a sub-rotina C, a pilha conterá a “posição de retorno” em A em cima da “posição de retorno” da unidade do programa principal.  
Após o término da sub-rotina C, a pilha conterá apenas a “posição de retorno” da unidade do programa principal.  
Após o término da sub-rotina C, a pilha estará vazia.
3. O ponteiro da pilha aponta para a célula imediatamente abaixo da fim da fila.
4. procedimento desempilha ()
 se (o ponteiro da pilha apontar abaixo do fim da fila)
 então (terminar com uma mensagem de erro)
 Extrair da pilha o elemento apontado pelo ponteiro da pilha;
 Ajustar o ponteiro da pilha para apontar para o elemento imediatamente inferior da pilha
5. Represente a pilha como uma matriz unidimensional e o ponteiro da pilha como uma variável do tipo inteiro. Em seguida, utilize este ponteiro para manter um registro da posição relativa do topo da pilha na matriz, em vez do endereço absoluto na memória.

## Seção 7.5

1. Veja figura no alto da página seguinte.
2. As condições vazia e lotada são indicadas pela coincidência dos ponteiros para o topo e para a fim da fila. Assim, torna-se necessária alguma informação adicional para distinguir entre as duas condições.
3. procedimento insere (NovoElemento)
 se (Lotada) então (terminar, emitindo mensagem de erro)
 Armazenar NovoElemento na posição indicada pelo ponteiro para o fim da fila;

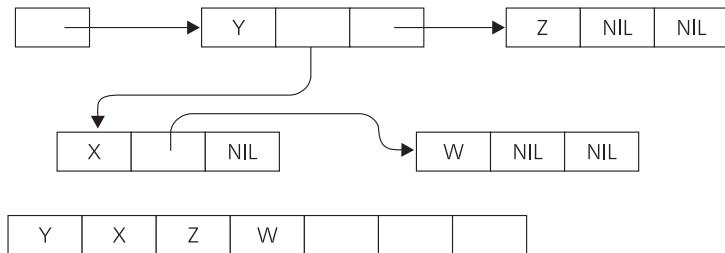


Avançar o ponteiro para o fim da fila;  
se (o ponteiro para o fim da fila apontar para fora do bloco reservado)  
então (alterar o ponteiro para o fim da fila, para que aponte para a  
primeira célula do bloco reservado)  
se (ponteiro para o início da fila = ponteiro para o fim da fila)  
então (Lotada ← verdadeiro)

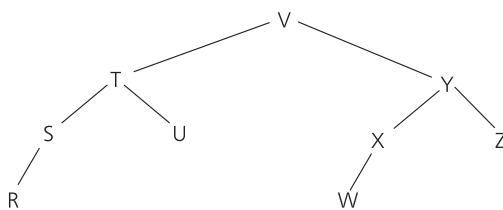
### Seção 7.6

1. A raiz é 11, os nós-folha são 1, 2, 6, 3 e 4. Há quatro subárvore (não-vazias) abaixo do nó 9, com raízes 5, 1, 2 e 6. Os nós 9 e 10 são irmãos, bem como 5 e 6, 1 e 2, 7 e 8.
2. O ponteiro da raiz é NIL.
3. Veja figura no alto da página seguinte.

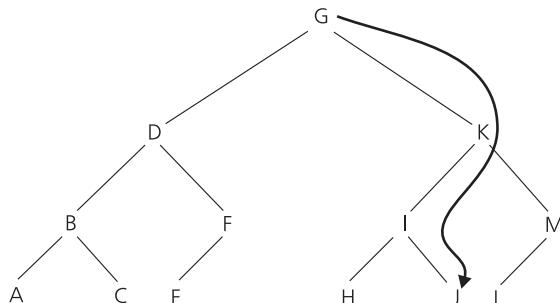
Ponteiro da raiz



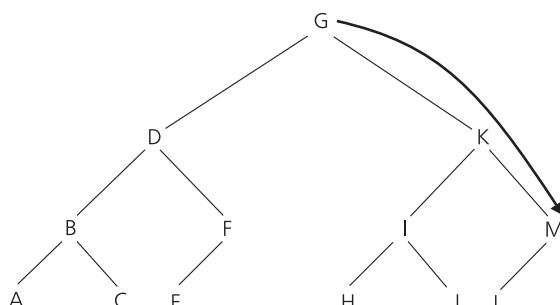
4.



5. Quando estiver buscando J:



Quando estiver buscando P:



**6.**

```
procedure ImprimeArvore(Arvore)
se (ponteiro do nó-raiz de Arvore não é Nil)
então (Aplicar o procedure ImprimeArvore à
árvore que aparece como o filho
esquerdo em Arvore;
Imprimir nó-raiz de Arvore;
Aplicar o procedure ImprimeArvore à árvore que
aparece como o filho direito em Arvore.)
```

```
procedure ImprimeArvore(Arvore)
se (ponteiro do nó-raiz de Arvore não é Nil)
então (Aplicar o procedure ImprimeArvore à
árvore que aparece como o filho
esquerdo em Arvore;
 Imprimir nó-raiz de Arvore;
 Aplicar o procedure ImprimeArvore à árvore que
aparece como o filho direito em Arvore.)
```

Aqui, quando K  
é impresso

7. Em cada nó, cada ponteiro para os filhos poderia ser usado para representar uma única letra do alfabeto. Uma palavra poderia ser representada por um caminho de cima para baixo na árvore, ao longo da seqüência de ponteiros, representando a soletração da palavra. Um nó será marcado de maneira especial se representar o fim de uma palavra soletrada corretamente.

**Seção 7.7**

1. Um tipo de dados é um gabarito, enquanto uma instância daquele tipo de dados é uma entidade real, construída a partir do gabarito. Por exemplo, cachorro é um tipo de animal, enquanto Lassie e Rex são instâncias deste tipo.
2. Tanto os tipos definidos pelo usuário como as classes são gabaritos. Entretanto, os tipos definidos pelo usuário apenas permitem ao programador descrever os gabaritos de dados. As classes, porém, permitem ao programador escrever gabaritos que envolvem procedimentos.
4. Alguns itens em uma classe são designados privados, para impedir que outras unidades de programa tenham acesso direto a eles. Se um item for privado, então a repercussão de alterá-lo deve ficar restrita ao interior da classe.
5. Uma fila de inteiros pode ser implementada utilizando como estrutura subjacente uma lista contígua ou uma lista ligada, ou talvez até uma fila circular, restrita a um bloco específico de células de memória ou a um bloco arbitrário de células de memória, embora esta última implementação possa vir a ser perigosa para as demais estruturas de dados residentes na memória.

**Seção 7.8**

1. a. A5                  b. A5                  c. A5
2. D50F, 2EFE, 5FFE
3. 2EA0, 2FB0, 2101, 20B4, D50E, E50F, 5EE1, 5FF1, DF14, B008, C000
4. Quando se percorre uma lista ligada na qual cada elemento consiste em duas células de memória (uma célula de dados seguida de um ponteiro para o próximo elemento), uma instrução no formato DR0S poderia ser usada para recuperar os dados, e no formato DR1S para recuperar o ponteiro para o próximo elemento. Se o formato DRTS fosse usado, a célula de memória que estivesse sendo referenciada seria ajustada modificando-se o valor no registrador T.

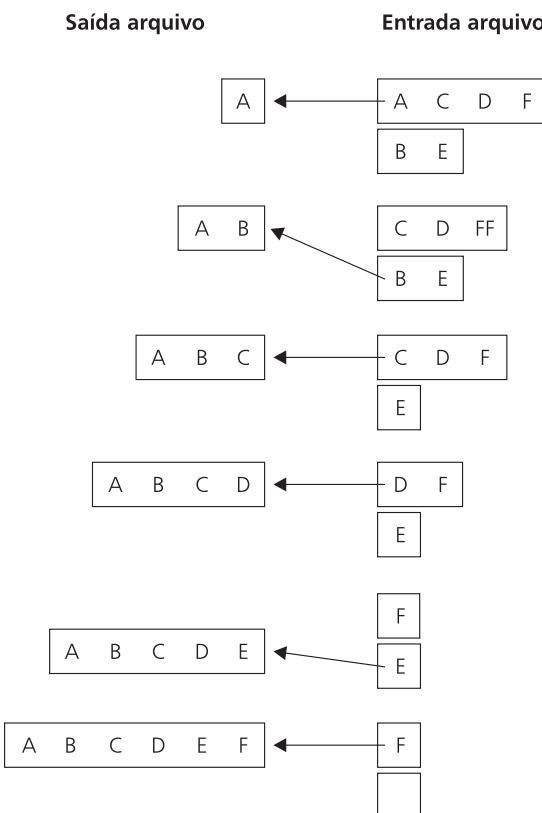
## Capítulo 8

### Seção 8.1

1. Os registros físicos são lidos do armazenamento em massa, em uma área de retenção de onde o programa de aplicação tem acesso aos dados em termos de registros lógicos.
2. Um descritor de arquivo é uma tabela que mantém a informação necessária a um sistema operacional para manipular o arquivo.
3. O gerente de arquivos é que constrói o descritor.

### Seção 8.2

1. Você deveria passar pelos seguintes estágios:



2. A idéia é primeiramente dividir o arquivo a ser ordenado em vários arquivos separados, sendo que cada qual contém um registro. A seguir, agrupar aos pares os arquivos com um registro e, para cada par, aplicar o algoritmo de intercalação. Isto diminuiria para a metade o número de arquivos, cada qual com dois registros. Além disso, cada arquivo de dois registros está ordenado. Podemos agrupá-los em pares e novamente aplicar o algoritmo de intercalação para os pares. Mais uma vez, teremos menos arquivos, porém maiores e ordenados. Continuando desta forma, teremos, em última instância, apenas um arquivo com todos os registros originais ordenados. (Se ocorrer um número ímpar de arquivos em alguma fase deste processo,

precisaremos apenas deixar de lado o arquivo sem par, para mais tarde agrupá-lo com um dos arquivos maiores da próxima fase.)

3. Se o arquivo for armazenado em fita ou CD, a sua organização física provavelmente será seqüencial. Contudo, se for armazenado em disco magnético, provavelmente estará distribuído em vários setores no disco, e a natureza seqüencial do arquivo é uma propriedade conceitual mantida por um sistema de ponteiros ou alguma forma de lista na qual os setores onde o arquivo está armazenado serão registrados.
4. Um arquivo texto é essencialmente um arquivo seqüencial no qual cada registro consiste em um único símbolo.
5. Normalmente, uma boa parte do documento é mantida na memória principal, o que permite acesso direto a essa parte do documento. À medida que a parte posterior do documento vai sendo processada, a inicial pode ser colocada no armazenamento em massa. Assim, mais tarde, se desejar atualizar esta parte anterior e ela estiver armazenada como um arquivo texto, o processador deverá ler o documento desde o início para localizar a parte a ser atualizada.
6. O teclado produziria os padrões 00110010 e 00110100, que representam os caracteres 2 e 4. A variável *Idade* receberia o padrão de bits 0000000000011000, que é a representação de 24 em complemento de dois.
7. O espaço para o arquivo no armazenamento em massa (disco) é alocado em setores ou em coleções de setores chamadas aglomerados. Assim, o tamanho do arquivo muda em função dessas unidades, e não em função de caracteres.

### **Seção 8.3**

1. Primeiro, o sistema operacional procura no índice qual o segmento a ser consultado. Após ter definido o número do segmento desejado, o sistema operacional pode verificar se o segmento já se encontra na memória principal (pode ser o mesmo segmento anteriormente acessado). Se já estiver lá, o sistema operacional buscará nele o registro correto e o transmitirá para o programa. Caso contrário, o segmento será trazido para a memória a partir do armazenamento em massa e depois consultado.
2. Em um ambiente de tempo compartilhado, o sistema operacional faz o máximo para utilizar eficientemente todo o tempo. Se o segmento necessário de disco já não estiver na memória principal, o sistema operacional solicitará ao controlador do disco que obtenha o segmento correto e, enquanto isso, em vez de esperar pela chegada dos dados, encerrará a fatia de tempo do processo original e iniciará outro processo. Depois que o controlador tiver colocado na memória principal o segmento solicitado, o sistema operacional irá retornar ao processo original, fornecer-lhe o registro desejado e autorizá-lo a continuar a sua execução na sequência normal das fatias de tempo.
3. Um índice de arquivo é essencialmente um arquivo. Assim, podemos pensar em um arquivo indexado como dois arquivos — o índice e o arquivo associado. O índice obviamente deve ser menor que o arquivo associado. Além disso, o índice provavelmente seria lido em sua totalidade na memória principal quando o arquivo fosse aberto pela primeira vez, enquanto apenas partes do arquivo associado seriam transferidas para a memória principal, quando necessário.
4. O índice (se tiver sido modificado por adições e eliminações de registros) deve ser gravado no dispositivo de armazenamento em massa, juntamente com o arquivo.

## Seção 8.4

1. Este é um bom exemplo dos detalhes que devem ser considerados ao selecionar um algoritmo de *hash*. Neste caso, utilizar os três primeiros dígitos do número do seguro social é uma escolha infeliz, pois tais dígitos estão associados às regiões do país. Por conseguinte, cidadãos de uma mesma região do país apresentam os mesmos dígitos no início dos seus números de seguro social, e isto resultaria no aparecimento de um número maior de colisões do que seria normal em arquivos *hash*.
2. Um algoritmo *hash* com escolha imprópria resultaria em maior número de colisões e, portanto, no aparecimento de estouros em maior quantidade. Como o transbordamento proveniente de cada seção de armazenamento em massa é organizado como uma lista ligada, fazer busca em registros de transbordamento é essencialmente fazer busca em um arquivo seqüencial.
3. Os baldes associados são:

|      |      |      |
|------|------|------|
| a. 0 | b. 0 | c. 3 |
| d. 0 | e. 3 | f. 3 |
| g. 3 | h. 3 | i. 3 |
| j. 0 |      |      |

Assim, todos os registros são mapeados nos baldes 0 e 3, deixando os baldes 1, 2, 4 e 5 vazios. O problema aqui é que o número de baldes usados (6) e os valores de campos chave têm o fator comum igual a 3. (Você pode refazer estes valores de campos chave utilizando 7 baldes e verificar as melhorias obtidas.)

4. O ponto aqui é que estamos essencialmente aplicando um algoritmo de *hash* para colocar as pessoas de um grupo em uma das 365 categorias. Obviamente, o algoritmo de *hash* é o cálculo da data de aniversário da pessoa. O fato surpreendente é que são necessárias apenas 23 pessoas para que seja provável a ocorrência de pelo menos dois aniversariantes no mesmo dia. Em termos de um arquivo *hash*, isto indica que, quando se dividem registros, por meio de *hash*, em 365 baldes disponíveis em disco, é provável que ocorram colisões após a entrada de apenas 23 registros.
5. Um objeto do tipo `Properties` é essencialmente uma instância da `Hashtable` que pode ser guardada no armazenamento em massa. Com as grandes memórias atuais, isso é prática corrente, ou seja, o que normalmente seria implementado com arquivo *hash* é feito com tabela *hash*.

## Capítulo 9

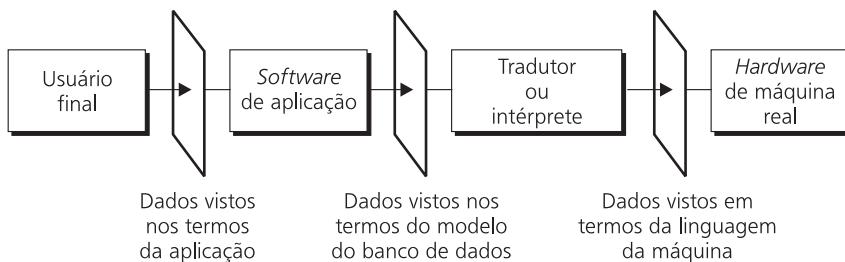
### Seção 9.1

1. O departamento de compras estaria interessado em registros do cadastro de matéria-prima para fazer novos pedidos, enquanto o de contabilidade precisaria da informação para fazer o balanço dos livros fiscais.
2. Os funcionários, estudantes, bacharéis, financiamentos, inscrições, materiais/equipamentos e assim por diante.
3. O subesquema para o departamento de compras provavelmente incluiria os endereços dos vários fornecedores de peças no cadastro e talvez o nome do representante de vendas de cada empresa. O subesquema para o departamento de contabilidade dificilmente incluiria essa informação.

### Seção 9.2

1. Não. O uso de sistemas de arquivo dita, invariavelmente, que o programa de aplicação seja expresso em termos da organização real dos registros no arquivo. Assim, uma mudança na estrutura do registro requer mudanças em todos os programas que acessam o arquivo.

2.



3. O *software* de aplicação traduz as solicitações do usuário, feitas na terminologia da aplicação, para uma terminologia compatível com o sistema de gerência do banco de dados. Este converte os pedidos para uma forma compreendida pelas rotinas que, de fato, manipularão os dados no armazenamento em massa. Estas últimas rotinas executam a recuperação propriamente dita dos dados.

### Seção 9.3

1. a. G. Jerry Smith  
b. Cheryl H. Clark  
c. S26Z

2. Uma solução seria:

```

TEMP ← SELECT from CARGO
 where Departamento = "PESSOAL"
LISTA ← PROJECT NomeDoCargo from TEMP

```

Em alguns sistemas, disto resultaria uma lista com cargos repetidos, dependendo do número de suas ocorrências no Departamento de Pessoal. Assim, a nossa lista poderia conter numerosas ocorrências do cargo de secretaria. No entanto, é mais comum projetar a operação PROJECT para que ela remova as ênuplas duplicadas da relação resultante.

3. Uma solução seria:

```

TEMP1 ← JOIN CARGO and ATRIBUICAO
 where CARGO.IdCargo = ATRIBUICAO.IdCargo
TEMP2 ← SELECT from TEMP1
 where DataSaida = "*"
TEMP3 ← JOIN FUNCIONARIO and TEMP2
 where FUNCIONARIO.IdFuncionario = TEMP2.IdFuncionario
RESULTADO ← PROJECT Nome,
 Departamento from TEMP3

```

4. select NomeDoCargo
 from CARGO
 where Departamento = "PESSOAL"
Select FUNCIONARIO.Nome, CARGO.Departamento

```
from CARGO, ATRIBUICAO, and FUNCIONARIO
where (CARGO.IdCargo = ATRIBUICAO.IdCargo) and
 (ATRIBUICAO.IdFuncionario = FUNCIONARIO.IdFuncionario
 and (ATRIBUICAO.DataSaida = "*"))
```

5. O próprio modelo não garante a independência de dados. Esta é uma propriedade do sistema de gerência de dados. A independência de dados é obtida dando ao sistema de gerência de dados a capacidade de apresentar uma organização relacional consistente ao *software* da aplicação, ainda que a organização real possa se alterar.
6. Por meio de atributos comuns. Por exemplo, a relação FUNCIONARIO desta seção está atrelada pelo atributo IdFUNCIONARIO à relação ATRIBUICAO, e esta, à relação CARGO pelo atributo IdFUNCIONARIO. Atributos utilizados para conectar relações como estes às vezes são chamados atributos de conexão.

## Seção 9.4

1. Pode haver métodos para associar e obter DataEntrada e DataSaida. Um outro método pode ser disponibilizado para informar o tempo total de serviço.
2. Uma forma seria estabelecer um objeto para cada tipo de produto do cadastro. Cada objeto poderia manter o cadastro completo do produto correspondente, o seu custo e indicações das solicitações de compra mais significativas desse produto.
3. Conforme foi indicado no início desta seção, bancos de dados orientados a objeto parecem manipular tipos de dados compostos com mais facilidade que os bancos de dados relacionais. Além disso, o fato de os objetos poderem conter métodos que desempenham um ativo papel na resposta às consultas promete dar aos bancos de dados orientados a objeto uma vantagem sobre os relacionais, cujas relações meramente mantêm os dados.

## Seção 9.5

1. Uma vez que uma transação alcance o seu ponto de comprometimento, o sistema de gerência do banco de dados aceita a responsabilidade de providenciar que a transação completa seja executada no mesmo. Uma transação que não tenha alcançado o seu ponto de comprometimento não tem tal garantia. Se surgir algum problema, a transação terá de ser ressubmetida.
2. Um método seria deixar, por um momento, de entrelaçar transações, de forma que todas as transações correntes possam ser totalmente completadas. Isto estabelece um ponto no qual um futuro desfazer de transações (*rollback*) em cascata será encerrado.
3. Resultaria em um saldo de \$100 se as transações fossem executadas uma de cada vez, ou em um saldo de \$200 se a primeira transação fosse executada depois que a segunda tivesse obtido o resultado do saldo original, mas antes que esta armazenasse o novo saldo. Resultaria em um saldo de \$300 se a segunda transação fosse executada depois que a primeira tivesse obtido o resultado do saldo original, mas antes que esta armazenasse o novo saldo.
4. a. Se nenhuma outra transação tiver acesso exclusivo, o acesso compartilhado será concedido.  
b. Se outra transação já tiver obtido alguma forma de acesso, o sistema de gerência do banco de dados normalmente fará a nova transação aguardar, ou então desfará as outras transações e dará acesso à nova.
5. Ocorreria enlace mortal se cada transação adquirisse acesso exclusivo a diferentes dados e, em seguida, cada uma solicitasse acesso ao dado da outra.

6. O enlace mortal anteriormente citado poderia ser removido desfazendo-se uma das transações (usando o diário) e dando à outra acesso ao dado que estava de posse da transação desfeita.

### ***Seção 9.6***

1. O ponto aqui é comparar a sua resposta a esta questão e à próxima. As duas tratam essencialmente do mesmo problema, mas em contextos diferentes.
2. Veja o problema anterior.
3. Você pode receber propagandas ou avisos de oportunidades que de outra forma não receberia, mas também se tornar alvo de diversas solicitações, ou mesmo vítima de crime.
4. O ponto a ser salientado aqui é que a imprensa livre pode alertar o público quanto a abusos ou abusos em potencial, e assim trazer a opinião pública à ação. Na maioria dos casos relatados no texto, foi a imprensa livre que iniciou a ação corretiva, alertando o público.

## **Parte IV**

### **Capítulo 10**

#### ***Seção 10.1***

1. O nosso objetivo aqui não é dar uma resposta definitiva a este assunto, mas usá-lo para mostrar o quanto a argumentação sobre a existência de inteligência realmente é delicada.
2. Embora a maioria de nós provavelmente dissesse não, é possível que argumentássemos que, se um ser humano liberasse os mesmos produtos em um ambiente semelhante, a consciência estaria presente, embora não sejamos capazes de explicar a diferença.
3. Não existe resposta correta ou errada. A maioria concordaria em que a máquina, no mínimo, pareceria inteligente.

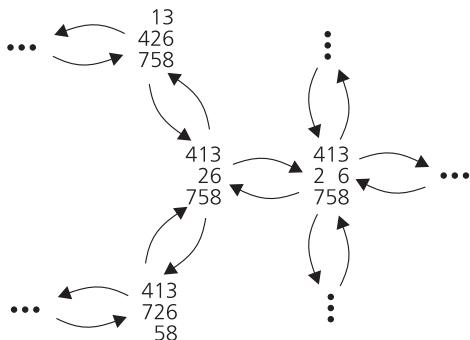
#### ***Seção 10.2***

1. No caso do controle remoto, o sistema necessita apenas retransmitir a imagem, enquanto para usar a figura na realização de manobras, o robô deverá ser capaz de “entender” o seu significado.
2. As possíveis interpretações de um pedaço do desenho não combinam com qualquer uma de outro pedaço. Para embutir esta percepção em um programa, pode-se isolar as interpretações possíveis para várias junções de linha e depois escrever um programa que tentasse encontrar um conjunto de interpretações compatíveis (uma para cada junção). De fato, se você parar para pensar sobre isto este provavelmente seria o nosso procedimento ao avaliar o desenho. Você já percebeu que os seus olhos esquadrinham de um lado para outro entre as duas extremidades do desenho enquanto suas percepções procuram integrar as possíveis interpretações? (Se este assunto for do seu interesse, leia os trabalhos de D. A. Huffman, M. B. Clowes e D. Waltz.)
3. Existem quatro blocos na pilha, mas apenas três estão visíveis. O ponto é que o entendimento desse conceito aparentemente simples exige uma “inteligência” significativa.

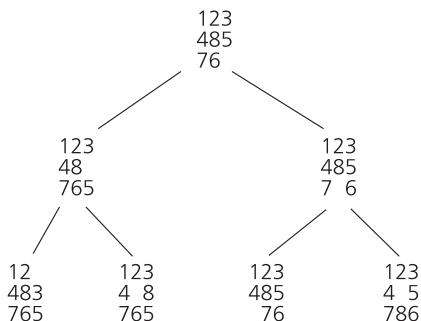
### Seção 10.3

- Sistemas de produções fornecem uma abordagem uniforme para o tratamento de diversos problemas. Assim, embora aparentemente diferentes nas suas formas originais, todos os problemas reformulados em termos de sistemas de produções recaem no problema de encontrar uma trajetória em um grafo de estados.

2.



- A árvore apresenta profundidade de quatro movimentos. A parte superior aparece conforme a seguir:

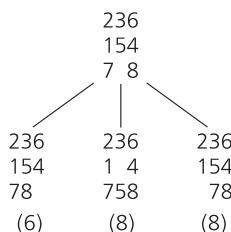


- A tarefa exige muito papel e muito tempo.
- O nosso sistema heurístico para resolver o quebra-cabeças de oito peças está baseado em uma análise da situação imediata, similar ao caso do alpinista. Esta visão curta inicialmente conduziu o nosso algoritmo a percorrer a trajetória errada do exemplo desta seção, da mesma forma como um alpinista poderia ser sempre conduzido a situações perigosas por delineiar, um caminho baseado apenas no terreno local. (Geralmente, esta analogia dá aos sistemas heurísticos, baseados em informação local ou imediata, o nome de sistemas de escalar colinas.)
- O sistema gira as peças 5, 6 e 8, em sentido horário ou anti-horário, até que seja alcançado o estado-meta.
- O problema aqui é que o nosso esquema heurístico não considera a importância de manter vago um espaço adjacente às peças que estão fora de posição. Se tal espaço for rodeado de peças já em suas posições corretas, algumas delas deverão ser movidas, de modo a deslocar as que estiverem incorretamente posicionadas. Assim, seria errado considerar, *a priori*, que todas as peças que circundam o espaço vago estariam, de fato, em suas posições corretas. Para corrigir essa falha, poderíamos primeiramente observar que uma peça, em sua posição correta, mas

bloqueando o espaço vago de peças incorretamente posicionadas, deve ser deslocada para longe de sua posição correta e devolvida a ela mais tarde. Assim, cada peça corretamente posicionada em uma trajetória entre o espaço vago e a peça mais próxima que estiver incorretamente posicionada responde por, pelo menos, dois movimentos na solução existente até este instante. Logo, poderíamos modificar o nosso cálculo de custo projetado para a seguinte forma:

Primeiramente, calcule o custo projetado como era feito anteriormente. Se o espaço vago estiver totalmente isolado das peças incorretamente posicionadas, encontre um caminho mínimo entre o espaço e uma peça incorretamente posicionada, multiplique o número de peças deste caminho por dois e some o valor resultante ao custo projetado anterior. Com esse sistema, os nós-folha da Figura 10.10 projetariam custos iguais a 6, 6 e 4 (da esquerda para a direita) e, assim, o ramo correto já seria percorrido inicialmente.

O nosso novo sistema não é simples. Por exemplo, suponha a seguinte configuração. A solução é deslizar para baixo a peça 5, girar a peça do topo para duas linhas em sentido horário até que estejam em posições corretas, mover a peça 5 de volta para cima e finalmente mover a peça 8 para a sua posição correta. Contudo, a nossa nova heurística exige que iniciemos movendo a peça 8, pois o estado obtido a partir do movimento inicial tem um custo projetado de apenas 6, menor que as demais opções que têm custo 8.



## Seção 10.4

1. Todos os padrões produzem uma saída igual a 0, com exceção do padrão 1, 0, que produz uma saída igual a 1.
2. Atribua peso 1 a cada entrada, e à unidade, um valor de referência igual a 1,5.
3. Projete uma rede de dois níveis, conforme descrito no texto. A unidade de nível inferior deve atribuir às suas entradas o peso 1 e ter um valor de referência igual a  $7\frac{1}{2}$ . Assim, uma destas unidades produzirá uma saída igual a 1 se o padrão no campo de visão for o círculo; caso contrário, todas as unidades de nível inferior produzirão uma saída igual a 0. Entretanto, se a unidade de nível superior atribuir às suas entradas o peso 1 e tiver um valor de referência igual a  $\frac{1}{2}$ , a rede inteira produzirá saída 1 quando o padrão for um círculo e 0 quando for um X.
4. Projete uma rede de dois níveis, conforme descrito no texto. Cada unidade de nível inferior deverá ter um valor de referência igual a  $2\frac{1}{2}$  e atribuir peso 0 aos quadrados dos cantos do seu campo de visão e peso 1 aos demais. A única forma de um destes elementos produzir saída 1 é ter o padrão de C no seu campo de visão. De fato, se este for o padrão, duas unidades do nível inferior produzirão uma saída 1. Assim, se a unidade de nível mais alto atribuir às suas entradas um peso 1 e tiver uma referência 1, produzirá uma saída 1 quando o padrão em seu campo de visão for um C e uma saída 0 se for um V.
5. A rede cairá na configuração em que a unidade de processamento do centro estiver excitada e todas as outras inibidas.

## Seção 10.5

1. 0010100110000010000100000
2. Que tal o problema de desenvolver uma estratégia para investir no mercado de ações?
3. A estrutura de um programa baseado no paradigma funcional é de funções dentro de funções. Isto é, a estrutura é homogênea em todos os níveis. Contudo, um programa baseado no paradigma orientado a objeto consiste em objetos que podem conter métodos, bem como outros objetos. Assim, a estrutura não é tão homogênea quanto a do paradigma funcional. Por isso, é mais complicado misturar componentes de programas orientados a objeto para formar novos programas.

## Seção 10.6

1. A oração está descrevendo o tipo de cavalo ou está contando o que algumas pessoas estão fazendo?
2. O processo de análise sintática produz estruturas idênticas, mas a análise semântica reconhece que a frase prepositionada da primeira oração conta onde a cerca foi construída, enquanto a frase da segunda oração conta quando a cerca foi construída.
3. Eles são irmãos.
4. Ele usa a suposição de universo fechado.
5. Os dois são iguais em muitos aspectos. Contudo, os bancos de dados tradicionais tendem a conter somente fatos como o nome do funcionário, endereço e assim por diante, enquanto bases de conhecimento tendem a incluir regras como “se estiver chovendo, confira o pluviômetro”, que podem ser usadas para orientar o processo de raciocínio.

## Seção 10.7

1. Não existe resposta correta ou errada.
2. Não existe resposta correta ou errada.
3. Não existe resposta correta ou errada.

# Capítulo 11

## Seção 11.1

1. O cálculo da dívida referente a um empréstimo, a área de um círculo ou a quilometragem de um carro.
2. Os matemáticos as chamam de funções transcendentais. Como exemplos, citam-se as funções logarítmicas e trigonométricas. Tais exemplos particulares ainda podem ser computados, mas não por meios algébricos. Por exemplo, as funções trigonométricas podem ser calculadas desenhando-se o triângulo em questão, medindo-se os seus lados e só então usando-se a operação algébrica de divisão.

## Seção 11.2

1. O resultado é o seguinte diagrama:



2.

| Estado corrente | Conteúdo da célula | Valor a ser registrado | Direção de movimento | Próximo estado |
|-----------------|--------------------|------------------------|----------------------|----------------|
| START           | *                  | *                      | esquerdo             | ESTADO 1       |
| ESTADO 1        | 0                  | 0                      | esquerdo             | ESTADO 2       |
| ESTADO 1        | 1                  | 0                      | esquerdo             | ESTADO 2       |
| ESTADO 1        | *                  | 0                      | esquerdo             | ESTADO 2       |
| ESTADO 2        | 0                  | *                      | direito              | ESTADO 3       |
| ESTADO 2        | 1                  | *                      | direito              | ESTADO 3       |
| ESTADO 2        | *                  | *                      | direito              | ESTADO 3       |
| ESTADO 3        | 0                  | 0                      | direito              | HALT           |
| ESTADO 3        | 1                  | 0                      | direito              | HALT           |

3.

| Estado corrente | Conteúdo da célula corrente | Valor a ser registrado | Direção do movimento | Próximo estado |
|-----------------|-----------------------------|------------------------|----------------------|----------------|
| START           | *                           | *                      | esquerdo             | SUBTRACT       |
| SUBTRACT        | 0                           | 1                      | esquerdo             | BORROW         |
| SUBTRACT        | 1                           | 0                      | esquerdo             | NO BORROW      |
| BORROW          | 0                           | 1                      | esquerdo             | BORROW         |
| BORROW          | 1                           | 0                      | esquerdo             | NO BORROW      |
| BORROW          | *                           | *                      | direito              | ZERO           |
| NO BORROW       | 0                           | 0                      | esquerdo             | NO BORROW      |
| NO BORROW       | 1                           | 1                      | esquerdo             | NO BORROW      |
| NO BORROW       | *                           | *                      | direito              | RETURN         |
| ZERO            | 0                           | 0                      | direito              | ZERO           |
| ZERO            | 1                           | 0                      | direito              | ZERO           |
| ZERO            | *                           | *                      | nenhum movimento     | HALT           |
| RETURN          | 0                           | 0                      | direito              | RETURN         |
| RETURN          | 1                           | 1                      | direito              | RETURN         |
| RETURN          | *                           | *                      | nenhum movimento     | HALT           |

4. O ponto aqui é a suposição de que o conceito de uma máquina de Turing capte o significado do termo “computar”. Assim, sempre que ocorrer alguma computação, os componentes e as ações de uma máquina de Turing deverão estar presentes. Por exemplo, uma pessoa que faz sua declaração de renda está executando um tipo de computação. A máquina computacional é a pessoa e a fita é representada pelo papel em que os valores são registrados.

5. A máquina descrita pela seguinte tabela termina se tiver começado com uma entrada par, porém nunca termina se começar com uma entrada ímpar:

| Estado corrente | Conteúdo da célula | Valor a ser registrado | Direção do movimento | Próximo estado |
|-----------------|--------------------|------------------------|----------------------|----------------|
| START           | *                  | *                      | esquerdo             | ESTADO 1       |
| ESTADO 1        | 0                  | 0                      | direito              | HALT           |
| ESTADO 1        | 1                  | 1                      | nenhum movimento     | ESTADO 1       |
| ESTADO 1        | *                  | *                      | nenhum movimento     | ESTADO 1       |

### Seção 11.3

1. clear AUX;  
incr AUX;  
while X not 0 do;  
  clear X;  
  clear AUX;  
end;  
while AUX not 0 do;  
  incr X;  
  clear AUX;  
end;
2. while X not 0 do;  
  decr X;  
end;
3. copy X to AUX;  
while AUX not 0 do;  
  S1  
  clear AUX;  
end;  
copy X to AUX;  
invert AUX; (Veja Questão 1)  
while AUX not 0 do;  
  S2  
  clear AUX;  
end;  
while X not 0 do;  
  clear AUX;  
  clear X;  
end;
4. Se admitirmos que X se refere à célula de memória de endereço 40 e que cada trecho de programa começa na posição 00, teremos a seguinte tabela de conversão:

|          | Endereço | Conteúdo |
|----------|----------|----------|
| clear X; | 00       | 20       |
|          | 01       | 00       |
|          | 02       | 30       |
|          | 03       | 40       |

|                                       | <i>Endereço</i> | <i>Conteúdo</i> |
|---------------------------------------|-----------------|-----------------|
| incr X;                               | 00              | 11              |
|                                       | 01              | 40              |
|                                       | 02              | 20              |
|                                       | 03              | 01              |
|                                       | 04              | 50              |
|                                       | 05              | 01              |
|                                       | 06              | 30              |
|                                       | 07              | 40              |
| decr X;                               | Endereço        | <i>Conteúdo</i> |
|                                       | 00              | 20              |
|                                       | 01              | 00              |
|                                       | 02              | 23              |
|                                       | 03              | 00              |
|                                       | 04              | 11              |
|                                       | 05              | 40              |
|                                       | 06              | 22              |
|                                       | 07              | 01              |
|                                       | 08              | B1              |
|                                       | 09              | 10              |
|                                       | 0A              | 40              |
|                                       | 0B              | 03              |
|                                       | 0C              | 50              |
|                                       | 0D              | 02              |
|                                       | 0E              | B1              |
|                                       | 0F              | 06              |
| while X<br>not 0 do;<br>. . .<br>end; | 10              | 33              |
|                                       | 11              | 40              |
|                                       | Endereço        | <i>Conteúdo</i> |
|                                       | 00              | 20              |
|                                       | 01              | 00              |
|                                       | 02              | 11              |
|                                       | 03              | 40              |
|                                       | 04              | B1              |
|                                       | 05              | WZ              |
|                                       | .               | .               |
|                                       | .               | .               |
|                                       | .               | .               |
|                                       | WX              | B0              |
|                                       | WY              | 00              |

5. Da mesma forma que em uma máquina real, os números negativos poderiam ser tratados por meio de um sistema de codificação. Por exemplo, o bit mais à direita em cada cadeia pode ser usado como bit de sinal e os demais bits utilizados para representar a amplitude do valor.
6. A função é a multiplicação por 2.

#### Seção 11.4

1. Sim. De fato, este programa pára para qualquer valor de entrada. Assim, deve parar se as suas variáveis forem iniciadas com a representação codificada do programa.

2. O programa pára somente se o valor inicial de X terminar com um 1. Uma vez que a representação ASCII do ponto-e-vírgula é 00111011, a versão codificada do programa deve terminar com um 1. Portanto, o programa termina por si mesmo.
3. O ponto aqui é a lógica ser a mesma do nosso argumento de que o problema da parada não possui uma solução algorítmica. Se o pintor pintar a sua própria casa, ele não a terá pintado, e vice-versa.

## Seção 11.5

1. Poderíamos concluir apenas que o problema possui complexidade  $\Theta(2^n)$ . Se conseguíssemos mostrar que o “melhor algoritmo” para resolver o problema pertence a  $\Theta(2^n)$ , então concluiríamos que o problema pertence a  $\Theta(2^n)$ .
2. Não. Como regra geral, um algoritmo em  $\Theta(n^2)$  terá melhor desempenho do que outro em  $\Theta(2^n)$ , mas para pequenos valores de entrada, um algoritmo exponencial freqüentemente é melhor que um polinomial. De fato, é verdade que os algoritmos exponenciais algumas vezes são preferidos quando a aplicação envolve apenas pequenas entradas.
3. O fato é que o número de subgrupos cresce exponencialmente, e a partir deste ponto, o trabalho de listar todas as possibilidades se torna uma tarefa muito trabalhosa.
4. Na classe de problemas polinomiais, há o problema de ordenação, o qual pode ser resolvido por meio de algoritmos polinomiais, como a ordenação por inserção.

Na classe dos problemas não-polinomiais está a tarefa de listar todos os subgrupos que poderiam ser formados a partir de um determinado grupo ancestral.

Qualquer problema polinomial é um problema NP. O problema do caixeiro viajante é um exemplo de problema NP para o qual não se provou ainda a existência de solução polinomial.

5. Não. Nossa uso do termo *complexidade* se refere ao tempo necessário para executar um algoritmo — não à dificuldade de entendê-lo.

## Seção 11.6

1.  $691 + 365 + 651 + 493 = 2200$
2.  $257 + 2184 + 782 = 3223$ . Este é o sistema de criptografia desenvolvido nesta seção. Resolva o problema convertendo-o em um problema baseado na lista  
$$\begin{array}{cccccccc} 4 & 6 & 12 & 25 & 51 & 105 & 210 & 421 & 850 \end{array}$$
3. Você deve encontrar um múltiplo de 5 que seja uma unidade maior que um de 23. Note que  $5 \times 14 = 70$  e  $3 \times 23 = 69$ . Portanto, o inverso multiplicativo de 5 é 14 no sistema modular de módulo 23.
4. Multiplique cada elemento da lista por 30, divida cada produto por 67 e guarde os restos. Isso produz a lista 60, 23, 46, 55, 50, que servirá como chave pública da criptografia.



# Í N D I C E

---

## A

- Ábaco, 20-21  
Abrir (operação de arquivo), 314-315  
Abstração, 24-25, 44  
Access (sistema de banco de dados da Microsoft), 343-344  
Acesso direto à memória (DMA), 97-98  
Acionador de dispositivo, 118  
Acoplamento (intermódulo), 253-254  
Acoplamento de controle, 253-254  
Acoplamento de dados, 253-254  
Acoplamentos implícitos, 254  
Acordo de não-abertura, 266-267  
Ada, 201-202, 205-207, 231-232, 455  
Adleman, Leonard, 431-432  
Administração do Seguro Social, 359-360  
Administrador de banco de dados (DBA), 338-339  
Aglomerados, 314  
Aiken, Howard, 21-22  
Alexander, Christopher, 256-257  
Algoritmo, 18-19, 150-151  
    complexidade/eficiência, 178-179, 422-425  
    descoberta, 159  
    representação, 152-153  
    verificação, 181-182  
Algoritmo de busca binária, 173-174  
    complexidade, 181-182  
Algoritmo de Euclides, 18-19  
Algoritmo de intercalação, 318-319, 425-426  
Algoritmo de ordenação no monte, 173-174  
Algoritmo de ordenação por intercalação, 425-426  
    complexidade, 426-427  
Algoritmo de ordenação por seleção, 173-174  
Algoritmo de ordenação rápida, 173-174  
Algoritmo do método da bolha 173-174  
Algoritmo não-determinístico, 429  
Algoritmo paralelo, 150-151  
Algoritmos genéticos, 390-391  
America Online, 359-360  
American National Standards Institute (ANSI), 48, 196-197  
American Standard Code for Information Interchange (ASCII) 48, 319-320, 322, 447  
Amostra de teste, 93-94  
Analizador léxico, 218-219  
Analizador sintático, 218-219  
Análise contextual, 393-394  
Análise de imagens, 372  
Análise de valores de fronteira, 263-264  
Análise do médio caso, 179-180  
Análise do melhor caso, 179-180  
Análise do pior caso, 179-180  
Análise léxica, 218-219  
Análise numérica, 63-64  
Análise semântica, 393-394  
Análise sintática, 218-219, 393  
Analógico (*versus digital*), 55-56  
AND, 34-35, 92-93  
Anulação em cascata (*cascading rollback*), 356-357  
Aparência e jeito, 266  
API, 206-207  
Applet (Java), 134-135  
Arquivo de transação, 318-319  
Argumento (de um predicado), 236-237  
Aritmética modular, 433-434  
Armações (*framework*), 258  
Armazenamento em massa, 42-43  
Armazenamento em disco, 42-43  
Arquitetura von Neumann, 98-99  
Arquivo, 46  
Arquivo de texto, 319-320  
Arquivo *hashed*, 327-328  
Arquivo indexado, 323-324  
Arquivo invertido, 324-325  
Arquivo seqüencial, 315-316  
Arquivo-mestre, 318-319  
Arquivos achados, 338  
Arquivos binários, 319-320  
Arquivos *zip*, 65-66  
Arranjo heterogêneo, 204-205  
Arranjo homogêneo, 204  
Árvore, 291-292  
Árvore balanceada, 294-295  
Árvore binária, 291-292  
Árvore cheia, 294-295  
Árvore de análise sintática, 220-221  
Árvore de busca, 376-377  
ASCII. Ver American Standard Code for Information Interchange  
Asserções, 184-185  
Association for Computing Machinery (ACM), 246  
AT&T, 185  
Atanasoff, John, 22-23  
Ativação, 176-177  
Atraso de rotação, 43-44  
Atributo, 342-343  
Auto-referência, 419-210  
Axioma, 184-185

Axônio, 382-383

## B

Babbage, Charles, 20-21, 23, 455

*Backtracking*, 285-286

Balanceamento de carga, 114-115

Balde (*hashing*), 327-328

Banco de dados, 338, 396-397

Banco de dados distribuídos, 338, 340-341

Banco de dados em universo fechado, 397-398

Banco de dados espacial, 355-356

Banco de dados orientado a objeto, 352-353

Banco de dados temporal, 345

Barreira de proteção. Ver *Firewall*.

Base de conhecimento, 398-399

Base dois. Ver Sistema binário

Berry, Clifford, 22-23

Biblioteca de Modelos Padrão, 303

*Bit*, 34-35

*Bit* de paridade, 68-69

*Bit* de sinal, 57-58, 61

*Bit* mais significativo, 40

*Bit* menos significativo, 40

*Bits* por segundo (bps), 98-99

Bloco de controle de arquivo, 314-315

Boole, George, 34-35

*Booting*, 119-120

*Bootstrap*, 119-120

Borne shell, 117-118

Bps. Ver *Bits* por segundo

*Browser*, 132-133

*Buffer*, 46

*Bus*, 80, 96-97

Busca em largura, 378-379

Busca em profundidade, 378-379

Busca seqüencial, 165

complexidade de, 179-180

Byron, Augusta Ada, 21-22, 455

*Byte*, 40

*Byte* de verificação, 69-70

## C

C, 201-205, 206-207, 208-211, 212-213, 215-216, 217-218,

326-327, 331-332, 455-456

C shell, 117-118

C#, 201-202, 203-204, 206-207, 208-211, 215-216, 218-219, 226-230, 457-460

C++, 201-207, 208-211, 215-216, 218-219, 226-228, 229-230, 303-304, 456-457

Cabeçalho do procedimento, 211

Camada de aplicação (Internet), 136

Camada de ligação (Internet), 136

Camada de rede (Internet), 136

Camada de transporte (Internet), 136

Camada segura de encaixes, 141-142

Campo, 314

Campo da mantissa, 61

Campo de expoente, 61

Campo-chave, 317-318, 323-324

Capacitor, 36-37

Carregador, 224-225

*Carrier Sense, Multiple Access with Collision Detection (CSMA/CD)*, 133-135, 138-139

Casca (*shell*), 116-117

CASE. Ver Engenharia de software assistida por computador

Caso básico, 176-177

Caso degenerativo, 176-177

CD. Ver Disco ótico

CD-DA. Ver *Compact disk-digital audio*

Célula (memória), 40

CERT. Ver Computer emergency response team

CGL. Ver Common gateway interface

Chave por deslocamento de freqüência, 99-100

Chaveamento de processos, 120-121

*Chip*, 37-38

Church, Alonzo, 414

Ciclo da máquina, 86

Ciclo de vida do software, 247-248

Ciência da computação, 17, 23-24

Cilindro, 42-43

CISC. Ver *Complex instruction set computer*

Classe, 226-227, 301-302

Cláusula, 234-235

Cliente, 122-123

Clowes, M. B., 494-495

COBOL, 195-196, 331-332

Codificação adaptável de dicionário, 66

Codificação de Lempel-Ziv, 66

Codificação de tamanho de seqüência, 65

Codificação Manchester, 133-134

Codificação relativa, 65

Código de bytes, 219-220

Código de correção de erros, 70-71

Código de Huffman, 66, 494-495

Código de operação, 82-84

Código dependente da freqüência, 65

Códigos de comprimento variável, 65

Códigos de redundância cíclica, 70

Coerção, 223-224

Coesão (intramódulo), 254

Coesão funcional, 254-255

Coesão lógica, 254

Coleta de lixo, 294-295

Colisão (*hashing*), 328-329

Colossus, 22-23

Comentários, 202-203, 209-211

Comissão Federal de Comunicações (FCC), 142-143

Common gateway interface (CGI), 135

Communication Assistance for Law Enforcement Act (CALEA), 142-143

*Compact disk-digital audio* (CD-DA), 44, 70-71

Compilador, 195-196

Complemento (de um *bit*), 57-58  
*Complex instruction set computer* (CISC), 82  
 Complexidade de espaço, 424-425  
 Complexidade de tempo, 424-425  
 Complexidade/eficiência, 178-179, 422-424  
   de busca binária, 179-180, 181-182  
   de busca seqüencial, 179-180  
   ordenação por inserção, 180-181  
   ordenação por intercalação, 426-427  
 Compressão de dados, 64-65  
 Computador Apple, Inc., 23, 51-52, 82-83  
 Computador pessoal, 23  
 Computer Emergency Response Team (CERT), 143  
 Comunicação entre processos, 122-123  
 Comunicação paralela, 98-99  
 Comunicação serial, 99-100  
 Concatenação, 207-208  
 Conceito de programa armazenado, 81  
 Condição terminal, 166-167  
 Conhecimento declarativo, 303-304  
 Conhecimento procedimental, 303-304  
 Constante, 205-206  
 Construtor, 228-229  
 Contador de instruções, 86, 276-277  
 Controlador, 96-97  
 Controle de repetição de estruturas iterativas, 165  
   recursão, 176-177  
*Cookies*, 147-148  
 CORBA (Common Object Request Broker Architecture), 123, 226-227  
*Core wars*, 104-105  
 Corpo (de uma iteração), 165-166  
 Corrente (*stream*), 37-38  
 CRC. Ver Fichas CRC  
 Criptografia de chave pública, 141-142, 430-431  
 CSMA/CD. Ver *Carrier Sense, Multiple Access with Collision Detection*

**D**

Dados globais, 254  
 Danificação do cabeçote (*head crash*), 43-44  
 Dartmouth College, 368  
 Darwin, Charles, 400-401  
*Deadlock*, 124-125  
 Decomposição sem perdas, 345-347  
 Dedução lógica, 233-234  
 Defense Advanced Research Projects Agency (DARPA), 129  
 Dendrites, 382-383  
 Departamento de Defesa norte-americano, 455  
 Depuração (*debugging*), 194-195  
 Descritor de arquivo, 118, 314  
 Desempilhamento (operação em pilha), 284-285  
 Desenvolvimento de código aberto, 258-259  
 Deslocamento lógico (*logical shift*), 94  
 Despachante, 118, 120-121  
 Diagrama de colaboração, 253-254

Diagrama de entidades e relacionamentos, 259-260  
 Diagrama de estrutura, 252  
 Diagrama de fluxo de dados, 259-260  
 Diagrama de sintaxe, 219-220  
 Diagramas de classe, 252  
 Dicionário de dados, 262  
 Digital (*versus analógico*), 55-56  
 Digital subscriber line (DSL), 99-100  
 Digital versatile disk (DVD), 44-46  
 Diretório, 117-118  
 Disco ótico, 44, 52-53, 70-71  
 Disco rígido, 43-44  
 Discos magnéticos, 42-43  
 Discos Zip, 43-44  
 Disquete (*floppy disk*), 43-44  
 Distância de Hamming, 70-71  
 DMA. Ver Acesso direto à memória  
 Documentação, 264-265  
   de comentários, 209-211  
 Documentação de sistemas, 264-265  
 Documentação do usuário, 264-265  
 Domínio, 129  
 Domínio de alto nível (TDL), 130-131  
 DVD. Ver *Digital versatile disk*

**E**

Eckert, J. Presper, 22-23, 81  
 Edison, Thomas, 81  
 Editor, 225-226, 319-320  
 Efeito colateral, 254  
 Efetivo, 150-151  
 Electronic Communication Privacy Act (ECPA), 141-142  
 ELIZA, 368-369  
*E-mail*, 131-132, 316-317  
 Empilhamento (operação em pilha), 284-285  
 Encapsulamento, 230  
 Endereçamento direto, 305  
 Endereçamento imediato, 305  
 Endereçamento indireto, 305  
 Endereçamento relativo, 470-471  
 Endereço (da memória da célula), 40  
 Endereço do hospedeiro, 130-131  
 Endereço IP, 129-130  
 Endereço polinomial, 279-280  
 Engenharia de inferência, 398-399  
 Engenharia de *software*, 18-19, 246  
 Engenharia de *software* assistida por computador (CASE), 251  
 ENIAC, 22-23, 54-55  
 Enlace mortal. Ver *Deadlock*  
 Entrada efetiva (de um processador), 384-385  
 Entrada ou saída formatada, 217-218  
 Entrada/saída, 82-83  
 Entrada/Saída mapeada em memória, 97-98  
 Ènupla (em uma relação), 342-343  
 EOF. Ver Marca de fim de arquivo  
 Erro de arredondamento, 63

Erro de *overflow*, 50-51, 58-59  
 Erros de truncamento, 50-51, 63  
 Escalação, 114-115  
 Escalador, 118, 120-121  
 Escalar colinas, 495-496  
 Especificações de sistemas, 249-250  
 Esperando (processo), 120-121  
 Esquema, 338-339  
 Estado  
   da máquina de Turing, 411-412  
   do processo, 119-120, 150-151  
   do sistema de produção, 372-373  
 Estrutura de controle Case, 208-209  
 Estrutura de controle Repeat, 167-168  
 Estrutura de controle while, 156-157, 165-166, 184-185, 208-209  
 Estrutura de dados, 204  
   dinâmica *versus* estática, 276  
 Estrutura iterativa, 143-144, 164-165, 463  
 Estrutura laço, 165-166  
 Estrutura recursiva, 172-173, 463  
 Ethernet, 133-134  
 Euclid, 18-19  
 Exceção, 455  
 Exclusão mútua, 124-125, 232-233  
 Extensible Markup Language (XML), 320-323  
 Extração (técnica *hash*), 327-328  
 Extração de informação, 393-394  
 Extremidade de alta ordem, 40  
 Extremidade de baixa ordem, 40

**F**

FAT. Ver Tabela de alocação de arquivos  
 Fátia de tempo, 113-114, 120-121  
 Fator de carga (arquivo *hash*), 329-330  
 Fatorial, 188-189  
 FCC. Ver Comissão Federal de Comunicações  
 Fechar (operação de arquivo), 314-315  
 Ferramentas abstratas, 25-26, 36-37, 252, 276, 283-284, 286-287, 295-296, 340-341  
 Ferramentas CASE, 251  
 Fichas CRC (*class-responsability-collaboration*), 262  
 FIFO. Ver *First-in, first out*  
 Fila, 112, 287-288  
 Fila circular, 289-290  
 Fila de *jobs*, 112  
 Filhos (em uma árvore), 291-292  
 Filósofos jantando, 145-146  
 Final da fila, 287-288  
 Firewall, 143  
*First-in, first out* (FIFO), 112, 287-288  
 Fita magnética, 44-46  
*Flip-flop*, 35-37  
 Flowers, Tommy, 22-23  
 Fluxo de dados, 152-153  
 Fontes em escala, 51-52

Forma normalizada, 62-63, 468-469  
 Formatação (de um disco), 42-43  
 Fortemente tipificado, 223-224  
 FORTRAN, 195-196, 201-202, 204-205, 207-208, 457-460  
 FTP. Ver Protocolo de transferência de arquivos  
 Função  
   *abstract*, 410  
   computação da, 410  
   unidade de programa, 215-216  
 Função *hash*, 327-328  
 Função membro, 226-227  
 Função sucessor, 414  
 Funções transcendentais, 497

**G**

Gandhi, Mahatma, 400-401  
*Gateway*. Ver Portão.  
 GB. Ver Gigabyte  
 Gbps. Ver Giga-bps  
 Gêmeos (em uma árvore), 291-292  
 General Motors, 107-108  
 Geometria euclidiana, 184-185  
 Gerações (de linguagens de programação), 195  
 Gerador de código, 218-219  
 Gerente de arquivos, 117-118  
 Gerente de memória, 118  
 Gerentes de janelas, 117-118  
 Gibi, 41-42  
 GIF, 51-52, 67  
 Giga-bps, 98-99  
*Gigabyte*, 41-42  
 Gödel, Kurt 20-21, 412-413  
 Gráfico, 373-374  
 Gráfico de estados, 373-374  
 Grafo dirigido, 373-374  
 GUI. Ver Interface gráfica com o usuário

**H**

Hamming, R. W., 70-71  
*Handshaking*, 118  
*Hardware*, 18-19  
*Hashing*, 327-328  
 Herança, 229-230  
 Hertz (Hz), 93-94  
 Heurística, 378-379  
 Hipermídia, 131-132  
 Hipertexto, 131-132  
 Hollerith, Herman, 21-22  
*Home page*, 132-133  
 Hopper, Grace, 195-196, 393  
 Hospedeiro, 130-131  
 HTML. Ver Hypertext Markup Language  
 Huffman, David A., 65-66  
 Hypertext Markup Language (HTML), 132-133, 320-322

**I**

I/O. Ver Entrada/saída  
 IA forte, 401  
 IA fraca, 401  
 IBM, 21-22, 23, 82-83, 133-134  
 ICANN. Ver Internet Corporation for Assigned Names and Numbers  
 Identificador de rede, 129-130  
 Identificadores, 194-195  
 IEEE, 132, 246-247  
 IEEE Computer Society, 246-247  
 II Guerra Mundial, 22-23  
 IMAP. Ver Internet Mail Access Protocol  
 Inanição, 145-146  
 Independência de dados, 341-342  
 Independência de máquina, 196-197  
 Índice parcial, 325-326  
 Índices, 204-205  
 Início da fila, 287-288  
 Instância de classe, 228-229  
 Instância de um tipo de dados, 301-302  
 Instrução Close, 314-315  
 Instrução For, 208-210  
 Instrução Goto, 207-208  
 Instrução If, 155-156, 208-209, 220-221  
 Instrução Open, 314-315  
 Instruções de atribuição, 155-156, 206-207  
 Instruções de controle, 207-208  
 Instruções declarativas, 202-203  
 Instruções de entrada e saída, 216-217  
 Instruções de máquina, 81-82
 

- ADD, 85, 94-95
- AND, 82-83, 92-93
- BRANCH, 82-84
- HALT, 89-90
- I/O, 82-83
- JUMP, 82-84, 87, 100, 207-208
- LOAD, 82-83
- OR, 82-83, 92-94
- ROTATE, 82-83, 94
- SHIFT, 82-83, 94
- STORE, 82-83
- Test-and-set*, 124-125
- XOR (OR exclusivo), 82-83, 92-94

Instruções imperativas, 202-203  
 Intel, 82-83  
 Inteligência artificial, 20-21, 368
 

- método orientado à simulação, 368-369
- método orientado ao desempenho, 368

Inteligência baseada em comportamento, 373-374  
 Interface gráfica com o usuário (GUI), 116-117, 206-207, 211, 251  
 International Organization for Standardization (ISO), 48-49, 65-66, 67-68, 138-139, 196-197  
 Internet, 113-114, 129  
 Internet Corporation for Assigned Names and Numbers (ICANN), 129-130

Internet Mail Access Protocol (IMAP), 140

Intérprete, 195-196  
 Interrupção, 120-121  
 Invariante da iteração, 166-167  
 Invariante do laço, 184-185  
 Inversos multiplicativos, 434-435  
 Iomega Corporation, 43-44  
 Iowa State College (Universidade), 22-23  
 IP. Ver Protocolo da Internet  
 Irmãos Wright, 81  
 ISO. Ver International Organization for Standardization  
 ISP. Ver Internet service provider  
 Iverson, Kenneth E., 206-207

**J**

Jacquard, Joseph, 21-22  
 Java, 201-207, 208-211, 215-216, 218-220, 226-228, 229-230, 231-233, 330-332, 457-460  
 Java *applets*, 134-135  
 Java Development Kit (JDK), 256  
 Java Servlets, 134-135  
 Javascript, 134-135  
 JCL (*job control language*), 112  
 Jobs, Steve, 23  
 JOIN (operação de banco de dados), 345-347  
 JPEG, 51-52, 67

**K**

KB. Ver Quilobyte  
 Kbps. Ver Quilo-bps  
 Kernel 101  
 Kibibyte, 41-42  
 Korn shell, 117-118

**L**

Laboratórios Bell, 21-22, 455-457  
 Laço pós-teste, 167-168  
 Laço pré-teste, 167-168  
 LAN (*Local area network*). Ver Rede local  
 Largura de banda, 98-99  
*Last-in, first-out* (LIFO), 284-285  
 Lei de direitos autorais, 265-266  
 Lei de patente, 266-267  
 Lei de segredos industriais, 266-267  
 Lempel, Abraham, 65-66  
 Leonardo da Vinci, 81  
 Liebniz, Gottfried Wilhelm, 20-21  
 LIFO. Ver *Last-in, first out*  
 Ligador, 224-225  
 Limitado por entrada/saída, 143-144  
 Linguagem assembler, 195  
 Linguagem Bare Bones, 414-415, 463

universalidade, 418-419  
 Linguagem de consulta estruturada, 349-350  
 Linguagem de formato fixo, 219-220  
 Linguagem de máquina, 81-82, 304-305  
 Linguagem de primeira geração, 195  
 Linguagem de programação, 18-19, 153-154  
 Linguagem de programação universal, 414-415  
 Linguagem de segunda geração, 195  
 Linguagem de terceira geração, 195-196  
 Linguagem em formato livre, 219-220  
 Linguagem hospedeira, 342-343  
 Linguagens de marcação, 320-322  
 Lingüística, 368-369  
 Linha (Java), 231-232  
 Linux, 117-118  
 LISP, 199-200  
 Lista, 280-281  
     contígua, 280-281  
     ligada, 281-282  
 Literal, 205-206  
 Livros de culinária, 257-258  
 Lógica de predicados de primeira ordem, 208-209  
 Lotus 1-2-3, 266-267  
 Lotus Development Corporation, 266-267  
 LZ77, 66

**M**

Mapa de bits (*bit map*), 50-51, 92-93  
 Máquina analítica, 21-22  
 Máquina das diferenças, 21-22, 23  
 Máquina de Turing, 411-412  
 Marca (em linguagem de programação), 320-322  
 Marca de fim de arquivo (EOF), 317-318  
 Mariner 18 — sonda espacial, 246-247  
 Mark I, 21-22, 181-182  
 Máscara, 92-93  
 Mascaramento, 92-93  
 Matar (um processo), 125-126  
 Matriz, 204, 277-278  
     heterogênea, 204-205  
     homogênea, 204-205, 277-278  
 Mauchly, John, 22-23  
 Máximo divisor comum, 18-19  
 MB. Ver Megabyte  
 Mbps. Ver Mega-bps  
 McCarthy, John, 368  
 Mebi, 41-42  
 Mega-bps, 98-99  
 Megabyte, 41-42  
 Memória *cache*, 81-82  
 Memória de acesso aleatório (RAM), 40-41  
 Memória dinâmica, 37-38, 40-41  
 Memória principal, 40  
 Memória somente para leitura (ROM), 119-120  
 Memória virtual, 118  
 Método, 226-227

Método meio-quadrado (técnica *hash*), 328  
 Metodologia ascendente (*bottom-up*), 162-163, 256  
 Metodologia descendente (*top-down*), 162-163, 256  
 Métrica, 246  
 Microsoft Corporation, 23, 51-52, 115-116, 201-202, 457-460  
 Microsegundo, 73-74  
 MIDI. Ver Musical Instrument Digital Interface  
 Milissegundo, 44  
 Miller, George A., 152-153  
 MIMD, 101  
 MIME. Ver Multipurpose Internet Mail Extensions  
 Mineração de dados, 358-359  
 Modelo Cliente/Servidor, 122-123  
 Modelo da cachoeira, 250-251  
 Modelo de banco de dados, 341-342  
 Modelo de referência OSI, 138-139  
 Modelo do quadro negro, 398-399  
 Modelo incremental, 250-251  
 Modelo relacional de banco de dados, 341-342  
 Modem, 99-100  
 Modularidade, 252  
 Módulo, 156-157, 433-434  
 Módulo de carga, 224-225  
 Módulo relocável, 224-225  
 Mondrian, Piet, 174-175  
 Monitor, 232-233  
 Montador, 195  
 Mosaic Software, 266-267  
 Motion Picture Experts Group (MPEG), 65-66, 67-68  
 Motorola, 82-83  
 Mouse, 116-117  
 MP3, 65-66  
 MPEG. Ver Motion Picture Experts Group  
 MS-DOS, 117-118  
 Multipurpose Internet Mail Extensions (MIME), 316-317  
 Multitarefas, 25-26  
 Musical Instruments Digital Interface (MIDI), 52-53

**N**

Nanossegundo, 100  
 Não-terminal, 220-221  
 .NET framework, 230, 457-460  
 Netscape Communications, Inc., 134-135  
 Neurônios, 19-20, 382-383  
 Newton, Isaac, 246-247  
 Nô, 291-292, 373-374  
 Nô ancestral, 291-292  
 Nô raiz, 291-292  
 Nô terminal, 291-292  
 Nô-folha, 291-292  
 Nome do domínio, 130-131  
 Nome do servidor, 130-131  
 NOT, 35-37  
 Notação binária, 43-44 (Ver também Sistema binário)  
 Notação Camelô, 153-154  
 Notação de complemento de dois, 56-57

Notação de excesso, 59-60  
 Notação de vírgula flutuante, 61  
     forma normalizada, 62-63  
 Notação decimal pontuada, 53-54, 130-131  
 Notação hexadecimal, 37-38  
 Notação O, 425-426  
 Notação Pascal, 153-154  
 Notação teta, 181-182, 425-426  
 Novell, Inc., 127  
 Núcleos, 36-37, 104-105  
 Número primo, 328-329  
 Nuvem (Internet), 129-130

**O**

Object Management Group, 123  
 Objeto, 199-200  
 Obsolescência planejada, 107-108  
*Off-line*, 42-43  
*On-line*, 42-43  
 OOP. Ver Programação orientada a objeto  
 Open System Interconnect (OSI), 138-139  
 Operação de gravação, 40-41  
 Operação de leitura, 40-41  
 Operações booleanas, 34-35  
 Operando, 82-84  
 OR, 34-35, 92-93  
 Ordenação por inserção, 170-171  
     complexidade, 180-181  
 Orwell, George (Eric Blair), 28-29  
 Otimização de código, 223-224  
 Ou exclusivo (XOR), 34-35, 92-93

**P**

P. Ver Problemas polinomiais  
 Pacote, 137-138  
 Pacotes de ajuda, 264-265  
 Padrões de projeto, 256-257  
 Página (memória), 118  
 Página da Web, 131-132  
 Palavra de *status* (*status word*), 98-99  
 Palavras reservadas, 219-220  
 Palavras-chave, 219-220  
 Paradigma declarativo, 197-198  
 Paradigma funcional, 198-199, 392  
 Paradigma imperativo, 197-198  
 Paradigma procedural, 197-198  
 Paradigmas de programação, 18-19, 197-198  
 Parâmetro, 212-213  
     passado por referência, 214-215  
     passado por valor, 214-215  
 Parâmetro formal, 212-213  
 Parâmetro real, 212-213  
 Pareto, Vilfredo, 262-263  
 Paridade ímpar, 69-70

Paridade par, 69-70  
 Pascal, 201-202, 204, 206-207, 209-210, 217-218, 460-462  
 Pascal, Blaise, 20-21  
 Pastas, 117-118  
 PC. Ver Computador pessoal  
 Pentium, 82-83, 89-90, 93-94, 181-182  
 Período de incubação, 160-161  
 Peso (em um processador), 384-385  
 PGP. Ver Pretty Good Privacy  
 Phillips, E. W., 54-55  
 Pilha, 284-285  
*Pipelining*, 100  
*Pixel*, 50-51  
 Placa mãe, 96-97  
 Poincare, H., 160-161  
 Polimorfismo, 230  
 Polya, G., 159, 250-251  
 Ponte (*bridge*), 127  
 Ponteiro, 276-277, 289-290, 304-305  
 Ponteiro da pilha, 286-287  
 Ponteiro da raiz, 293-294  
 Ponteiro do final da fila, 288-289  
 Ponteiro NIL, 281-282  
 Ponteiro para o filho direito, 292-293  
 Ponteiro para o filho esquerdo, 292-293  
 Ponteiro para o início, 281-282, 288-289  
 Ponto de comprometimento, 355-356  
 POP3. Ver Post Office Protocol-version 18-19  
 Porta (E/S), 97-98  
 Porta (Internet), 138-139  
 Porta lógica, 35-37  
 Portão (*gateway*), 129-130  
 Post Office Protocol-version 18-19, 139-140  
 Post, Emil, 412-413  
*Postscript*, 51-52  
 PowerPC, 82-83, 93-94  
 Precedência de operador, 207-208  
 Precondições (prova de correção), 184-185  
 Predicado, 235-236  
 Pretty Good Privacy (PGP), 431-432  
 Primitiva, 153-154  
 Princípio de Pareto, 262-263  
 Privacy Act of 1974, 358-359  
 Problema da atualização perdida, 356-357  
 Problema da mochila, 431-432  
 Problema da parada, 419-210  
 Problema da soma incorreta, 356-357  
 Problema do caixeiro-viajante, 428-429  
 Problema insolúvel, 422-423  
 Problemas intratáveis, 428-429  
 Problemas polinomiais, 427-428  
 Problemas polinomiais não determinísticos (NP), 429-430  
 Procedimento, 156-157, 210-211  
 Processamento concorrente, 231-232  
 Processamento da imagem, 372  
 Processamento de linguagens, 393  
 Processamento de linguagens naturais, 368-369  
 Processamento em lote, 112

Processamento em tempo real, 112-113  
 Processamento interativo, 112-113  
 Processo, 119-120, 151-152  
 Processo paralelo, 101, 231-232  
 Profundidade (de uma árvore), 291-292  
 Programa, 18-19, 151-152  
 Programa DOCTOR (ELIZA), 368-369  
 Programa termina por si mesmo, 420-421  
 Programação estruturada, 209-210  
 Programação extrema, 251  
 Programação orientada a objeto, 199-200, 206  
 Programa-fonte, 218-219  
 Programa-objeto, 218-219  
 PROJECT (operação em banco de dados), 345-347  
 Projeto assistido por computador (CAD), 51-52  
 Projeto orientado a objeto, 199-200  
 Prolog, 235-236  
 Pronto (processo), 121  
 Proposição inconsistente, 234-235  
 Protocolo, 133-134  
 Protocolo *commit/rollback*, 355-356  
 Protocolo da Internet (IP), 138-139  
 Protocolo de transferência de arquivos (FTP), 136-137  
 Protocolo de travamento, 357  
 Protocolo sem conexão, 139-140  
 Protocolo TCP/IP, 138-139  
 Protocolo *token ring*, 133-134, 138-139  
 Prototipação, 250-251  
 Prototipação descartável, 251  
 Prova de correção, 184-185  
 Provedor de serviços da Internet, 129-130  
 Pseudocódigo, 152-153, 154-155

**Q**

Quantum, 120-121  
 Quebra-cabeças de oito peças, 370-371  
 Quilo-bps, 98-99  
 Quilobyte, 41-42

**R**

RAM. Ver Memória de acesso aleatório  
 Ravel, Maurice, 131-132  
 Recuperação de informação, 393-394  
 Recursão, 176-177, 393-394, 463  
 Rede, 113-114  
 Rede aberta, 127  
 Rede com topologia de via, 127  
 Rede com topologia em estrela, 127  
 Rede de longa distância (WAN — *wide area network*), 127  
 Rede em topologia de anel, 127  
 Rede fechada, 127  
 Rede local (LAN — *local area network*), 127  
 Rede neural artificial, 101, 381-382  
 Rede proprietária, 127

Rede semântica, 394-395  
*Reduced instruction set computer* (RISC), 82  
 Referência, 289-290  
 Refinamento passo a passo, 162-163  
 Região crítica, 124-125  
 Registrador, 80  
 Registrador de instruções, 86  
 Registrador de propósito específico, 80  
 Registrador de propósito geral, 80  
 Registrador físico, 46  
 Registro lógico, 46  
 Relação (banco de dados), 342-343  
 Relacionamentos de muitos para muitos, 261  
 Relacionamentos um para muitos, 261  
 Relacionamentos um para um, 261  
 Relógio, 38-39, 93-94  
 Remote method invocation (RMI), 226-227  
 Representação em sistema bi-quínário, 54-55  
 Requisitos (de software), 248-249  
 Resolução, 233-234  
 Resolução de problemas, 159  
 Resolvente, 233-234  
 Retentor. Ver Buffer  
 RISC. Ver *Reduced instruction set computer*  
 Ritchie, Dennis, 455-456  
 Rivest, Ron, 431-432  
 RMI. Ver Remote method invocation  
 Robótica, 394-395  
*Roll back*, 356-357  
 ROM. Ver Memória somente para leitura  
 Roteador, 129  
 Rotina de tratamento de interrupção, 121-122  
 RSA, 266-267, 431-432

**S**

Salto condicional, 82-84  
 Salto incondicional, 82-84  
 Saltos (*hop count*), 139-140  
 Segurança em redes, 140-141  
 SELECT (operação de banco de dados), 345-347  
 Selective Service, 358-359  
 Semáforo, 124-125  
 Semântica, 153-154  
 Sentinela, 317-318  
 Sequência de colunas, 278-279  
 Sequência de Fibonacci, 187-188  
 Sequência de linhas, 278-279  
 Servidor, 122-123  
 Servidor de correio, 131-132  
 Setores, 42-43  
 SGML. Ver Standard Generalized Markup Language  
 Shamir, Adi, 431-432  
*Shift* aritmético, 94  
 Significativamente similar, 266  
 SIMD, 101  
 Simple Mail Transfer Protocol (SMTP), 136-137

- Sinapse, 382-383  
 Sintaxe, 153-154  
 SISD, 101  
 Sistema Adobe, 51-52  
 Sistema binário, 48-49, 53-54  
 Sistema de gerência de banco de dados (DBMS), 340-341  
 Sistema de produção, 373  
     estado inicial, 373  
     estado-meta, 373  
     grafo de estados, 373-374  
     sistema de controle, 374  
 Sistema distribuído, 127  
 Sistema operacional, 19-20  
 Sistemas especialistas, 398-399  
 Sítio da Web, 131-132  
 Sítio de busca, 134-135  
 Sloan, Alfred, 107-108  
 SMTP. Ver Simple Mail Transfer Protocol  
 Sobrecarga (*overloading*), 207-208  
*Software*, 18-19  
*Software* de aplicação, 115-116  
*Software* de sistemas, 115-116  
*Software* dirigidos por eventos, 211  
*Software* multiplataforma, 195  
*Software* utilitário, 115-116  
 Soma ponderada, 382-383  
 Somador com ondulação, 447-448  
 Somador com previsão de transporte, 447-448  
*Spooling*, 125-126  
 SQL. Ver Linguagem de consulta estruturada  
 Standard Generalized Markup Language (SGML), 320-322  
 Standard Template Library (STL). Ver Biblioteca de Modelos  
     Padrão  
 Stibitz, George, 21-22  
 Stroustrup, Bjarne, 456-457  
*Stub*, 250-251  
 Subárvore, 291-292  
 Subesquema, 338-339  
 Subprograma, 156-157  
 Sub-rotina, 156-157  
 Sun Microsystems, 201-202, 457-460  
 System/360 (IBM), 266
- T**
- Tabela de alocação de arquivos (FAT), 314  
 Tabela de processo, 120-121  
 Tabela de símbolos, 222-223  
 Tabela *hash*, 327-328  
 Tarefas, 231-232  
 Taxa de transferência, 43-44  
 TCP. Ver *Transmission Control Protocol*  
 Tear de Jacquard, 21-22, 394-395  
 Telnet, 136-137  
 Tempo compartilhado (*time-sharing*), 113-114, 120-121  
 Tempo de acesso, 43-44  
 Tempo de busca, 43-44
- Tempo de latência, 43-44  
 Tempo exponencial, 428-429  
 Teorema de incompleteza de Gödel 20-21, 23-24  
 Teoria da codificação algébrica, 70-71  
 Teoria das funções recursivas, 410-411  
 Teoria de conjuntos, 184-185  
 Teoria dos grafos, 262-263  
 Terminal (em um diagrama de sintaxe), 220-221  
 Termo de negação de responsabilidade, 266-267  
 Tese de Church-Turing, 414, 418-419, 463, 464  
 Tese Rogeriana, 369-370  
 Teste beta, 258-259, 263-264  
 Teste de QI, 401  
 Teste de Turing, 368-369  
 Teste do caminho base, 263  
 Teste na caixa de vidro, 263  
 Teste na caixa preta, 263  
 Testes, 263  
*Therac-25*, 265-266  
 Thoreau, Henry David, 107-108  
 Tipo de dados, 202-203  
     áudio, 204  
     Booleano, 203-204  
     caractere, 203-204  
     *float*, 202-203  
     inteiro, 202-203  
     real, 202-203  
     vídeo, 204  
 Tipo de dados definidos pelo usuário, 299-300  
 TLD. Ver Domínio de alto nível  
*Token* (em um tradutor), 218-219  
*Token* (em uma rede), 133-134  
 Topo da pilha, 284-285  
 Topologia de rede, 127  
 Topologia de rede irregular, 127  
 Torres de Hanoi, 188-189  
 Torvalds, Linus, 117-118  
 Trabalho, 112  
 Tradução, 218-219  
 Tradutor, 195-196  
 Trajetória, 117-118  
*Transmission Control Protocol* (TCP), 139  
*Trapdoor*, 140-141  
 Travãa compartilhada, 357  
 Travamento exclusivo, 357  
 Trilha, 42-43  
 TrueType, 51-52  
 Turing, Alan M., 368, 368-369, 411-412, 414  
 Turing-computável, 414
- U**
- UDP. Ver *User Datagram Protocol*  
 UML. Ver *Unified Modeling Language*  
*Unicod*, 48-49, 319-320  
 Unidade aritmética/lógica, 80  
 Unidade central de processamento (UCP), 80

Unidade de controle, 80  
Unidade de fita *streaming*, 44-46  
Unidade de processamento, 382-383  
Unificação, 235-236  
Unified Modeling Language (UML), 252-253  
*Uniform resource locator* (URL), 132-133  
Universidade Carnegie-Mellon, 143  
Universidade de Harvard, 21-22  
Universidade de Helsinqui, 117-118  
Universidade de Pennsylvania, 22-23  
UNIX, 117-118  
URL. Ver *Uniform resource locator*  
User Datagram Protocol (UDP), 138-139  
Utilitarismo, 28-29

**V**

Variáveis globais, 211  
Variáveis locais, 211  
Variável, 202-203  
Variável de instância, 226-227  
Vazamento de memória, 294-295  
Vazão de processamento, 100  
Verificação do *software*, 181-182  
Verificação de soma (*check sum*), 70  
Verme, 143  
Versão beta, 263-264  
Vetoriais (*imagens*), 50-51  
Vírgula (*radix point*), 54-55  
Vírus, 142-143

Visual Basic, 209-211  
von Helmholtz, H., 160-161  
von Neumann, John, 81, 373-374  
gargalo, 98-99

**W**

Waltz, D., 494-495  
WAN (*Wide area network*). Ver Rede de longa distância  
Web semântica, 322-323  
Weizenbaum, Joseph, 401-402  
Windows, 115-116, 117-118, 314  
Wirth, Niklaus, 460-462  
World Wide Web Consortium (W3C), 317-318  
World Wide Web, 131-132  
Wound-wait protocol, 357  
Wozniak, Stephen, 23

**X**

XHTML, 320-322  
XML. Ver Extensible Markup Language  
XOR. Ver Ou exclusivo

**Z**

Zimmermann, Philip, 431-432  
Ziv, Jacob, 65-66

