ROBERT W. SEBESTA

CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO

11ª Edição

O autor

Robert Sebesta é professor associado emérito no Departamento de Ciência da Computação da Universidade do Colorado – Colorado Springs. É bacharel em matemática aplicada pela Universidade do Colorado em Boulder e mestre e doutor em ciência da computação pela Universidade Estadual da Pensilvânia. Ele ensina ciência da computação há mais de 40 anos. Seus interesses profissionais são o projeto e a avaliação de linguagens de programação e a programação para a Web.

S443c Sebesta, Robert W.
Conceitos de linguagens de programação [recurso eletrônico] / Robert W. Sebesta; tradução: João Eduardo Nóbrega Tortello. – 11. ed. – Porto Alegre : Bookman, 2018.

Editado também como livro impresso em 2018.
ISBN 978-85-8260-469-4

1. Ciência da computação. 2. Linguagens de programação de computador. I. Título.

CDU 004.43

Catalogação na publicação: Karin Lorien Menoncin – CRB 10/2147

ROBERT W. SEBESTA

Universidade do Colorado em Colorado Springs

CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO

11ª Edição

Tradução:
João Eduardo Nóbrega Tortello

Versão impressa
desta obra: 2018

2018

Obra originalmente publicada sob o título *Concepts of Programming Languages*, 11th Edition
ISBN 9780133943023

Gerente editorial: *Arysinha Jacques Affonso*

Colaboraram nesta edição:

Editora: *Mariana Belloli Cunha*

Capa: *Marcio Monticelli*

Imagem da capa: ©shutterstock.com / Ovidiu Lazar, one of the most spectacular locations that we have seen, the Swiss Alps

Preparação de original: *Gabriela Dal Bosco Sitta*

Tradução da 9ª ed.: *Eduardo Kessler Piveta*

Projeto gráfico e editoração: *Techbooks*

Agradecimentos

As sugestões de ótimos revisores contribuíram muito para o formato atual e para o conteúdo deste livro. Em ordem alfabética de sobrenomes, são eles:

Aaron Rababaah	*Universidade de Maryland em Eastern Shore*
Amar Raheja	*Universidade Politécnica do Estado da Califórnia – Pomona*
Amer Diwan	*Universidade do Colorado*
Bob Neufeld	*Universidade Estadual de Wichita*
Bruce R. Maxim	*Universidade de Michigan – Dearborn*
Charles Nicholas	*Universidade de Maryland – Condado de Baltimore*
Cristian Videira Lopes	*Universidade da Califórnia – Irvine*
Curtis Meadow	*Universidade do Maine*
David E. Goldschmidt	
Donald Kraft	*Universidade do Estado da Louisiana*
Duane J. Jarc	*Universidade de Maryland, University College*
Euripides Montagne	*Universidade da Flórida Central*
Frank J. Mitropoulos	*Universidade de Nova Southeastern*
Gloria Melara	*Universidade do Estado da Califórnia – Northridge*
Hossein Saiedian	*Universidade do Kansas*
I-ping Chu	*Universidade DePaul*
Ian Barland	*Universidade de Radford*
K. N. King	*Universidade do Estado da Georgia*
Karina Assiter	*Instituto de Tecnologia de Wentworth*
Mark Llewellyn	*Universidade da Flórida Central*
Matthew Michael Burke	
Michael Prentice	*SUNY Buffalo*
Nancy Tinkham	*Universidade Rowan*
Neelam Soundarajan	*Universidade Estadual de Ohio*
Nigel Gwee	*Universidade do Sul – Baton Rouge*
Pamela Cutter	*Faculdade Kalamazoo*
Paul M. Jackowitz	*Universidade de Scranton*
Paul Tymann	*Instituto de Tecnologia de Rochester*
Richard M. Osborne	*Universidade do Colorado – Denver*
Richard Min	*Universidade do Texas em Dallas*

Robert McCloskey	*Universidade de Scranton*
Ryan Stansifer	*Instituto de Tecnologia da Flórida*
Salih Yurttas	*Universidade A&M do Texas*
Saverio Perugini	*Universidade de Dayton*
Serita Nelesen	*Faculdade Calvin*
Simon H. Lin	*Universidade do Estado da Califórnia – Northridge*
Stephen Edwards	*Virginia Tech*
Stuart C. Shapiro	*SUNY Buffalo*
Sumanth Yenduri	*Universidade do Sul de Mississippi*
Teresa Cole	*Universidade Estadual de Boise*
Thomas Turner	*Universidade de Oklahoma Central*
Tim R. Norton	*Universidade do Colorado – Colorado Springs*
Timothy Henry	*Universidade de Rhode Island*
Walter Pharr	*Faculdade de Charleston*
Xiangyan Zeng	*Universidade Estadual de Fort Valley*

Diversas outras pessoas contribuiram com sugestões para as edições anteriores de *Conceitos de linguagens de programação* em diferentes estágios de seu desenvolvimento. Todos os comentários foram úteis e detalhadamente considerados. Em ordem alfabética de sobrenomes, são eles: Vicki Allan, Henry Bauer, Carter Bays, Manuel E. Bermudez, Peter Brouwer, Margaret Burnett, Paosheng Chang, Liang Cheng, John Crenshaw, Charles Dana, Barbara Ann Griem, Mary Lou Haag, John V. Harrison, Eileen Head, Ralph C. Hilzer, Eric Joanis, Leon Jololian, Hikyoo Koh, Jiang B. Liu, Meiliu Lu, Jon Mauney, Robert McCoard, Dennis L. Mumaugh, Michael G. Murphy, Andrew Oldroyd, Young Park, Rebecca Parsons, Steve J. Phelps, Jeffery Popyack, Steven Rapkin, Hamilton Richard, Tom Sager, Raghvinder Sangwan, Joseph Schell, Sibylle Schupp, Mary Louise Soffa, Neelam Soundarajan, Ryan Stansifer, Steve Stevenson, Virginia Teller, Yang Wang, John M. Weiss, Franck Xia e Salih Yurnas.

Matt Goldstein (editor), Kelsey Loanes (assistente editorial), Scott Disanno (gerente de produtos líder de equipe), Pavithra Jayapaul e Mahalatchoumy Saravanan merecem minha gratidão por seus esforços para produzir a 11ª edição rápida e cuidadosamente.

Prefácio

MUDANÇAS NA 11ª EDIÇÃO

Os objetivos, a estrutura geral e a abordagem desta 11ª edição de *Conceitos de linguagens de programação* permanecem os mesmos das dez anteriores. Os objetivos principais são apresentar as construções fundamentais das linguagens de programação contemporâneas e fornecer ao leitor as ferramentas necessárias para uma avaliação crítica de linguagens existentes e futuras. Um objetivo secundário é preparar o leitor para o estudo de projeto de compiladores, fornecendo uma discussão aprofundada sobre estruturas de linguagens de programação, apresentando um método formal de descrição de sintaxe e explicitando estratégias para as análises sintática e léxica.

A 11ª edição evoluiu em relação à anterior devido a vários tipos diferentes de mudanças. Para garantir a manutenção do conteúdo atual, grande parte da discussão sobre linguagens de programação mais antigas, particularmente Ada e Fortran, foi removida. Por exemplo, as descrições dos registros, uniões e ponteiros de Ada foram retiradas do Capítulo 6. Do mesmo modo, a descrição da sentença **`for`** de Ada foi retirada do Capítulo 8 e a discussão dos tipos de dados abstratos de Ada foi retirada do Capítulo 11.

Por outro lado, no Capítulo 12 foi acrescentada uma seção sobre reflexão, incluindo dois exemplos de programa completos; no Capítulo 14 foi acrescentada uma seção sobre tratamento de exceções em Python e Ruby, e no Capítulo 12 foi acrescentada uma tabela das escolhas de projeto de algumas linguagens comuns para suportar programação orientada a objetos.

Em alguns casos, material foi deslocado. Por exemplo, a Seção 9.10 foi antecipada para se tornar a nova Seção 9.8.

Além disso, houve outra alteração: o exemplo de programa `MAIN_2` no Capítulo 10, anteriormente escrito em Ada, foi reescrito em JavaScript.

O Capítulo 12 foi significativamente revisado e recebeu vários parágrafos novos, duas novas seções e numerosas outras alterações realizadas para aumentar sua clareza.

A VISÃO

Este livro descreve os conceitos fundamentais de linguagens de programação ao discutir as questões de projeto de diversas construções de linguagens, examinando as escolhas para essas construções em algumas das linguagens mais comuns e comparando criticamente alternativas de projeto.

Qualquer estudo sério sobre linguagens de programação requer um exame de alguns tópicos relacionados, entre os quais estão os métodos de descrição de sintaxe e de semântica, tratados no Capítulo 3. Além disso, devem ser consideradas técnicas de implementação para várias construções da linguagem: as análises sintática e léxica são discutidas no Capítulo 4 e a implementação de ligação de subprogramas é abordada no Capítulo 10. A implementação de outras construções de linguagem é discutida em outras partes do livro.

DESCRIÇÃO DOS CAPÍTULOS

O Capítulo 1 começa com uma discussão sobre por que estudar linguagens de programação. Ele então aborda os critérios usados para avaliar linguagens e construções de linguagem. As principais influências no projeto de linguagens, algumas escolhas comuns de projeto e as estratégias básicas para a implementação também são examinadas.

O Capítulo 2 descreve a evolução das linguagens discutidas neste livro. Apesar de não ser feita nenhuma tentativa de descrever qualquer linguagem integralmente, são discutidas as origens, os propósitos e as contribuições de cada uma. Essa visão histórica é valiosa, porque fornece o conhecimento necessário para se entender as bases teóricas e práticas do projeto contemporâneo de linguagens. Também motiva o estudo adicional do projeto e da avaliação de linguagens. Já que nenhum dos capítulos restantes depende do Capítulo 2, ele pode ser lido independentemente.

O Capítulo 3 mostra o principal modelo formal para descrever a sintaxe de linguagens de programação – BNF. Em seguida, apresenta as gramáticas de atributos, que descrevem tanto a sintaxe quanto a semântica estática de linguagens. A difícil tarefa da descrição semântica é então explorada, incluindo breves introduções para os três métodos mais comuns: semântica operacional, denotacional e axiomática.

O Capítulo 4 introduz as análises léxica e sintática. Esse capítulo é voltado para os departamentos de ciência da computação que não exigem mais uma disciplina de projeto de compiladores em seu currículo. Como o Capítulo 2, este também é autônomo e pode ser estudado independentemente do restante do livro, a não ser pelo Capítulo 3, do qual ele depende.

Os Capítulos 5 a 14 descrevem em detalhes as questões de projeto para as principais construções das linguagens de programação. Em cada um dos casos são apresentadas e avaliadas as escolhas de projeto para diversas linguagens de exemplo. Especificamente, o Capítulo 5 aborda as muitas características das variáveis, o Capítulo 6 mostra os tipos de dados e o Capítulo 7 explica as expressões e as sentenças de atribuição. O Capítulo 8 descreve as sentenças de controle e os Capítulos 9 e 10 discutem os subprogramas e suas implementações. O Capítulo 11 examina os recursos de abstração de dados. O Capítulo 12 fornece uma discussão aprofundada dos recursos de linguagem que suportam a programação orientada a objetos (herança e vinculação dinâmica de métodos), o Capítulo 13 discute as unidades de programa concorrentes e o Capítulo 14 aborda o tratamento de exceções, com uma breve discussão sobre o tratamento de eventos.

Os Capítulos 15 e 16 descrevem dois dos paradigmas de programação mais importantes: a programação funcional e a programação lógica. Contudo, algumas das estruturas de dados e construções de controle das linguagens de programação funcional são discutidas nos Capítulos 6 e 8. O Capítulo 15 apresenta uma introdução à linguagem Scheme, incluindo descrições de algumas de suas funções primitivas, formas especiais e formas funcionais, além de alguns exemplos de funções simples escritas nela. São dadas introduções breves a ML, Haskell e F# para ilustrar alguns rumos diferentes no projeto de linguagens funcionais. O Capítulo 16 apresenta a programação lógica e a linguagem Prolog.

PARA O PROFESSOR

No início do curso de linguagens de programação na Universidade do Colorado, em Colorado Springs, este livro é usado da seguinte forma: estudamos os Capítulos 1 e 3 em detalhes, e – apesar de os estudantes acharem-no interessante e de leitura útil – o Capítulo 2 recebe pouco tempo devido à falta de conteúdo altamente técnico. Como nenhum material dos capítulos subsequentes depende do Capítulo 2, ele pode ser ignorado completamente. E, já que exigimos uma disciplina de projeto de compiladores, o Capítulo 4 não é estudado em aula.

Os Capítulos 5 a 9 são relativamente simples para estudantes com extensa experiência em programação em C++, Java ou C#. Os Capítulos 10 a 14 são mais desafiadores e exigem leituras mais atentas.

Os Capítulos 15 e 16 são inteiramente novos para a maioria dos alunos principiantes. De preferência, processadores de linguagem para Scheme e Prolog devem estar disponíveis para que o estudo desses capítulos. É incluído material suficiente para permitir que os estudantes lidem com alguns programas simples.

Cursos de graduação provavelmente não serão capazes de abordar todo o conteúdo dos últimos dois capítulos. Cursos de pós-graduação, entretanto, conseguirão discutir o material na íntegra, pulando algumas partes dos primeiros capítulos acerca de linguagens imperativas.

MATERIAIS COMPLEMENTARES

Os seguintes materiais estão disponíveis online no site *loja.grupoa.com.br*.

Para os estudantes:

- Todas as figuras do livro em apresentações em PowerPoint (em português)

Para os professores:

- Um conjunto de apresentações em PowerPoint com notas de aulas (em inglês)
- Todas as figuras do livro em apresentações em PowerPoint (em português)
- Manual de soluções (em inglês)

DISPONIBILIDADE DE PROCESSADOR PARA A LINGUAGEM

Processadores e informações acerca de algumas das linguagens de programação discutidas neste livro podem ser encontrados nestes sites:

C, C++, Fortran e Ada	gcc.gnu.org
C# e F#	microsoft.com
Java	java.sun.com
Haskell	haskell.org
Lua	www.lua.org
Scheme	www.plt-scheme.org/software/drscheme
Perl	www.perl.com
Python	www.python.org
Ruby	www.ruby-lang.org

JavaScript está incluso em praticamente todos os navegadores; PHP integra quase todos os servidores Web.

Sumário

Capítulo 1 Preliminares 1

1.1 Razões para estudar conceitos de linguagens de programação 2
1.2 Domínios de programação 5
1.3 Critérios de avaliação de linguagens 6
1.4 Influências no projeto de linguagens 17
1.5 Categorias de linguagens 20
1.6 *Trade-offs* no projeto de linguagens 21
1.7 Métodos de implementação 22
1.8 Ambientes de programação 29
Resumo • Questões de revisão • Problemas 31

Capítulo 2 Evolução das principais linguagens de programação 33

2.1 Plankalkül de Zuse 36
2.2 Pseudocódigos 37
2.3 IBM 704 e Fortran 40
2.4 Programação funcional: Lisp 45
2.5 O primeiro passo em direção à sofisticação: ALGOL 60 50
2.6 Informatização de registros comerciais: COBOL 56
2.7 O início do compartilhamento de tempo: Basic 60
2.8 Tudo para todos: PL/I 63
Entrevista: ALAN COOPER – Projeto de usuário e projeto de linguagens 64
2.9 Duas das primeiras linguagens dinâmicas: APL e SNOBOL 69
2.10 O começo da abstração de dados: SIMULA 67 70
2.11 Projeto ortogonal: ALGOL 68 71
2.12 Alguns dos primeiros descendentes de ALGOLs 72
2.13 Programação baseada em lógica: Prolog 76
2.14 O maior esforço de projeto da história: Ada 78
2.15 Programação orientada a objetos: Smalltalk 82
2.16 Combinação de recursos imperativos e orientados a objetos: C++ 84
2.17 Uma linguagem orientada a objetos baseada no paradigma imperativo: Java 87

2.18 Linguagens de *scripting* 90
2.19 A principal linguagem .NET: C# 97
2.20 Linguagens híbridas de marcação-programação 100
Resumo • Notas bibliográficas • Questões de revisão • Problemas • Exercícios de programação 107

Capítulo 3 Descrição da sintaxe e da semântica 109
3.1 Introdução 110
3.2 O problema geral de descrever sintaxe 111
3.3 Métodos formais para descrever sintaxe 113
3.4 Gramáticas de atributos 128
3.5 Descrição do significado de programas: semântica dinâmica 134
Resumo • Notas bibliográficas • Questões de revisão • Problemas 157

Capítulo 4 Análise léxica e sintática 161
4.1 Introdução 162
4.2 Análise léxica 163
4.3 O problema da análise sintática 171
4.4 Análise sintática descendente recursiva 174
4.5 Análise sintática ascendente 183
Resumo • Questões de revisão • Problemas • Exercícios de programação 195

Capítulo 5 Nomes, vinculações e escopos 197
5.1 Introdução 198
5.2 Nomes 199
5.3 Variáveis 201
5.4 O conceito de vinculação 203
5.5 Escopo 211
5.6 Escopo e tempo de vida 222
5.7 Ambientes de referenciamento 222
5.8 Constantes nomeadas 224
Resumo • Questões de revisão • Problemas • Exercícios de programação 232

Capítulo 6 Tipos de dados 233
6.1 Introdução 234
6.2 Tipos de dados primitivos 236
6.3 Cadeias de caracteres 240
6.4 Tipos enumeração 245
6.5 Tipos de matrizes 247
6.6 Matrizes associativas 259

Entrevista: ROBERTO IERUSALIMSCHY – Lua 260
6.7 Registros 263
6.8 Tuplas 266
6.9 Listas 268
6.10 Uniões 270
6.11 Ponteiros e referências 273
6.12 Verificação de tipos 285
6.13 Tipagem forte 286
6.14 Equivalência de tipos 287
6.15 Teoria e tipos de dados 291
Resumo • Notas bibliográficas • Questões de revisão • Problemas • Exercícios de programação 297

Capítulo 7 Expressões e sentenças de atribuição 299
7.1 Introdução 300
7.2 Expressões aritméticas 300
7.3 Operadores sobrecarregados 309
7.4 Conversões de tipos 310
7.5 Expressões relacionais e booleanas 313
7.6 Avaliação em curto-circuito 315
7.7 Sentenças de atribuição 317
7.8 Atribuição de modo misto 321
Resumo • Questões de revisão • Problemas • Exercícios de programação 326

Capítulo 8 Estruturas de controle no nível de sentença 327
8.1 Introdução 328
8.2 Sentenças de seleção 330
8.3 Sentenças de iteração 341
8.4 Desvio incondicional 352
8.5 Comandos protegidos 353
8.6 Conclusões 355
Resumo • Questões de revisão • Problemas • Exercícios de programação 359

Capítulo 9 Subprogramas 363
9.1 Introdução 364
9.2 Fundamentos dos subprogramas 364
9.3 Questões de projeto para subprogramas 372
9.4 Ambientes de referenciamento local 373
9.5 Métodos de passagem de parâmetros 375
9.6 Parâmetros que são subprogramas 391

9.7 Chamada indireta de subprogramas 393
9.8 Questões de projeto para funções 395
9.9 Subprogramas sobrecarregados 397
9.10 Subprogramas genéricos 398
9.11 Operadores sobrecarregados definidos pelo usuário 404
9.12 Fechamentos 404
9.13 Corrotinas 406
Resumo • Questões de revisão • Problemas • Exercícios de programação 413

Capítulo 10 Implementação de subprogramas 415
10.1 A semântica geral de chamadas e retornos 416
10.2 Implementação de subprogramas "simples" 416
10.3 Implementação de subprogramas com variáveis locais dinâmicas da pilha 419
10.4 Subprogramas aninhados 427
10.5 Blocos 434
10.6 Implementação de escopo dinâmico 436
Resumo • Questões de revisão • Problemas • Exercícios de programação 444

Capítulo 11 Tipos de dados abstratos e construções de encapsulamento 445
11.1 O conceito de abstração 446
11.2 Introdução à abstração de dados 447
11.3 Questões de projeto para tipos de dados abstratos 450
11.4 Exemplos de linguagem 451
Entrevista: BJARNE STROUSTRUP++: nascimento, onipresença e críticas comuns 452
11.5 Tipos de dados abstratos parametrizados 470
11.6 Construções de encapsulamento 474
11.7 Nomeação de encapsulamentos 478
Resumo • Questões de revisão • Problemas • Exercícios de programação 485

Capítulo 13 Suporte para programação orientada a objetos 487
12.1 Introdução 488
12.2 Programação orientada a objetos 489
12.3 Questões de projeto para linguagens orientadas a objetos 493
12.4 Suporte para programação orientada a objetos em linguagens específicas 498
Entrevista: BJARNE STROUSTRUP: Sobre paradigmas e uma programação melhor 502
12.5 Implementação de construções orientadas a objetos 526

12.6 Reflexão 529
Resumo • Questões de revisão • Problemas • Exercícios de programação 540

Capítulo 13 Concorrência 541
13.1 Introdução 542
13.2 Introdução à concorrência em nível de subprograma 547
13.3 Semáforos 552
13.4 Monitores 557
13.5 Passagem de mensagens 559
13.6 Suporte de Ada para concorrência 560
13.7 Linhas de execução em Java 568
13.8 Linhas de execução em C# 578
13.9 Concorrência em linguagens funcionais 583
13.10 Concorrência em nível de sentença 586
Resumo • Notas bibliográficas • Questões de revisão • Problemas • Exercícios de programação 593

Capítulo 14 Tratamento de exceções e tratamento de eventos 595
14.1 Introdução ao tratamento de exceções 596
14.2 Tratamento de exceções em C++ 602
14.3 Tratamento de exceções em Java 606
14.4 Tratamento de exceções em Python e Ruby 613
14.5 Introdução ao tratamento de eventos 616
14.6 Tratamento de eventos com Java 617
14.7 Tratamento de eventos em C# 622
Resumo • Notas bibliográficas • Questões de revisão • Problemas • Exercícios de programação 629

Capítulo 15 Linguagens de programação funcional 631
15.1 Introdução 632
15.2 Funções matemáticas 633
15.3 Fundamentos das linguagens de programação funcional 636
15.4 A primeira linguagem de programação funcional: Lisp 637
15.5 Uma Introdução a Scheme 641
15.6 Common Lisp 659
15.7 ML 661
15.8 Haskell 666
15.9 F# 671
15.10 Suporte para programação funcional em linguagens basicamente imperativas 674
15.11 Uma comparação entre linguagens funcionais e imperativas 677

Resumo • Notas bibliográficas • Questões de revisão • problemas • Exercícios de programação 684

Capítulo 16 Linguagens de programação lógica 687

16.1 Introdução 688
16.2 Uma breve introdução ao cálculo de predicados 688
16.3 Cálculo de predicados e prova de teoremas 692
16.4 Panorama da programação lógica 694
16.5 As origens de Prolog 696
16.6 Os elementos básicos de Prolog 696
16.7 Deficiências de Prolog 711
16.8 Aplicações de programação lógica 716

Resumo • Notas bibliográficas • Questões de revisão • Problemas • Exercícios de programação 721

Bibliografia 723

1

Preliminares

1.1 Razões para estudar conceitos de linguagens de programação

1.2 Domínios de programação

1.3 Critérios de avaliação de linguagens

1.4 Influências no projeto de linguagens

1.5 Categorias de linguagens

1.6 *Trade-offs* no projeto de linguagens

1.7 Métodos de implementação

1.8 Ambientes de programação

Antes de iniciarmos a discussão sobre os conceitos de linguagens de programação, precisamos considerar aspectos preliminares. Primeiro, explicaremos algumas razões pelas quais os estudantes de ciência da computação e os desenvolvedores de software profissionais devem estudar conceitos gerais sobre o projeto e a avaliação de linguagens. Essa discussão é particularmente valiosa para quem acredita que um conhecimento funcional de uma ou duas linguagens de programação é suficiente para cientistas da computação. Na sequência, descreveremos brevemente os principais domínios de programação. A seguir, como o livro avalia construções e recursos de linguagens, apresentaremos uma lista de critérios que podem servir de base para tais julgamentos. Então, discutiremos as duas maiores influências no projeto de linguagens: a arquitetura de máquinas e as metodologias de projeto de programas. Depois, introduziremos as diversas categorias de linguagens de programação. A seguir, descreveremos alguns dos principais *trade offs* durante o projeto de linguagens.

Como este livro trata também da implementação de linguagens de programação, o Capítulo 1 inclui um panorama das abordagens mais comuns para a implementação. Finalmente, descrevemos alguns exemplos de ambientes de programação e discutimos seu impacto na produção de software.

1.1 RAZÕES PARA ESTUDAR CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO

É natural que os estudantes se perguntem como se beneficiarão com o estudo de conceitos de linguagens de programação. Afinal, muitos outros tópicos em ciência da computação são merecedores de um estudo sério. A seguir, temos uma lista de potenciais vantagens de estudar esses conceitos:

- *Aumento da capacidade de expressar ideias.* Acredita-se que a profundidade com a qual as pessoas podem pensar é influenciada pelo poder de expressão da linguagem que usam para comunicar seus pensamentos. As pessoas com fraco entendimento da linguagem natural são limitadas na complexidade de seus pensamentos, particularmente na profundidade de abstração. Em outras palavras, é difícil para essas pessoas criar conceitos de estruturas que elas não conseguem descrever verbalmente ou expressar na escrita.

  Programadores no processo de desenvolvimento de software apresentam a mesma limitação. A linguagem na qual eles desenvolvem software impõe restrições nos tipos de estruturas de controle, estruturas de dados e abstrações que podem usar – logo, as formas dos algoritmos que eles constroem também são limitadas. Conhecer uma variedade mais ampla de recursos das linguagens de programação pode reduzir essas limitações no desenvolvimento de software. Os programadores podem aumentar a divesidade de seus processos mentais de desenvolvimento de software ao aprender novas construções de linguagens.

  Pode-se argumentar que aprender os recursos de outras linguagens não ajuda um programador obrigado a usar determinada linguagem que não tem tais recursos. Esse argumento não se sustenta, porque normalmente as construções das linguagens podem ser simuladas em outras que não as suportam diretamente. Por exemplo, um programador de C que tenha aprendido a estrutura e os usos das matrizes associa-

tivas em Perl (Christianson et al., 2012) poderá projetar estruturas que simulem as matrizes associativas dessa linguagem. Em outras palavras, o estudo dos conceitos de linguagem de programação valoriza recursos e construções valiosos da linguagem e estimula os programadores a usá-los, mesmo quando a linguagem que estão utilizando não suporta diretamente tais recursos e construções.

- *Embasamento para escolher linguagens adequadas.* Alguns programadores profissionais tiveram pouca educação formal em ciência da computação – em vez disso, desenvolveram suas habilidades em programação independentemente ou em programas de treinamento em suas empresas. Tais programas normalmente limitam o ensino a uma ou duas linguagens diretamente relevantes para os projetos atuais da organização. Outros programadores receberam seu treinamento formal há anos. As linguagens que aprenderam na época não são mais usadas e muitos recursos agora disponíveis não eram amplamente conhecidos então. O resultado é que muitos programadores, quando podem escolher a linguagem para um novo projeto, usam aquela com a qual estão mais familiarizados, mesmo que não seja a mais adequada ao projeto. Se esses programadores conhecessem uma gama mais ampla de linguagens e construções de linguagens, estariam mais capacitados a escolher a que inclui os recursos mais bem adaptados para tratar do problema.

  Alguns dos recursos de uma linguagem podem ser simulados em outra. Entretanto, é preferível usar um recurso cujo projeto tenha sido integrado em uma linguagem do que usar uma simulação – normalmente menos elegante, de manipulação mais difícil e menos segura.

- *Aumento da habilidade para aprender novas linguagens.* A programação de computadores ainda é uma disciplina relativamente nova e as metodologias de projeto, ferramentas de desenvolvimento de software e linguagens de programação ainda estão em evolução. Isso torna o desenvolvimento de software uma profissão empolgante, mas também exige aprendizado contínuo. O processo de aprender uma nova linguagem de programação pode ser longo e difícil, especialmente para alguém que se sinta à vontade com apenas uma ou duas e que nunca examinou os conceitos de linguagens de programação de um modo geral. Uma vez que um entendimento preciso dos conceitos fundamentais das linguagens tenha sido adquirido, fica mais fácil ver como esses conceitos são incorporados no projeto da linguagem aprendida. Por exemplo, programadores que entendem os conceitos de programação orientada a objetos aprenderão Ruby muito mais facilmente (Thomas et al., 2013) do que aqueles que nunca usaram tais conceitos.

  O mesmo ocorre nas linguagens naturais. Quanto melhor você conhece a gramática de seu idioma nativo, mais fácil será aprender uma segunda língua. Além disso, aprender uma segunda língua tem a vantagem de ensinar a você mais sobre a primeira.

  A Comunidade de Programação TIOBE disponibiliza um índice (http://www.tiobe.com/index.php/content/paperinfo/tpci/index.htm) que funciona como indicador da popularidade relativa das linguagens de programação. Por exemplo, de acordo com o índice, C, Java e Objective-C foram as três linguagens mais populares em uso em fevereiro de 2014.[1] Entretanto, dezenas de outras linguagens foram ampla-

[1] Observe que esse índice é apenas uma medida da popularidade das linguagens de programação e sua precisão não é aceita universalmente.

mente usadas nessa época. Os dados do índice também mostram que a distribuição do uso das linguagens de programação está sempre mudando. O número de linguagens em uso e a natureza dinâmica das estatísticas implicam que os desenvolvedores de software devem estar preparados para aprender linguagens diferentes.

Por fim, é essencial que programadores em atividade conheçam o vocabulário e os conceitos fundamentais das linguagens de programação para poderem ler e entender suas descrições e avaliações, assim como a literatura promocional de linguagens e compiladores. Essas são as fontes de informações necessárias para escolher e aprender uma linguagem.

- *Melhor entendimento da importância da implementação.* Ao aprender os conceitos de linguagens de programação, é tão interessante quanto necessário abordar aspectos de implementação que afetam esses conceitos. Em alguns casos, um entendimento de questões de implementação leva ao motivo pelo qual as linguagens foram projetadas de determinada forma. Por sua vez, esse conhecimento leva à habilidade de usar uma linguagem de maneira mais inteligente, conforme foi projetada para ser usada. Podemos ser programadores melhores ao entender as escolhas entre construções de linguagens de programação e as consequências dessas escolhas.

  Certos tipos de erros em programas podem ser encontrados e corrigidos apenas por programadores que conhecem alguns detalhes de implementação relacionados. Outro benefício de entender questões de implementação é permitir a visualização da forma como um computador executa as diversas construções de uma linguagem. Em certos casos, algum conhecimento sobre questões de implementação fornece dicas sobre a eficiência relativa de construções alternativas que podem ser escolhidas para um programa. Por exemplo, programadores que conhecem pouco sobre a complexidade da implementação de chamadas a subprogramas muitas vezes não se dão conta de que um pequeno subprograma chamado frequentemente pode ser uma decisão de projeto altamente ineficiente.

  Como este livro aborda apenas algumas questões de implementação, os dois parágrafos anteriores também servem como uma explicação das vantagens de estudar o projeto de compiladores.

- *Melhor uso de linguagens já conhecidas.* Em sua maioria, as linguagens de programação contemporâneas são extensas e complexas. É incomum um programador conhecer e usar todos os recursos da linguagem que utiliza. Ao estudar os conceitos de linguagens de programação, os programadores podem aprender sobre partes antes desconhecidas e não utilizadas das linguagens com que já trabalham e começar a utilizá-las.

- *Avanço geral da computação.* Por fim, existe uma visão geral de computação que pode justificar o estudo de conceitos de linguagens de programação. Apesar de normalmente ser possível determinar por que determinada linguagem se tornou popular, muitos acreditam, ao menos em retrospecto, que as linguagens de programação mais populares nem sempre são as melhores disponíveis. Em alguns casos, pode-se concluir que uma linguagem se tornou amplamente usada, ao menos em parte, porque aqueles em posições de escolha não estavam suficientemente familiarizados com conceitos de linguagens de programação.

Por exemplo, muitas pessoas acreditam que teria sido melhor se ALGOL 60 (Backus et al., 1963) tivesse substituído Fortran (McCracken, 1961) no início dos anos 1960, porque era mais elegante e tinha sentenças de controle muito melhores, entre outras razões. Ela não a substituiu, em parte por causa dos programadores e dos gerentes de desenvolvimento de software da época; muitos não entendiam o projeto conceitual de ALGOL 60. Eles achavam sua descrição difícil de ler (o que era verdade) e mais difícil ainda de entender. Eles não gostaram das vantagens da estrutura de blocos, da recursão e das sentenças de controle bem estruturadas, então falharam em ver as melhorias de ALGOL 60 em relação a Fortran.

É claro, muitos outros fatores contribuíram para a falta de aceitação de ALGOL 60, como veremos no Capítulo 2. Entretanto, o fato de os usuários de computadores não estarem cientes das vantagens da linguagem teve um papel significativo em sua rejeição.

Em geral, se aqueles que escolhem as linguagens fossem bem informados, talvez linguagens melhores superassem outras piores.

1.2 DOMÍNIOS DE PROGRAMAÇÃO

Os computadores são utilizados em uma infinidade de tarefas, desde controlar usinas nucleares até disponibilizar jogos eletrônicos em telefones celulares. Por causa dessa diversidade de uso, linguagens de programação com objetivos muito diferentes foram desenvolvidas. Nesta seção, discutimos brevemente algumas das áreas de aplicação dos computadores e das linguagens a eles associadas.

1.2.1 Aplicações científicas

Os primeiros computadores digitais, surgidos no final dos anos 1940, início dos anos 1950, foram desenvolvidos e usados para aplicações científicas. Normalmente, as aplicações científicas daquela época utilizavam estruturas de dados relativamente simples, mas exigiam diversos cálculos aritméticos de ponto flutuante. As estruturas de dados mais comuns eram os vetores e as matrizes; as estruturas de controle mais comuns eram os laços de contagem e as seleções. As primeiras linguagens de programação de alto nível inventadas para aplicações científicas foram projetadas para suprir tais necessidades. Sua concorrente era a linguagem assembly – logo, a eficiência era uma preocupação primordial. A primeira linguagem para aplicações científicas foi Fortran. ALGOL 60 e a maioria de seus descendentes também tinham a intenção de serem usados nessa área, apesar de terem sido projetados também em áreas relacionadas. Para aplicações científicas nas quais a eficiência é a principal preocupação, como as que eram comuns nos anos 1950 e 1960, nenhuma linguagem subsequente é significativamente melhor que Fortran, o que explica por que ela ainda é usada.

1.2.2 Aplicações empresariais

O uso de computadores para aplicações comerciais começou nos anos 1950. Computadores especiais foram desenvolvidos para esse propósito, com linguagens especiais. A

primeira linguagem de alto nível para negócios a ser bem-sucedida foi o COBOL (ISO/IEC, 2002), com sua primeira versão aparecendo em 1960. COBOL provavelmente ainda é a linguagem mais utilizada para tais aplicações. Linguagens de negócios são caracterizadas por facilidades para a produção de relatórios elaborados, maneiras precisas de descrever e armazenar números decimais e caracteres, e a habilidade de especificar operações aritméticas decimais.

Poucos avanços ocorreram nas linguagens para aplicações comerciais além do desenvolvimento e da evolução de COBOL. Logo, este livro inclui apenas discussões limitadas sobre as estruturas em COBOL.

1.2.3 Inteligência artificial

A Inteligência Artificial (IA) é uma ampla área de aplicações computacionais caracterizadas pelo uso de computações simbólicas em vez de numéricas. Computações simbólicas são aquelas nas quais símbolos, compostos de nomes em vez de números, são manipulados. Além disso, a computação simbólica é feita de modo mais fácil por meio de listas ligadas de dados do que por meio de vetores. Esse tipo de programação algumas vezes requer mais flexibilidade do que outros domínios de programação. Por exemplo, em algumas aplicações de IA, a capacidade de criar e executar segmentos de código durante a execução é conveniente.

A primeira linguagem de programação amplamente utilizada, desenvolvida para aplicações de IA, foi a linguagem funcional Lisp (McCarthy et al., 1965), que apareceu em 1959. A maioria das aplicações de IA desenvolvidas antes de 1990 foi escrita em Lisp ou em um de seus parentes próximos. No início dos anos 1970, entretanto, uma abordagem alternativa a algumas dessas aplicações apareceu – programação lógica usando a linguagem Prolog (Clocksin e Mellish, 2013). Mais recentemente, algumas aplicações de IA foram escritas em linguagens de sistemas como C. Scheme (Dybvig, 2009), um dialeto de Lisp, e Prolog são apresentadas nos Capítulos 15 e 16, respectivamente.

1.2.4 Software para a Web

A World Wide Web é mantida por uma eclética coleção de linguagens, que vão desde linguagens de marcação, como HTML, que não é de programação, até linguagens de programação de propósito geral, como Java. Dada a necessidade universal de conteúdo dinâmico na Web, alguma capacidade de computação geralmente é incluída na tecnologia de apresentação de conteúdo. Essa funcionalidade pode ser fornecida por código de programação embarcado em um documento HTML. Tal código é normalmente escrito com uma linguagem de *scripting*, como JavaScript ou PHP (Tatroe, 2013). Existem também algumas linguagens similares às de marcação que têm sido estendidas para incluir construções que controlam o processamento de documentos, as quais são discutidas na Seção 1.5 e no Capítulo 2.

1.3 CRITÉRIOS DE AVALIAÇÃO DE LINGUAGENS

Conforme mencionado, o objetivo deste livro é examinar os conceitos fundamentais das diversas construções e capacidades das linguagens de programação. Vamos também ava-

liar esses recursos, enfocando seu impacto no processo de desenvolvimento de software, incluindo a manutenção. Para isso, precisamos de um conjunto de critérios de avaliação. Tal lista é necessariamente controversa, porque é difícil fazer dois cientistas da computação concordarem com o valor de certas características das linguagens em relação às outras. Apesar dessas diferenças, a maioria concordaria que os critérios discutidos nas subseções a seguir são importantes.

Algumas das características que influenciam três dos quatro critérios mais importantes são mostradas na Tabela 1.1, e os critérios propriamente ditos são discutidos nas seções seguintes.[2] Note que apenas as características mais importantes são incluídas na tabela, espelhando a discussão nas subseções seguintes. Alguém poderia argumentar que, se fossem consideradas características menos importantes, praticamente todas as posições da tabela poderiam ser marcadas.

Note que algumas características são amplas – e, de certa forma, vagas –, como a facilidade de escrita, enquanto outras são construções específicas de linguagens, como o tratamento de exceções. Além disso, apesar de a discussão parecer implicar que os critérios têm uma importância idêntica, não é o caso.

1.3.1 Legibilidade

Um dos critérios mais importantes para julgar uma linguagem de programação é a facilidade com que os programas podem ser lidos e entendidos. Antes de 1970, o desenvolvimento de software era amplamente pensado em termos da escrita de código. A principal característica positiva das linguagens de programação era a eficiência. As construções das linguagens foram projetadas mais do ponto de vista do computador do que do dos usuários. Nos anos 1970, entretanto, o conceito de ciclo de vida de software (Booch, 1987) foi desenvolvido; a codificação foi relegada a um papel muito menor e a manutenção foi reconhecida como uma parte importante do ciclo, particularmente em

TABELA 1.1 Critérios de avaliação de linguagens e as características que os afetam

	CRITÉRIOS		
Característica	**Legibilidade**	**Facilidade de escrita**	**Confiabilidade**
Simplicidade	•	•	•
Ortogonalidade	•	•	•
Tipos de dados	•	•	•
Projeto de sintaxe	•	•	•
Suporte para abstração		•	•
Expressividade		•	•
Verificação de tipos			•
Tratamento de exceções			•
Apelidos restritos			•

[2] O quarto critério importante é o custo, o qual não está incluído na tabela porque não tem muita relação com os outros critérios e com as características que os influenciam.

termos de custo. Como a facilidade de manutenção é determinada, em grande parte, pela legibilidade dos programas, esta se tornou uma medida importante da qualidade dos programas e das linguagens de programação. Isso foi um marco importante na evolução das linguagens de programação. Ocorreu uma transição de foco bem definida: da orientação à máquina para a orientação às pessoas.

A legibilidade deve ser considerada no contexto do domínio do problema. Por exemplo, se um programa que descreve um cálculo é escrito em uma linguagem que não foi projetada para tal uso, ele pode não ser natural e ser desnecessariamente complexo, tornando complicada sua leitura (quando, em geral, seria algo simples).

As subseções a seguir descrevem características que contribuem para a legibilidade de uma linguagem de programação.

1.3.1.1 Simplicidade geral

A simplicidade geral de uma linguagem de programação afeta muito sua legibilidade. Uma linguagem com muitas construções básicas é mais difícil de aprender do que uma com poucas. Os programadores que precisam usar uma linguagem extensa aprendem um subconjunto dessa linguagem e ignoram outros recursos. Esse padrão de aprendizagem é usado como desculpa para a grande quantidade de construções de uma linguagem, mas o argumento não é válido. Problemas de legibilidade ocorrem sempre que o autor de um programa aprendeu um subconjunto diferente daquele com o qual o leitor está familiarizado.

Uma segunda característica de uma linguagem de programação que pode ser um complicador é a **multiplicidade de recursos** – ou seja, haver mais de uma maneira de realizar uma operação. Por exemplo, em Java um usuário pode incrementar uma simples variável inteira de quatro maneiras:

```
count = count + 1
count += 1
count++
++count
```

Apesar de as últimas duas sentenças terem significados um pouco diferentes uma da outra e em relação às duas primeiras em alguns contextos, todas elas têm o mesmo significado quando usadas em expressões isoladas. Essas variações são discutidas no Capítulo 7.

Um terceiro problema em potencial é a **sobrecarga de operadores**, na qual um operador tem mais de um significado. Apesar de ser útil, pode levar a uma redução da legibilidade se for permitido aos usuários criar suas próprias sobrecargas e eles não o fizerem de maneira sensata. Por exemplo, é aceitável sobrecarregar o operador + para usá-lo tanto para a adição de inteiros quanto para a adição de valores de ponto flutuante. Na verdade, essa sobrecarga simplifica uma linguagem ao reduzir o número de operadores. Entretanto, suponha que o programador definiu o símbolo + usado entre vetores de uma única dimensão para significar a soma de todos os elementos de ambos os vetores. Como o significado usual da adição de vetores é bastante diferente, esse significado incomum poderia confundir o autor e os leitores do programa. Um exemplo ainda mais extremo da confusão em programas seria um usuário definir que o + entre dois operandos do tipo

vetor é a diferença entre os primeiros elementos de cada vetor. A sobrecarga de operadores é discutida com mais detalhes no Capítulo 7.

A simplicidade em linguagens pode, é claro, ser levada ao extremo. Por exemplo, a forma e o significado da maioria das sentenças de uma linguagem assembly são modelos de simplicidade, como você pode ver quando considera as sentenças que aparecem na próxima seção. Essa simplicidade extrema, entretanto, torna menos legíveis os programas escritos em linguagem assembly. Devido à falta de sentenças de controle mais complexas, a estrutura de um programa é menos óbvia; como as sentenças são simples, mais delas são necessárias do que em programas equivalentes escritos em uma linguagem de alto nível. Os mesmos argumentos se aplicam para os casos menos extremos de linguagens de alto nível com construções inadequadas de controle e de estruturas de dados.

1.3.1.2 Ortogonalidade

Ortogonalidade em uma linguagem de programação significa que um conjunto relativamente pequeno de construções primitivas pode ser combinado a um número relativamente pequeno de formas para construir as estruturas de controle e de dados da linguagem. Além disso, cada possível combinação de primitivas é válida e significativa. Por exemplo, considere os tipos de dados. Suponha que uma linguagem tenha quatro tipos primitivos de dados (inteiro, ponto flutuante, ponto flutuante de dupla precisão e caractere) e dois operadores de tipo (vetor e ponteiro). Se os dois operadores de tipo puderem ser aplicados a eles mesmos e aos quatro tipos de dados primitivos, um grande número de estruturas de dados pode ser definido.

O significado de um recurso de linguagem ortogonal é independente do contexto de sua aparição em um programa. (A palavra *ortogonal* é proveniente do conceito matemático de vetores ortogonais, os quais são independentes entre si.) A ortogonalidade vem de uma simetria de relacionamentos entre primitivas. A falta de ortogonalidade leva a exceções às regras da linguagem. Por exemplo, em uma linguagem de programação que suporta ponteiros, deve ser possível definir um ponteiro que aponte para qualquer tipo específico nela definido. Entretanto, se não for permitido aos ponteiros apontar para vetores, muitas estruturas de dados potencialmente úteis definidas pelos usuários não poderão ser definidas.

Podemos ilustrar o uso da ortogonalidade como conceito de projeto ao comparar um aspecto das linguagens assembly dos mainframes da IBM com a série VAX de minicomputadores. Consideramos uma situação simples: somar dois valores inteiros de 32 bits que residem na memória ou nos registradores e substituir um dos valores pela soma. Os mainframes IBM têm duas instruções para esse propósito, com a forma

```
A Reg1, memory_cell
AR Reg1, Reg2
```

onde `Reg1` e `Reg2` representam registros. A semântica deles é

```
Reg1 ← contents(Reg1) + contents(memory_cell)
Reg1 ← contents(Reg1) + contents(Reg2)
```

A instrução de adição VAX para valores inteiros de 32 bits é

```
ADDL operando_1, operando_2
```

cuja semântica é

```
operando_2 ← contents(operando_1) + contents(operando_2)
```

Nesse caso, cada operando pode ser um registrador ou uma célula de memória.

O projeto de instruções VAX é ortogonal no sentido de que uma instrução pode usar tanto registradores quanto células de memória como operandos. Existem duas maneiras de especificar operandos, as quais podem ser combinadas de todas as formas possíveis. O projeto da IBM não é ortogonal. Apenas duas das quatro possibilidades de combinações de operandos são válidas, e ambas necessitam de instruções diferentes, `A` e `AR`. O projeto da IBM é mais restrito e tem menor facilidade de escrita. Por exemplo, você não pode somar dois valores e armazenar a soma em um local de memória. Além disso, o projeto da IBM é mais difícil de aprender por causa das restrições e da instrução adicional.

A ortogonalidade está intimamente relacionada à simplicidade: quanto mais ortogonal o projeto de uma linguagem, menor é o número necessário de exceções às regras da linguagem. Menos exceções significam um maior grau de regularidade no projeto, o que torna a linguagem mais fácil de aprender, ler e entender. Qualquer um que tenha aprendido uma parte significativa da língua inglesa pode atestar a dificuldade de aprender suas muitas exceções à regra (por exemplo, *i* antes de *e*, exceto após *c*).

Como exemplos da falta de ortogonalidade em uma linguagem de alto nível, considere as seguintes regras e exceções em C. Apesar de C ter duas formas de tipos de dados estruturados, vetores e registros (**`struct`**s), os registros podem ser retornados por funções, mas os vetores não. Um membro de uma estrutura pode ser qualquer tipo de dados, exceto **`void`** ou uma estrutura do mesmo tipo. Um elemento de um vetor pode ser qualquer tipo de dado, exceto **`void`** ou uma função. Parâmetros são passados por valor, a menos que sejam vetores, o que faz com que sejam passados por referência (porque a ocorrência de um nome de um vetor sem um índice em um programa em C é interpretada como o endereço do primeiro elemento desse vetor).

Como exemplo da dependência do contexto, considere a seguinte expressão em C:

```
a + b
```

Ela significa que os valores de `a` e `b` são obtidos e somados. Entretanto, se `a` for um ponteiro e `b` um inteiro, isso afeta o valor de `b`. Por exemplo, se `a` aponta para um valor de ponto flutuante que ocupa 4 bytes, o valor de `b` deve ser ampliado – nesse caso, multiplicado por 4 – antes que seja somado a `a`. Logo, o tipo de `a` afeta o tratamento do valor de `b`. O contexto de `b` afeta seu significado.

Ortogonalidade demais também pode causar problemas. Talvez a linguagem de programação mais ortogonal seja ALGOL 68 (van Wijngaarden et al., 1969). Cada construção de linguagem em ALGOL 68 tem um tipo, e não existem restrições nesses tipos. Além disso, a maioria das construções produz valores. Essa liberdade de combinações permite construções extremamente complexas. Por exemplo, uma sentença condicional

pode aparecer no lado esquerdo de uma atribuição, com declarações e outras sentenças diversas, desde que o resultado seja um endereço. Essa forma extrema de ortogonalidade leva a uma complexidade desnecessária. Como as linguagens necessitam de um grande número de tipos primitivos, um alto grau de ortogonalidade resulta em uma explosão de combinações. Então, mesmo se as combinações forem simples, seu número já leva à complexidade.

Simplicidade em uma linguagem é, ao menos em parte, o resultado da combinação de um número relativamente pequeno de construções primitivas com o uso limitado do conceito de ortogonalidade.

Algumas pessoas acreditam que as linguagens funcionais oferecem uma boa combinação de simplicidade e ortogonalidade. Uma linguagem funcional, como Lisp, é aquela na qual as computações são feitas basicamente pela aplicação de funções a parâmetros dados. Em contraste a isso, em linguagens imperativas como C, C++ e Java, as computações são normalmente especificadas com variáveis e sentenças de atribuição. As linguagens funcionais oferecem a maior simplicidade de um modo geral, porque podem realizar tudo com uma construção, a chamada à função, a qual pode ser combinada com outras funções de maneiras simples. Essa elegância simples é a razão pela qual alguns pesquisadores de linguagens são atraídos pelas linguagens funcionais como principal alternativa às complexas linguagens não funcionais, como Java. Entretanto, outros fatores, como a eficiência, têm impedido que as linguagens funcionais sejam mais utilizadas.

1.3.1.3 Tipos de dados

A presença de mecanismos adequados para definir tipos e estruturas de dados é outro auxílio significativo à legibilidade. Por exemplo, suponha que um tipo numérico seja usado como uma flag porque não existe nenhum tipo booleano na linguagem. Em tal linguagem, poderíamos ter uma atribuição como:

```
timeOut = 1
```

O significado dessa sentença não é claro. Em uma linguagem que inclui tipos booleanos, teríamos:

```
timeOut = true
```

O significado dessa sentença é perfeitamente claro.

1.3.1.4 Projeto da sintaxe

A sintaxe, ou forma, dos elementos de uma linguagem tem um efeito significativo na legibilidade dos programas. A seguir, estão alguns exemplos de escolhas de projeto sintáticas que afetam a legibilidade:

- *Palavras especiais.* A aparência de um programa e sua legibilidade são muito influenciadas pela forma das palavras especiais de uma linguagem (por exemplo, **`while`**, **`class`** e **`for`**). O método para formar sentenças compostas, ou grupos de sentenças, é especialmente importante, principalmente em construções de controle. Algumas linguagens têm usado pares casados de palavras especiais ou de símbolos

para formar grupos. C e seus descendentes usam chaves para especificar sentenças compostas. Todas essas linguagens têm diminuído a legibilidade, porque os grupos de sentenças são sempre terminados da mesma forma, o que torna difícil determinar qual grupo está sendo finalizado quando aparece um **`end`** ou uma chave de fechamento. Fortran 95 e Ada (ISO/IEC, 2014) tornaram isso claro ao usar uma sintaxe distinta para cada tipo de grupo de sentenças. Por exemplo, Ada utiliza **`end if`** para terminar uma construção de seleção e **`end loop`** para terminar uma construção de repetição. Esse é um exemplo do conflito entre a simplicidade resultante ao serem usadas menos palavras reservadas, como em Java, e a maior legibilidade que pode resultar do uso de mais palavras reservadas, como em Ada.

Outra questão importante é se as palavras especiais de uma linguagem podem ser usadas como nomes de variáveis de programas. Se puderem, os programas resultantes podem ser bastante confusos. Por exemplo, em Fortran 95, palavras especiais como `Do` e `End` são nomes válidos de variáveis; logo, a ocorrência dessas palavras em um programa pode ou não conotar algo especial.

- *Forma e significado.* Projetar sentenças de maneira que sua aparência ao menos indique parcialmente seu propósito melhora a legibilidade. A semântica, ou significado, deve advir diretamente da sintaxe, ou da forma. Em alguns casos, esse princípio é violado por duas construções de uma mesma linguagem, idênticas ou similares na aparência, mas com significados diferentes, dependendo talvez do contexto. Em C, por exemplo, o significado da palavra reservada **`static`** depende do contexto no qual ela aparece. Se for usada na definição de uma variável dentro de uma função, significa que a variável é criada em tempo de compilação. Se for usada na definição de uma variável fora de todas as funções, significa que a variável é visível apenas no arquivo no qual sua definição aparece; ou seja, não é exportada desse arquivo.

  Uma das principais reclamações em relação aos comandos de shell do UNIX (Robbins, 2005) é que sua aparência nem sempre sugere sua funcionalidade. Por exemplo, o significado do comando UNIX `grep` pode ser decifrado apenas com conhecimento prévio, ou talvez com certa esperteza e familiaridade com o editor UNIX `ed`. A aparência do `grep` não tem conotação alguma para iniciantes no UNIX. (No `ed`, o comando */expressão_regular/* busca uma subcadeia correspondente à expressão regular. Preceder isso com `g` o faz ser um comando global, especificando que o escopo da busca é o arquivo inteiro que está sendo editado. Seguir o comando com `p` especifica que as linhas contendo a subcadeia correspondente devem ser impressas. Logo `g`*/expressão_regular/*`p`, que pode ser abreviado como `grep`, imprime todas as linhas de um arquivo que contenham subcadeias que correspondam à expressão regular.)

1.3.2 Facilidade de escrita

A facilidade de escrita é a medida do quão facilmente uma linguagem pode ser usada para criar programas para um domínio. A maioria das características de linguagem que afetam a legibilidade também afetam a facilidade de escrita. Isso deriva do fato

de que o processo de escrita de um programa exige que o programador releia a parte já escrita.

Como ocorre com a legibilidade, a facilidade de escrita deve ser considerada no contexto do domínio de problema alvo de uma linguagem. Não é justo comparar a facilidade de escrita de duas linguagens no contexto de determinada aplicação quando uma delas foi projetada para tal aplicação e a outra não. Por exemplo, as facilidades de escrita de Visual BASIC (VB) (Halvorson, 2013) e de C são drasticamente diferentes para criar um programa com uma interface gráfica do usuário (GUI), para o qual o VB é ideal. Suas facilidades de escrita também são bastante diferentes para a escrita de programas de sistema, como um sistema operacional, para os quais a linguagem C foi projetada.

As seções seguintes descrevem as características mais importantes que influenciam a facilidade de escrita de uma linguagem.

1.3.2.1 Simplicidade e ortogonalidade

Se uma linguagem tem um grande número de construções, alguns programadores não conhecerão todas elas. Essa situação pode levar ao uso incorreto de alguns recursos e a uma utilização escassa de outros que podem ser mais elegantes ou mais eficientes (ou ambos) que os usados. Pode até mesmo ser possível, conforme destacado por Hoare (1973), usar recursos desconhecidos acidentalmente, com resultados inesperados. Logo, é muito melhor ter um número menor de construções primitivas e um conjunto de regras consistente para combiná-las (isto é, ortogonalidade) do que muitas construções primitivas. Um programador pode projetar uma solução para um problema complexo após aprender apenas um conjunto simples de construções primitivas.

Por outro lado, ortogonalidade demais pode prejudicar a facilidade de escrita. Erros em programas podem passar despercebidos quando praticamente quaisquer combinações de primitivas são válidas. Isso pode levar a certos absurdos no código que não podem ser descobertos pelo compilador.

1.3.2.2 Expressividade

A expressividade em uma linguagem pode se referir a diversas características. Em uma linguagem como APL (Gilman e Rose, 1983), expressividade significa a existência de operadores muito poderosos que permitem muitas computações com um programa muito pequeno. Em geral, uma linguagem expressiva especifica computações de uma forma conveniente, em vez de deselegante. Por exemplo, em C, a notação `count++` é mais conveniente e menor que `count = count + 1`. Além disso, o operador booleano **`and then`** em Ada é uma maneira conveniente de especificar avaliação em curto-circuito de uma expressão booleana. A inclusão da sentença **`for`** em Java torna a escrita de laços de contagem mais fácil do que com o uso do **`while`**, também possível. Todas essas construções aumentam a facilidade de escrita de uma linguagem.

1.3.3 Confiabilidade

Diz-se que um programa é confiável quando está de acordo com suas especificações em todas as condições. As subseções a seguir descrevem diversos recursos de linguagens que têm um efeito significativo na confiabilidade dos programas em uma linguagem.

1.3.3.1 Verificação de tipos

A **verificação de tipos** é a execução de testes para detectar erros de tipos em um programa, tanto por parte do compilador quanto durante a execução de um programa. Ela é um fator importante na confiabilidade de uma linguagem. Como a verificação de tipos em tempo de execução é dispendiosa, a verificação em tempo de compilação é mais desejável. Além disso, quanto mais cedo os erros nos programas forem detectados, mais barato será fazer todos os reparos necessários. O projeto de Java requer verificações dos tipos de praticamente todas as variáveis e expressões em tempo de compilação. Isso praticamente elimina erros de tipos em tempo de execução em programas Java. Tipos e verificação de tipos são assuntos discutidos em profundidade no Capítulo 6.

Um exemplo de como a falha em verificar tipos, tanto em tempo de compilação quanto em tempo de execução, tem levado a incontáveis erros de programa é o uso de parâmetros de subprogramas na linguagem C original (Kernighan e Ritchie, 1978). Nessa linguagem, o tipo de um parâmetro em uma chamada à função não era verificado para determinar se seu tipo combinava com o parâmetro formal correspondente na função. Uma variável do tipo `int` poderia ser usada como parâmetro em uma chamada a uma função que esperava um tipo `float` como parâmetro formal, e nem o compilador nem o sistema de tempo de execução detectariam a inconsistência. Por exemplo, como a sequência de bits que representa o inteiro 23 é essencialmente não relacionada com a sequência de bits que representa o 23 de ponto flutuante, se um 23 inteiro fosse enviado a uma função que espera um parâmetro de ponto flutuante, quaisquer usos desse parâmetro na função produziriam resultados sem sentido. Além disso, tais problemas são difíceis de diagnosticar.[3] A versão atual de C eliminou esse problema ao exigir que todos os parâmetros sejam verificados em relação aos seus tipos. Subprogramas e técnicas de passagem de parâmetros são discutidos no Capítulo 9.

1.3.3.2 Tratamento de exceções

A habilidade de um programa de interceptar erros em tempo de execução (além de outras condições não usuais detectáveis pelo programa), tomar medidas corretivas e então continuar melhora a confiabilidade. Tal facilidade é chamada de **tratamento de exceções**. Ada, C++, Java e C# contêm recursos extensivos para tratamento de exceções, mas tais funções são praticamente inexistentes em algumas linguagens amplamente utilizadas, por exemplo C. O tratamento de exceções é discutido no Capítulo 14.

1.3.3.3 Apelidos

Em uma definição bastante informal, **apelidos** são utilizados quando é possível ter um ou mais nomes em um programa para acessar a mesma célula de memória. Atualmente, é geralmente aceito que o uso de apelidos é um recurso perigoso em uma linguagem de programação. A maioria das linguagens permite algum tipo de apelido – por exemplo, dois ponteiros (ou referências) configurados para apontar para a mesma variável, o que é possível na maioria das linguagens. O programador deve sempre lembrar que trocar

[3] Em resposta a isso e a outros problemas similares, os sistemas UNIX incluem um programa utilitário chamado lint, que examina os programas em C para verificar tais problemas.

o valor apontado por um dos dois ponteiros modifica o valor referenciado pelo outro. Alguns tipos de uso de apelidos, conforme descrito nos Capítulos 5 e 9, podem ser proibidos pelo projeto de uma linguagem.

Em algumas linguagens, apelidos são usados para resolver deficiências nos recursos de abstração de dados. Outras restringem o uso de apelidos para aumentar sua confiabilidade.

1.3.3.4 Legibilidade e facilidade de escrita

Tanto a legibilidade quanto a facilidade de escrita influenciam a confiabilidade. Um programa escrito em uma linguagem que não suporta maneiras naturais de expressar os algoritmos exigidos necessariamente usará estratégias artificiais. É menos provável que estratégias artificiais estejam corretas para todas as situações possíveis. Quanto mais fácil é escrever um programa, maior a probabilidade de ele estar correto.

A legibilidade afeta a confiabilidade tanto nas fases de escrita quanto nas de manutenção do ciclo de vida. Programas difíceis de ler são também difíceis de escrever e modificar.

1.3.4 Custo

O custo total de uma linguagem de programação é uma função de muitas de suas características.

Primeiro, existe o custo de treinar programadores para usar a linguagem, que é uma função da simplicidade, da ortogonalidade da linguagem e da experiência dos programadores. Apesar de linguagens mais poderosas não serem necessariamente mais difíceis de aprender, normalmente elas o são.

Segundo, há o custo de escrever programas na linguagem. Essa é uma função da facilidade de escrita da linguagem, a qual depende da proximidade com o propósito da aplicação em particular. Os esforços originais de projetar e implementar linguagens de alto nível foram dirigidos pelo desejo de diminuir os custos da criação de software.

Tanto o custo de treinar programadores quanto o custo de escrever programas em uma linguagem podem ser reduzidos significativamente em um bom ambiente de programação. Ambientes de programação são discutidos na Seção 1.8.

Terceiro, há o custo de compilar programas na linguagem. Um grande impeditivo para os primeiros usos de Ada era o custo proibitivamente alto da execução dos compiladores da primeira geração. Esse problema foi reduzido com a aparição de compiladores Ada aprimorados.

Quarto, o custo de executar programas escritos em uma linguagem é amplamente influenciado pelo projeto dela. Uma linguagem que exige muitas verificações de tipos em tempo de execução impedirá uma execução rápida de código, independentemente da qualidade do compilador. Apesar de a eficiência de execução ser a principal preocupação no projeto das primeiras linguagens, atualmente ela é considerada menos importante.

Uma escolha simples pode ser feita entre o custo de compilação e a velocidade de execução do código compilado. **Otimização** é o nome dado à coleção de técnicas que os compiladores podem usar para diminuir o tamanho e/ou aumentar a velocidade do código que produzem. Se há pouca ou nenhuma otimização, a compilação pode ser

muito mais rápida do que se for feito um esforço significativo para produzir código otimizado. A escolha entre as duas alternativas é influenciada pelo ambiente no qual o compilador será usado. Em um laboratório para estudantes iniciantes em programação – que geralmente compilam seus programas diversas vezes durante o desenvolvimento, mas usam pouco código em tempo de execução (seus programas são pequenos e precisam ser executados corretamente apenas uma vez) –, pouca ou nenhuma otimização deve ser feita. Em um ambiente de produção, onde os programas compilados são executados muitas vezes após o desenvolvimento, é melhor pagar o custo extra de otimizar o código.

O quinto fator é o custo do sistema de implementação da linguagem. Um dos fatores que explica a rápida aceitação de Java são os sistemas de compilação/interpretação gratuitos que se tornaram disponíveis logo após seu projeto ter sido disponibilizado ao público. Uma linguagem cujo sistema de implementação é caro ou pode ser executado apenas em plataformas de hardware caras terá uma chance muito menor de se tornar amplamente usada. Por exemplo, o alto custo da primeira geração de compiladores Ada ajudou a impedir que essa linguagem se tornasse popular em seus primeiros anos.

O sexto fator é o custo de uma confiabilidade baixa. Se um aplicativo de software falha em um sistema crítico, como uma usina nuclear ou uma máquina de raio-X para uso médico, o custo pode ser muito alto. As falhas de sistemas não críticos também podem ser muito caras em termos de futuros negócios perdidos ou processos decorrentes de sistemas de software defeituosos.

A consideração final é o custo da manutenção de programas, que inclui tanto as correções quanto as modificações para adicionar funcionalidades. O custo da manutenção de software depende de várias características da linguagem, principalmente da legibilidade. Como a manutenção é feita em geral por indivíduos que não são os autores originais do programa, uma legibilidade ruim pode tornar a tarefa extremamente desafiadora.

A importância da facilidade de manutenção de software não pode ser subestimada. Estima-se que, para grandes sistemas de software com tempos de vida relativamente longos, os custos de manutenção podem atingir o dobro ou o quádruplo dos custos de desenvolvimento (Sommerville, 2010).

De todos os fatores que contribuem para os custos de uma linguagem, três são os mais importantes: desenvolvimento de programas, manutenção e confiabilidade. Como esses fatores são funções da facilidade de escrita e da legibilidade, esses dois critérios de avaliação são, por sua vez, os mais importantes.

É claro, outros critérios podem ser usados para avaliar linguagens de programação. Um exemplo é a **portabilidade** – a facilidade com a qual os programas podem ser movidos de uma implementação para outra. A portabilidade é, na maioria das vezes, muito influenciada pelo grau de padronização da linguagem. Algumas linguagens não são padronizadas, o que dificulta que os programas nelas escritos sejam mudados de uma implementação para outra. Esse problema é atenuado, em alguns casos, pelo fato de as implementações para algumas linguagens terem agora fontes únicas. A padronização é um processo difícil e demorado. Um comitê começou a trabalhar em uma versão padrão de C++ em 1989. Ela foi aprovada em 1998.

A **generalidade** (a aplicabilidade a uma grande variedade de aplicações) e o fato de uma linguagem ser **bem definida** (em relação à completude e à precisão do documento oficial que define a linguagem) são outros dois critérios.

A maioria dos critérios, principalmente a legibilidade, a facilidade de escrita e a confiabilidade, não é precisamente definida nem precisamente mensurável. Independentemente disso, são conceitos úteis e fornecem ideias valiosas para o projeto e para a avaliação de linguagens de programação.

Uma observação final sobre critérios de avaliação: os critérios de projeto de linguagem têm diferentes pesos quando vistos de diferentes perspectivas. Implementadores de linguagens estão preocupados principalmente com a dificuldade de implementar as construções e os recursos da linguagem. Os usuários estão preocupados primeiramente com a facilidade de escrita e depois com a legibilidade. Os projetistas são propensos a enfatizar a elegância e a capacidade de atrair um grande número de usuários. Essas características geralmente são conflitantes.

1.4 INFLUÊNCIAS NO PROJETO DE LINGUAGENS

Além dos fatores descritos na Seção 1.3, outros também influenciam o projeto básico das linguagens de programação. Os mais importantes são a arquitetura de computadores e as metodologias de projeto de programas.

1.4.1 Arquitetura de computadores

A arquitetura básica dos computadores tem um efeito profundo no projeto de linguagens. A maioria das linguagens populares dos últimos 60 anos tem sido projetada considerando a principal arquitetura de computadores, chamada de **arquitetura de von Neumann**, cujo nome homenageia um de seus criadores, John von Neumann (pronuncia-se "fon Noiman"). Elas são chamadas de linguagens **imperativas**. Em um computador von Neumann, tanto os dados quanto os programas são armazenados na mesma memória. A unidade central de processamento (CPU), que executa instruções, é separada da memória. Logo, instruções e dados devem ser transmitidos da memória para a CPU. Resultados de operações na CPU devem ser retornados para a memória. Praticamente todos os computadores digitais construídos desde os anos 1940 são baseados nessa arquitetura. A estrutura geral de um computador von Neumann é mostrada na Figura 1.1.

Devido à arquitetura de von Neumann, os recursos centrais das linguagens imperativas são as variáveis, que modelam as células de memória; as sentenças de atribuição, baseadas na operação de envio de dados e instruções (*piping*); e a forma iterativa de repetição nessa arquitetura. Os operandos em expressões são enviados da memória para a CPU e o resultado da avaliação da expressão é enviado de volta à célula de memória, representada pelo lado esquerdo da atribuição. A iteração é rápida em computadores von Neumann porque as instruções são armazenadas em células adjacentes de memória, e repetir a execução de uma seção de código exige apenas uma instrução de desvio. Essa eficiência desencoraja o uso de recursão para repetição, embora a recursão seja, às vezes, mais natural.

A execução de um programa em código de máquina em uma arquitetura de computadores von Neumann ocorre em um processo chamado de **ciclo de obtenção e execução**. Conforme mencionado, os programas residem na memória, mas são executados na

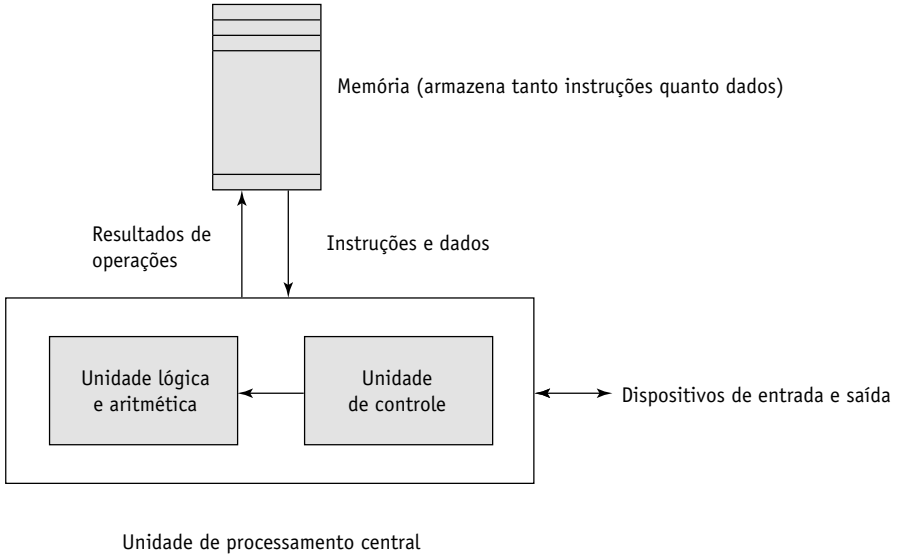


FIGURA 1.1
A arquitetura de computadores von Neumann.

CPU. Cada instrução a ser executada deve ser movida da memória para o processador. O endereço da próxima instrução a ser executada é mantido em um registrador chamado de **contador de programa**. Esse ciclo pode ser descrito de maneira simples pelo algoritmo:

```
inicialize o contador de programa
repita para sempre
    obtenha a instrução apontada pelo contador de programa
    incremente o contador de programa a fim de que aponte para a próxima instrução
    decodifique a instrução
    execute a instrução
fim repita
```

O passo "decodifique a instrução" no algoritmo significa que a instrução é examinada para determinar que ação ela especifica. A execução de um programa termina quando uma instrução de parada é encontrada, apesar de, em um computador real, uma instrução de parada raramente ser executada. Em vez disso, o controle é transferido do sistema operacional a um programa de usuário para sua execução e retorna para o sistema operacional quando a execução do programa de usuário termina. Em um sistema de computação no qual mais de um programa de usuário pode estar na memória em dado momento, esse processo é muito mais complexo.

Conforme mencionado, uma linguagem funcional, ou aplicativa, é aquela na qual a principal forma de computação é a aplicação de funções a parâmetros fornecidos. Em uma linguagem funcional, a programação pode ser feita sem os tipos de variáveis usados nas linguagens imperativas, sem sentenças de atribuição e sem iteração. Apesar de muitos cientistas da computação terem explicado a infinidade de vantagens oriundas das linguagens funcionais, como Scheme, é improvável que elas substituam as linguagens

imperativas, até que um computador que não use a arquitetura de von Neumann seja projetado e permita a execução eficiente de programas em linguagens funcionais. Entre aqueles que já reclamaram desse fato, o mais eloquente é John Backus (1978), o principal projetista da versão original de Fortran.

Apesar de a estrutura das linguagens de programação imperativas ser modelada em uma arquitetura de máquina, em vez de modelada de acordo com as habilidades e inclinações dos usuários das linguagens de programação, alguns acreditam que o uso de linguagens imperativas é mais natural que o uso de uma funcional. Logo, essas pessoas acreditam que, mesmo se os programas funcionais fossem tão eficientes como os programas imperativos, o uso das linguagens de programação imperativas seria dominante.

1.4.2 Metodologias de projeto de programas

O final dos anos 1960 e o início dos anos 1970 trouxeram uma análise intensa, iniciada em grande parte pelo movimento da programação estruturada, tanto do processo de desenvolvimento de software quanto do projeto de linguagens de programação.

Uma razão importante para essa pesquisa foi a mudança no custo maior da computação, do hardware para o software, pois os custos de hardware declinavam e os de programação aumentavam. Aumentos na produtividade dos programadores eram relativamente pequenos. Além disso, os computadores resolviam problemas maiores e mais complexos. Em vez de simplesmente resolver conjuntos de equações para simular rotas de satélites, como no início dos anos 1960, os programas eram escritos para executar tarefas extensas e complexas, como controlar grandes refinarias de petróleo e fornecer sistemas de reservas de passagens aéreas em âmbito mundial.

As novas metodologias de desenvolvimento de software que emergiram como um resultado da pesquisa nos anos 1970 foram chamadas de projeto descendente (top-down) e de refinamento passo a passo. As principais deficiências descobertas nas linguagens de programação eram a incompletude da verificação de tipos e a inadequação das sentenças de controle (que exigiam uso intenso de desvios incondicionais, também conhecidos como gotos).

No final dos anos 1970, iniciou-se uma mudança nas metodologias de projeto de programas, da orientação aos procedimentos para uma orientação aos dados. Os métodos orientados a dados enfatizam a modelagem de dados, concentrando-se no uso de tipos abstratos para solucionar problemas.

Para que as abstrações de dados sejam usadas efetivamente no projeto de sistemas de software, elas devem ser suportadas pelas linguagens utilizadas para a implementação. A primeira linguagem a fornecer suporte limitado para a abstração de dados foi SIMULA 67 (Birtwistle et al., 1973), apesar de não ter se tornado popular por causa disso. As vantagens da abstração de dados não foram amplamente reconhecidas até o início dos anos 1970. Entretanto, a maioria das linguagens projetadas desde o final daquela década oferece suporte à abstração de dados, discutida em detalhes no Capítulo 11.

O último grande passo na evolução do desenvolvimento de software orientado a dados, que começou no início dos anos 1980, é o projeto orientado a objetos. As metodologias orientadas a objetos começam com a abstração de dados, que encapsula o processamento com os objetos de dados, controla o acesso aos dados e também adiciona mecanismos de herança e vinculação dinâmica de métodos. A herança é um conceito poderoso que melhora o potencial reúso de software existente, fornecendo a possibili-

dade de melhorias significativas na produtividade, no contexto de desenvolvimento de software. Esse é um fator importante no aumento da popularidade das linguagens orientadas a objetos. A vinculação dinâmica de métodos (em tempo de execução) permite um uso mais flexível da herança.

A programação orientada a objetos se desenvolveu com uma linguagem que oferecia suporte para seus conceitos: Smalltalk (Goldberg e Robson, 1989). Apesar de Smalltalk nunca ter se tornado amplamente utilizada, como muitas outras linguagens, o suporte à programação orientada a objetos agora faz parte da maioria das linguagens imperativas populares, incluindo Java, C++ e C#. Conceitos de orientação a objetos também encontraram seu caminho em linguagens funcionais, como em CLOS (Bobrow et al., 1988) e F# (Syme et al. 2010), assim como na programação lógica em Prolog++ (Moss, 1994). O suporte à programação orientada a objetos é discutido em detalhes no Capítulo 12.

A programação orientada a procedimentos é, de certa forma, o oposto da orientada a dados. Apesar de os métodos orientados a dados dominarem o desenvolvimento de software atualmente, os métodos orientados a procedimentos não foram abandonados. Pelo contrário, nos últimos anos foram feitas muitas pesquisas sobre programação orientada a procedimentos, especialmente na área da concorrência. Esses esforços de pesquisa trouxeram a necessidade de recursos de linguagem para criar e controlar unidades de programas concorrentes. Java e C# incluem esses recursos. A concorrência está discutida em detalhes no Capítulo 13.

Todos esses passos evolutivos em metodologias de desenvolvimento de software levaram a novas construções de linguagem para suportá-los.

1.5 CATEGORIAS DE LINGUAGENS

As linguagens de programação normalmente são divididas em quatro categorias: imperativas, funcionais, lógicas e orientadas a objetos. Entretanto, não consideramos que linguagens que suportam a orientação a objetos formem uma categoria separada. Aqui, descrevemos como as linguagens mais populares que suportam a orientação a objetos cresceram a partir de linguagens imperativas. Apesar de o paradigma de desenvolvimento de software orientado a objetos diferir significativamente do paradigma orientado a procedimentos usado normalmente nas linguagens imperativas, as extensões a uma linguagem imperativa, necessárias para dar suporte à programação orientada a objetos, não são intensivas. Por exemplo, as expressões, sentenças de atribuição e sentenças de controle de C e Java são praticamente idênticas. (Por outro lado, os vetores, os subprogramas e a semântica de Java são muito diferentes dos de C.) O mesmo pode ser dito para linguagens funcionais que oferecem suporte à programação orientada a objetos.

Alguns autores se referem às linguagens de *scripting* como uma categoria separada de linguagens de programação. Entretanto, linguagens nessa categoria são mais unidas entre si por seu método de implementação, interpretação parcial ou completa, do que por um projeto de linguagem comum. As linguagens de *scripting*, entre elas Perl, JavaScript (Flanagan, 2011) e Ruby, são imperativas em todos os sentidos.

Uma linguagem de programação lógica é um exemplo de uma baseada em regras. Em uma linguagem imperativa, um algoritmo é especificado em muitos detalhes, e deve ser incluída a ordem de execução específica das instruções ou sentenças. Em uma lin-

guagem baseada em regras, entretanto, estas são especificadas sem uma ordem específica, e o sistema de implementação da linguagem deve escolher uma ordem na qual elas são usadas para produzir os resultados desejados. Essa abordagem para o desenvolvimento de software é radicalmente diferente daquelas usadas nas outras duas categorias de linguagens e requer um tipo completamente diferente de linguagem. Prolog, a linguagem de programação lógica mais usada, e a programação lógica propriamente dita são discutidas no Capítulo 16.

Nos últimos anos, surgiu uma nova categoria: as linguagens de marcação/programação híbridas. As linguagens de marcação não são de programação. Por exemplo, HTML, a linguagem de marcação mais utilizada, especifica a disposição da informação em documentos Web. Entretanto, alguns recursos de programação foram colocados em extensões de HTML e XML. Entre esses estão a biblioteca padrão de JSP, chamada de Java Server Pages Standard Tag Library (JSTL), e a linguagem chamada eXtensible Stylesheet Language Transformations (XSLT). Ambas são apresentadas brevemente no Capítulo 2. Elas não podem ser comparadas a nenhuma linguagem de programação completa e, dessa forma, não serão discutidas depois desse capítulo.

1.6 *TRADE-OFFS* NO PROJETO DE LINGUAGENS

Os critérios de avaliação de linguagens de programação descritos na Seção 1.3 fornecem um modelo para o projeto de linguagens. Infelizmente, esse modelo é contraditório. Em seu brilhante artigo sobre o projeto de linguagens, Hoare (1973) afirma que "existem tantos critérios importantes, mas conflitantes, que sua harmonização e satisfação estão entre as principais tarefas de engenharia".

Dois critérios conflitantes são a confiabilidade e o custo de execução. Por exemplo, a linguagem Java exige que todas as referências aos elementos de um vetor sejam verificadas para garantir que os índices estejam em suas faixas válidas. Esse passo aumenta muito o custo de execução de programas Java que contenham um grande número de referências a elementos de vetores. C não exige a verificação da faixa de índices – dessa forma, os programas em C executam mais rápido que programas semanticamente equivalentes em Java, apesar de estes serem mais confiáveis. Os projetistas de Java trocaram eficiência de execução por confiabilidade.

Como outro exemplo de critérios conflitantes que levam diretamente a *trade-offs* no projeto, considere o caso da APL. Ela contém um poderoso conjunto de operadores para operandos de vetor. Dado o grande número de operadores, um número significativo de novos símbolos teve de ser incluído em APL para representar esses operadores. Além disso, muitos operadores APL podem ser usados em uma expressão longa e complexa. Um resultado desse alto grau de expressividade é que, para aplicações envolvendo muitas operações de vetores, APL tem uma facilidade de escrita muito grande. De fato, uma enorme quantidade de computação pode ser especificada em um programa muito pequeno. Outro resultado é que os programas APL têm péssima legibilidade. Uma expressão compacta e concisa tem certa beleza matemática, mas é difícil para qualquer um, exceto o programador, entendê-la. O renomado autor Daniel McCraken (1970) afirmou em certa ocasião que levou quatro horas para ler e entender um programa APL de quatro linhas. O projetista trocou a legibilidade pela facilidade de escrita.

O conflito entre a facilidade de escrita e a legibilidade é comum no projeto de linguagens. Os ponteiros de C++ podem ser manipulados de diversas maneiras, o que possibilita um endereçamento de dados altamente flexível. Devido aos potenciais problemas de confiabilidade com o uso de ponteiros, eles não foram incluídos em Java.

Exemplos de conflitos entre critérios de projeto e de avaliação de linguagens são abundantes; alguns sutis, outros óbvios. Logo, é claro que a tarefa de escolher construções e recursos ao projetar uma linguagem de programação exige muito comprometimento e *trade-offs*.

1.7 MÉTODOS DE IMPLEMENTAÇÃO

Conforme descrito na Seção 1.4.1, dois dos componentes primários de um computador são sua memória interna e seu processador. A memória interna armazena programas e dados. O processador é uma coleção de circuitos que fornece a materialização de um conjunto de operações primitivas, ou instruções de máquina, como operações lógicas e aritméticas. Na maioria dos computadores, algumas dessas instruções, por vezes chamadas de macroinstruções, são implementadas com um conjunto de microinstruções, definidas em um nível mais baixo ainda. Como as microinstruções nunca são vistas pelo software, elas não serão discutidas mais a fundo aqui.

A linguagem de máquina do computador é seu conjunto de instruções. Na falta de outro software de suporte, sua própria linguagem de máquina é a única que a maioria dos computadores e que seu hardware "entendem". Teoricamente, um computador poderia ser projetado e construído com uma linguagem de alto nível como sua linguagem de máquina, mas isso seria muito complexo e caro. Além disso, seria altamente inflexível, pois seria difícil (mas não impossível) usá-lo com outras linguagens de alto nível. A escolha de projeto de máquina mais prática implementa, em hardware, uma linguagem de muito baixo nível que fornece as operações primitivas mais necessárias e exige sistemas de software para criar uma interface para programas em linguagens de alto nível.

Um sistema de implementação de linguagem não pode ser o único aplicativo de software em um computador. Também é necessária uma grande coleção de programas, chamada de sistema operacional, a qual fornece primitivas de nível mais alto do que aquelas fornecidas pela linguagem de máquina. Essas primitivas fornecem funções para o gerenciamento de recursos do sistema, operações de entrada e saída, um sistema de gerenciamento de arquivos, editores de texto e/ou de programas e uma variedade de outras funções. Como os sistemas de implementação de linguagens precisam de muitas das facilidades do sistema operacional, eles fazem uma interface com o sistema, em vez de diretamente com o processador (em linguagem de máquina).

O sistema operacional e as implementações de linguagem são colocados em camadas superiores à interface de linguagem de máquina de um computador. Essas camadas podem ser vistas como computadores virtuais, fornecendo interfaces para o usuário em níveis mais altos. Por exemplo, um sistema operacinal e um compilador C fornecem um computador C virtual. Com outros compiladores, uma máquina pode se tornar outros tipos de computadores virtuais. A maioria dos sistemas de computação fornece diferentes computadores virtuais. Os programas de usuários formam outra camada sobre a de computadores virtuais. A visão em camadas de um computador é mostrada na Figura 1.2.

Os sistemas de implementação das primeiras linguagens de programação de alto nível, construídas no final dos anos 1950, estavam entre os sistemas de software mais complexos da época. Nos anos 1960, esforços de pesquisa intensivos foram feitos para entender e formalizar o processo de construir essas implementações de linguagem de alto nível. O maior sucesso desses esforços foi na área de análise sintática, primariamente porque essa parte do processo de implementação é uma aplicação de partes da teoria de autômatos e da de linguagens formais que eram bem entendidas.

1.7.1 Compilação

As linguagens de programação podem ser implementadas por um de três métodos gerais. Em um extremo, os programas podem ser traduzidos para linguagem de máquina, a qual pode ser executada diretamente no computador. Esse método é chamado de **implementação baseada em compilação**, com a vantagem de ter uma execução de programas muito rápida, uma vez que o processo de tradução estiver completo. A maioria das implementações de produção das linguagens, como C, COBOL e C++, é feita por meio de compiladores.

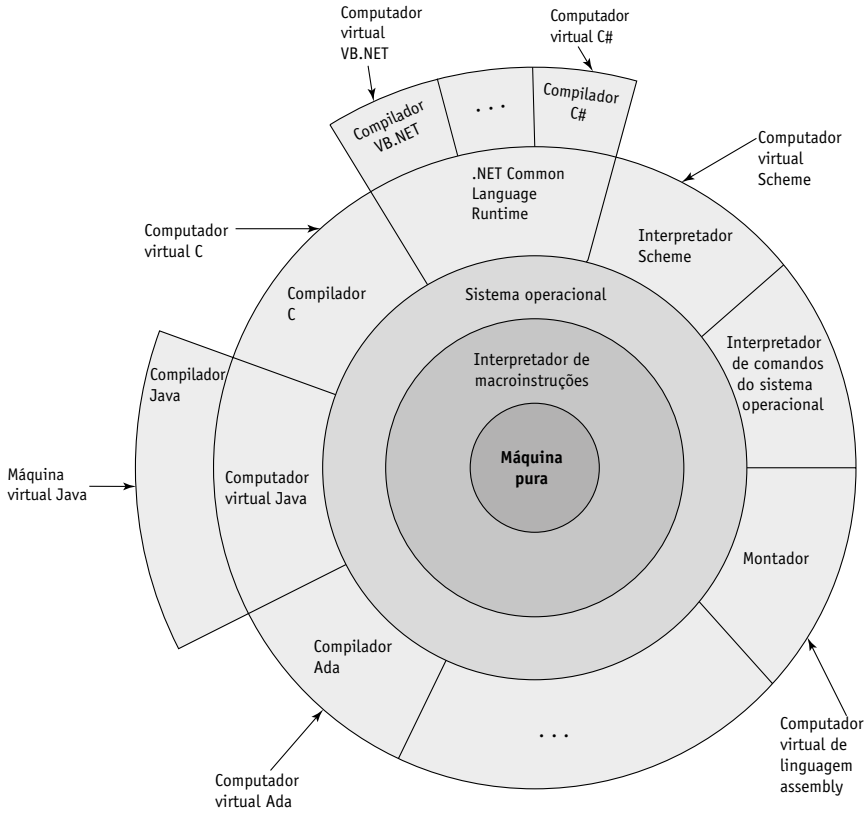


FIGURA 1.2
Interface em camadas de computadores virtuais, fornecida por um sistema de computação típico.

A linguagem que um compilador traduz é chamada de **linguagem-fonte**. O processo de compilação e a execução do programa ocorrem em fases diferentes, as mais importantes das quais são mostradas na Figura 1.3.

O analisador léxico reúne os caracteres do programa-fonte em unidades léxicas. As unidades léxicas de um programa são identificadores, palavras especiais, operadores e sinais de pontuação. O analisador léxico ignora comentários no programa-fonte, pois o compilador não tem uso para eles.

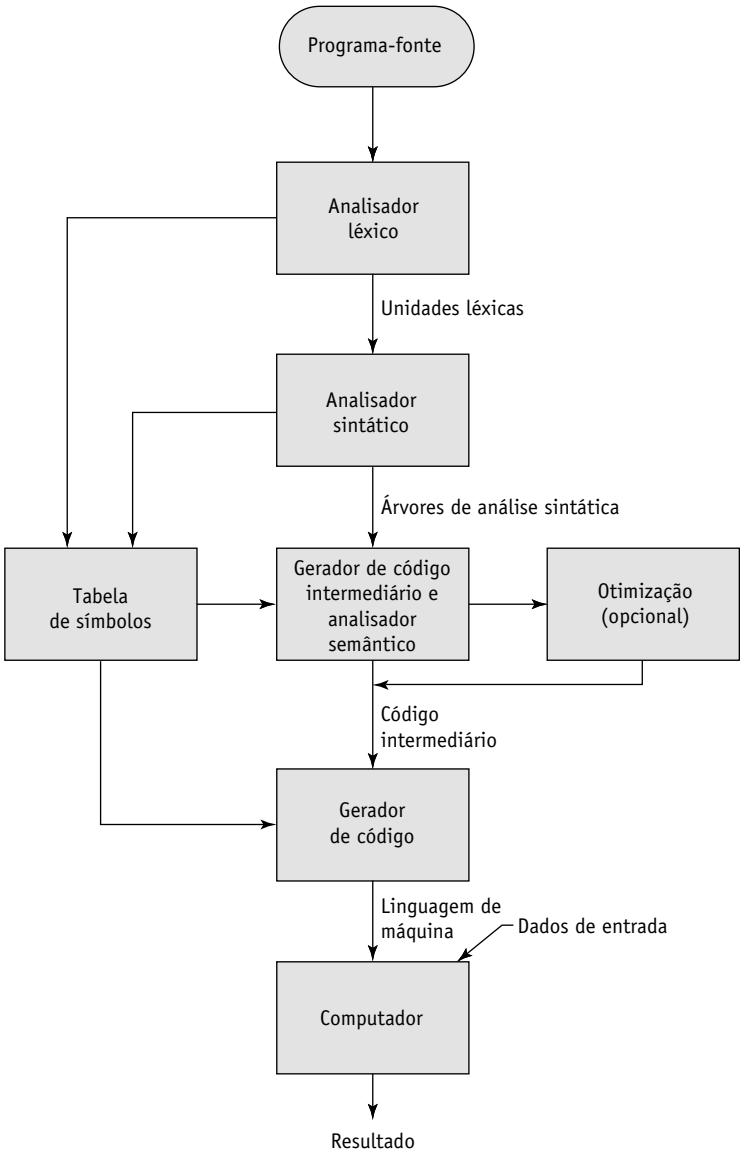


FIGURA 1.3
O processo de compilação.

O analisador sintático obtém as unidades léxicas do analisador léxico e as utiliza para construir estruturas hierárquicas chamadas de árvores de **análise sintática** (*parse trees*). Essas árvores de análise sintática representam a estrutura sintática do programa. Em muitos casos, nenhuma estrutura de árvore de análise sintática é realmente construída; em vez disso, a informação que seria necessária para construir a árvore é gerada e usada diretamente. Tanto as unidades léxicas quanto as árvores de análise sintática são discutidas mais detalhadamente no Capítulo 3. A análise léxica e a análise sintática (ou *parsing*) são discutidas no Capítulo 4.

O gerador de código intermediário produz um programa em uma linguagem diferente, em um nível intermediário entre o programa-fonte e a saída final do compilador: o programa em linguagem de máquina.[4] Linguagens intermediárias algumas vezes se parecem muito com as de montagem e, de fato, algumas vezes são linguagens de montagem propriamente ditas. Em outros casos, o código intermediário está em um nível um pouco mais alto do que uma linguagem assembly. O analisador semântico é parte integrante do gerador de código intermediário. Ele procura erros, como os de tipo, que são difíceis (senão impossíveis) de detectar durante a análise sintática.

A otimização – que melhora os programas (normalmente, em sua versão de código intermediário) tornando-os menores, mais rápidos ou ambos – é uma parte opcional da compilação. Como muitos tipos de otimização são difíceis de serem feitos em linguagem de máquina, a maioria é feita em código intermediário.

O gerador de código traduz a versão de código intermediário otimizado do programa em um programa equivalente em linguagem de máquina.

A tabela de símbolos serve como uma base de dados para o processo de compilação. O conteúdo primário na tabela de símbolos são informações de tipo e atributos de cada um dos nomes definidos pelo usuário no programa. Essa informação é colocada na tabela pelos analisadores léxico e sintático e é usada pelo analisador semântico e pelo gerador de código.

Conforme mencionado, apesar de ser possível executar a linguagem de máquina gerada por um compilador diretamente no hardware, ela quase sempre precisa ser executada com algum outro código. A maioria dos programas de usuário também necessita de programas do sistema operacional. Entre os mais comuns estão os programas para entrada e saída. O compilador constrói chamadas para os programas de sistema exigidos, quando são necessários para o programa de usuário. Antes de os programas em linguagem de máquina produzidos por um computador poderem ser executados, os programas exigidos pelo sistema operacional devem ser encontrados e ligados com o programa de usuário. A operação de ligação conecta o programa de usuário aos de sistema, colocando os endereços dos pontos de entrada dos programas de sistema nas chamadas a esses no programa de usuário. Juntos, o código de usuário e o de sistema são chamados de **módulo de carga** ou de **imagem executável**. O processo de coletar programas de sistema e ligá-los aos programas de usuário é chamado de **ligação e carga** ou apenas de **ligação**. Tal tarefa é realizada por um programa de sistema chamado de **ligador** (*linker*).

[4]Observe que as palavras programa e código muitas vezes são usadas indistintamente.

Além dos programas de sistema, os programas de usuário normalmente precisam ser ligados a outros programas de usuário previamente compilados que residem em bibliotecas. Logo, o ligador não apenas liga um programa a programas de sistema, mas também pode ligá-lo a outros programas de usuário.

A velocidade de conexão entre a memória de um computador e seu processador frequentemente determina a velocidade do computador. As instruções normalmente podem ser executadas mais rapidamente do que movidas para o processador para que possam ser executadas. Essa conexão é chamada de **gargalo de von Neumann** – esse é o principal fator limitante da velocidade de computadores com arquitetura de von Neumann. O gargalo de von Neumann tem sido um dos principais motivos para a pesquisa e o desenvolvimento de computadores paralelos.

1.7.2 Interpretação pura

A interpretação pura reside na extremidade oposta (em relação à compilação) dos métodos de implementação. Com essa abordagem, os programas são interpretados por outro, chamado interpretador, sem tradução. O interpretador age como uma simulação em software de uma máquina cujo ciclo de obtenção-execução trata de sentenças de programa de alto nível em vez de instruções de máquina. Essa simulação em software fornece uma máquina virtual para a linguagem.

A interpretação pura tem a vantagem de permitir uma fácil implementação de muitas operações de depuração em código-fonte, pois todas as mensagens de erro em tempo de execução podem referenciar unidades de código-fonte. Por exemplo, se um índice de vetor estiver fora da faixa, a mensagem de erro pode indicar facilmente a linha do código-fonte do erro e o nome do vetor. Em contrapartida, esse método tem uma séria desvantagem em relação ao tempo de execução, que é de 10 a 100 vezes mais lento do que nos sistemas compilados. A fonte primária dessa lentidão é a decodificação das sentenças em linguagem de máquina, muito mais complexas que as instruções de linguagem de máquina (embora possa haver bem menos sentenças que instruções no código de máquina equivalente). Além disso, independentemente de quantas vezes uma sentença for executada, ela deve ser decodificada a cada vez. Logo, a decodificação de sentenças, e não a conexão entre o processador e a memória, é o gargalo de um interpretador puro.

Outra desvantagem da interpretação pura é que ela normalmente exige mais espaço. Além do programa-fonte, a tabela de símbolos deve estar presente durante a interpretação. Além disso, o programa-fonte pode ser armazenado em uma forma projetada para fácil acesso e modificação, em vez de uma que ofereça tamanho mínimo.

Apesar de algumas das primeiras linguagens mais simples dos anos 1960 serem puramente interpretadas (APL, SNOBOL e Lisp), nos anos 1980 a estratégia já era raramente usada em linguagens de alto nível. Entretanto, nos últimos anos, a interpretação pura alcançou popularidade significativa com algumas linguagens de *scripting* para a Web, como JavaScript e PHP, muito usadas agora. O processo de interpretação pura é mostrado na Figura 1.4.

1.7.3 Sistemas de implementação híbridos

Alguns sistemas de implementação de linguagens são um meio-termo entre os compiladores e os interpretadores puros; eles traduzem os programas em linguagem de alto

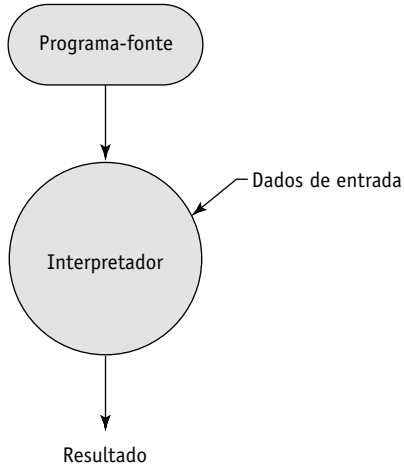


FIGURA 1.4
Interpretação pura.

nível para uma linguagem intermediária projetada para facilitar a interpretação. Esse método é mais rápido que a interpretação pura, porque as sentenças da linguagem-fonte são decodificadas apenas uma vez. Tais implementações são chamadas de **sistemas de implementação híbridos**.

O processo usado em um sistema de implementação híbrido é mostrado na Figura 1.5. Em vez de traduzir o código da linguagem intermediária na linguagem de máquina, ele interpreta o código intermediário.

Perl é implementado como um sistema híbrido. Os programas em Perl são parcialmente compilados para detectar erros antes da interpretação e para simplificar o interpretador.

As primeiras implementações de Java eram todas híbridas. Seu formato intermediário, chamado de **`byte code`**, fornece portabilidade para qualquer máquina que tenha um interpretador de bytecodes e um sistema de tempo de execução associado. Juntos, eles são chamados de Máquina Virtual Java. Existem agora sistemas que traduzem bytecodes Java em código de máquina, de forma a possibilitar uma execução mais rápida.

Um sistema de implementação Just-in-Time (JIT) inicialmente traduz os programas em uma linguagem intermediária. Então, durante a execução, compila os métodos da linguagem intermediária para linguagem de máquina, quando esses são chamados. A versão em código de máquina é mantida para chamadas subsequentes. Atualmente, sistemas JIT são bastante usados para programas Java. As linguagens .NET também são todas implementadas com um sistema JIT.

Algumas vezes, um implementador pode fornecer tanto implementações compiladas quanto interpretadas para uma linguagem. Nesses casos, o interpretador é usado para desenvolver e depurar programas. Então, após um estado (relativamente) livre de erros ser alcançado, os programas são compilados para aumentar sua velocidade de execução.

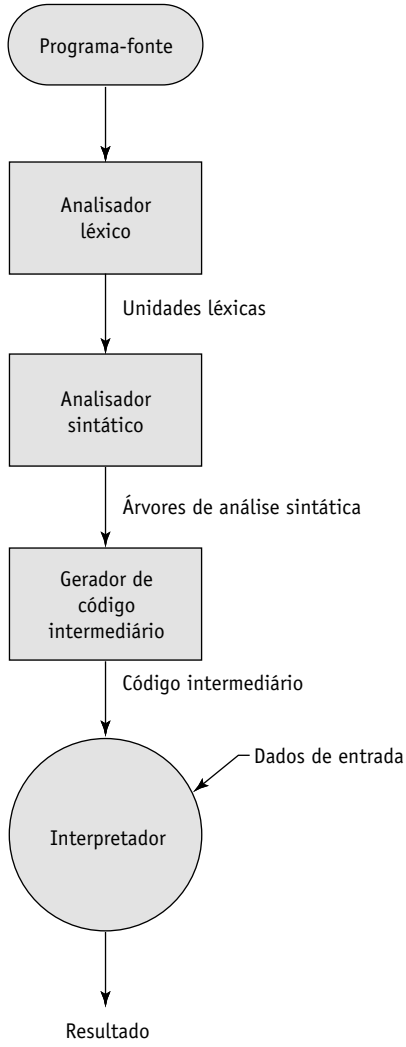


FIGURA 1.5
Sistema de implementação híbrido.

1.7.4 Pré-processadores

Um **pré-processador** é um programa que processa outro programa imediatamente antes de ele ser compilado. As instruções de pré-processador são embutidas em programas. O pré-processador é essencialmente um programa que expande macros. As instruções de pré-processador são comumente usadas para especificar que o código de outro arquivo deve ser incluído. Por exemplo, a instrução de pré-processador de C

```
#include "myLib.h"
```

o faz copiar o conteúdo de `myLib.h` no programa, na posição da instrução `#include`.

Outras instruções de pré-processador são usadas para definir símbolos para representar expressões. Por exemplo, alguém poderia usar

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

para determinar a maior das duas expressões. Por exemplo, a expressão

```
x = max(2 * y, z / 1.73);
```

seria expandida pelo pré-processador para

```
x = ((2 * y) > (z / 1.73) ? (2 * y) : (z / 1.73);
```

Note que esse é um daqueles casos nos quais os efeitos colaterais das expressões podem causar problemas. Por exemplo, se qualquer uma das expressões passadas para a macro `max` tiver efeitos colaterais – como `z++` –, podem ocorrer problemas. Como um dos dois parâmetros da expressão é avaliado duas vezes, isso pode resultar no duplo incremento de `z` pelo código produzido pela expansão da macro.

1.8 AMBIENTES DE PROGRAMAÇÃO

Um ambiente de programação é a coleção de ferramentas usadas no desenvolvimento de software. Essa coleção pode consistir em apenas um sistema de arquivos, um editor de textos, um ligador e um compilador. Ou pode incluir uma grande coleção de ferramentas integradas, cada uma acessada por meio de uma interface de usuário uniforme. Neste último caso, o desenvolvimento e a manutenção de software são enormemente melhorados. Logo, as características de uma linguagem de programação não são a única medida da capacidade de desenvolvimento de um sistema. Agora, descreveremos brevemente diversos ambientes de programação.

O UNIX é um ambiente de programação mais antigo, inicialmente distribuído em meados dos anos 1970, construído em torno de um sistema operacional de multiprogramação portável. Ele fornece uma ampla gama de ferramentas de suporte poderosas para a produção e a manutenção de software em uma variedade de linguagens. No passado, o recurso mais importante que não existia no UNIX era uma interface uniforme entre suas ferramentas. Isso o tornava mais difícil de aprender e usar. Entretanto, o UNIX é agora usado por meio de uma interface gráfica do usuário executada sobre o UNIX. Exemplos de GUIs no UNIX são o Solaris Common Desktop Environment (CDE), o GNOME e o KDE. Essas GUIs fazem com que a interface com o UNIX pareça similar à dos sistemas Windows e Macintosh.

O JBuilder é um ambiente de programação que fornece compilador, editor, depurador e sistema de arquivos integrados para desenvolvimento em Java, onde todos são acessados por meio de uma interface gráfica. O JBuilder é um sistema complexo e poderoso para criar software em Java.

O Microsoft Visual Studio .NET é um passo relativamente recente na evolução dos ambientes de desenvolvimento de software. Ele é uma grande e elaborada coleção de ferramentas de desenvolvimento, todas usadas por meio de uma interface baseada em janelas. Esse sistema pode ser usado para desenvolver software em qualquer uma das cinco linguagens .NET: C#, Visual Basic.NET, JScript (versão da Microsoft da JavaScript), F# (uma linguagem funcional) e C++/CLI.

O NetBeans é um ambiente usado primariamente para o desenvolvimento de aplicações com Java, mas também oferece suporte para JavaScript, Ruby e PHP. Tanto o Visual Studio quanto o NetBeans são mais do que ambientes de desenvolvimento – também são frameworks, que fornecem partes comuns do código da aplicação.

RESUMO

O estudo de linguagens de programação é valioso por alguns motivos importantes: ele aumenta nossa capacidade de usar diferentes construções ao escrevermos programas, nos permite escolher linguagens para projetos de forma mais inteligente e facilita o aprendizado de novas linguagens.

Os computadores são usados em uma variedade de domínios de solução de problemas. O projeto e a avaliação de determinada linguagem de programação são altamente dependentes do domínio para o qual ela será usada.

Entre os critérios mais importantes para a avaliação de linguagens estão a legibilidade, a facilidade de escrita, a confiabilidade e o custo geral. Esses critérios servirão de base para examinarmos e julgarmos os recursos das linguagens discutidas no restante do livro.

As principais influências no projeto de linguagens têm sido a arquitetura de máquina e as metodologias de projeto de software.

Projetar uma linguagem de programação é primariamente um esforço de engenharia, em que uma longa lista de *trade-offs* deve ser levada em consideração na definição de recursos, construções e capacidades.

Os principais métodos de implementar linguagens de programação são a compilação, a interpretação pura e a implementação híbrida.

Os ambientes de programação têm se tornado parte importante dos sistemas de desenvolvimento de software, nos quais a linguagem é apenas um dos componentes.

QUESTÕES DE REVISÃO

1. Por que é útil para um programador ter alguma experiência no projeto de linguagens, mesmo que ele nunca projete uma linguagem de programação?
2. Como o conhecimento de linguagens de programação pode beneficiar toda a comunidade da computação?
3. Que linguagem de programação tem dominado a computação científica nos últimos 60 anos?
4. Que linguagem de programação tem dominado as aplicações de negócios nos últimos 60 anos?
5. Que linguagem de programação tem dominado a inteligência artificial nos últimos 60 anos?
6. Em que linguagem a maior parte do UNIX é escrito?
7. Qual é a desvantagem de ter recursos demais em uma linguagem?

8. Como a sobrecarga de operador definida pelo usuário pode prejudicar a legibilidade de um programa?
9. Cite um exemplo da falta de ortogonalidade no projeto da linguagem C.
10. Qual linguagem usou a ortogonalidade como principal critério de projeto?
11. Que sentença de controle primitiva é usada para construir sentenças de controle mais complicadas em linguagens que não as têm?
12. O que significa para um programa ser confiável?
13. Por que verificar os tipos dos parâmetros de um subprograma é importante?
14. O que são apelidos?
15. O que é o tratamento de exceções?
16. Por que a legibilidade é importante para a facilidade de escrita?
17. Como o custo de compiladores para uma linguagem está relacionado ao projeto dela?
18. Qual foi a influência mais forte no projeto de linguagens de programação nos últimos 60 anos?
19. Qual é o nome da categoria de linguagens de programação cuja estrutura é ditada pela arquitetura de computadores von Neumann?
20. Quais duas deficiências das linguagens de programação foram descobertas na pesquisa sobre desenvolvimento de software dos anos 1970?
21. Quais são os três recursos fundamentais de uma linguagem orientada a objetos?
22. Qual foi a primeira linguagem a oferecer suporte aos três recursos fundamentais da programação orientada a objetos?
23. Dê um exemplo de dois critérios de projeto de linguagens que estão em conflito direto um com o outro.
24. Quais são os três métodos gerais de implementar uma linguagem de programação?
25. Qual produz uma execução de programas mais rápida: um compilador ou um interpretador puro?
26. Que papel a tabela de símbolos tem em um compilador?
27. O que faz um ligador?
28. Por que o gargalo de von Neumann é importante?
29. Quais são as vantagens de implementar uma linguagem com um interpretador puro?

PROBLEMAS

1. Você acredita que nossa capacidade de abstração é influenciada por nosso domínio de linguagens? Argumente.

2. Cite alguns dos recursos de linguagens de programação específicas que você conhece, cujo objetivo seja um mistério para você.
3. Que argumentos você pode formular a favor da ideia de uma única linguagem para todos os domínios de programação?
4. Que argumentos você pode formular contra a ideia de uma única linguagem para todos os domínios de programação?
5. Cite e explique outro critério pelo qual as linguagens podem ser julgadas (além dos discutidos neste capítulo).
6. Que sentença comum das linguagens de programação, em sua opinião, é mais prejudicial à legibilidade?
7. Java usa uma chave de fechamento para marcar o término de todas as sentenças compostas. Quais são os argumentos a favor e contra essa decisão de projeto?
8. Muitas linguagens fazem distinção entre letras minúsculas e maiúsculas em nomes definidos pelo usuário. Quais são as vantagens e desvantagens dessa decisão de projeto?
9. Explique os diferentes aspectos do custo de uma linguagem de programação.
10. Quais são os argumentos para escrever programas eficientes, mesmo sabendo que os sistemas de hardware são relativamente baratos?
11. Descreva alguns *trade-offs* de projeto entre a eficiência e a segurança em alguma linguagem que você conheça.
12. Em sua opinião, quais recursos importantes uma linguagem de programação perfeita incluiria?
13. A primeira linguagem de programação de alto nível que você aprendeu era implementada com um interpretador puro, um sistema de implementação híbrido ou um compilador? (Talvez seja necessário pesquisar isso.)
14. Descreva as vantagens e desvantagens de alguns ambientes de programação que você já tenha usado.
15. Como sentenças de declaração de tipos para variáveis simples afetam a legibilidade de uma linguagem, considerando que algumas não precisam de tais declarações?
16. Escreva uma avaliação de alguma linguagem de programação que você conheça, usando os critérios descritos neste capítulo.
17. Algumas linguagens de programação – por exemplo, Pascal – têm usado ponto e vírgula para separar sentenças, enquanto Java os utiliza para terminar sentenças. Qual desses usos, em sua opinião, é mais natural e menos provável de resultar em erros de sintaxe? Justifique sua resposta.
18. Muitas linguagens contemporâneas permitem dois tipos de comentários: um no qual os delimitadores são usados em ambas as extremidades (comentários de várias linhas) e um no qual um delimitador marca apenas o início do comentário (comentário de uma linha). Discuta as vantagens e desvantagens de cada um dos tipos de acordo com nossos critérios.

2

Evolução das principais linguagens de programação

2.1 Plankalkül de Zuse

2.2 Pseudocódigos

2.3 IBM 704 e Fortran

2.4 Programação funcional: Lisp

2.5 O primeiro passo em direção à sofisticação: ALGOL 60

2.6 Informatização de registros comerciais: COBOL

2.7 O início do compartilhamento de tempo: Basic

2.8 Tudo para todos: PL/I

2.9 Duas das primeiras linguagens dinâmicas: APL e SNOBOL

2.10 O início da abstração de dados: SIMULA 67

2.11 Projeto ortogonal: ALGOL 68

2.12 Alguns dos primeiros descendentes de ALGOLs

2.13 Programação baseada em lógica: Prolog

2.14 O maior esforço de projeto da história: Ada

2.15 Programação orientada a objetos: Smalltalk

2.16 Combinação de recursos imperativos e orientados a objetos: C++

2.17 Uma linguagem orientada a objetos baseada no paradigma imperativo: Java

2.18 Linguagens de *scripting*

2.19 A principal linguagem .NET: C#

2.20 Linguagens híbridas de marcação-programação

Este capítulo descreve o desenvolvimento de uma coleção de linguagens de programação. Ele explora o ambiente no qual cada uma delas foi projetada e foca nas contribuições da linguagem e a motivação para seu desenvolvimento. Não são incluídas descrições gerais de linguagens; em vez disso, discutimos alguns dos novos recursos introduzidos por elas. De particular interesse são os recursos que mais influenciaram linguagens subsequentes ou o campo da ciência da computação.

Este capítulo não inclui uma discussão aprofundada de nenhum recurso ou conceito de linguagem; isso é deixado para os seguintes. Explanações breves e informais de recursos serão suficientes para nossa jornada pelo desenvolvimento dessas linguagens.

O capítulo discute uma ampla variedade de linguagens e conceitos desconhecidos de muitos leitores. Esses assuntos serão discutidos em detalhes somente em capítulos posteriores. Aqueles que acharem isso um obstáculo talvez prefiram postergar a leitura deste capítulo até que o restante do livro tenha sido estudado.

A escolha de quais linguagens discutir foi subjetiva, e alguns leitores talvez notem a ausência de uma ou mais de suas linguagens prediletas. Entretanto, para não estender muito esse histórico, isso foi necessário. As escolhas foram feitas com base em nossas estimativas da importância de cada uma para o desenvolvimento das linguagens e para o mundo da computação como um todo. Também incluímos descrições breves de outras linguagens referenciadas mais adiante no livro.

A organização deste capítulo é a seguinte: as versões iniciais das linguagens geralmente são discutidas em ordem cronológica. Entretanto, versões subsequentes delas aparecem com sua versão inicial, em vez de em seções posteriores. Por exemplo, Fortran 2003 é discutida na mesma seção que Fortran I (1956). Além disso, em alguns casos, linguagens de importância secundária relacionadas a uma linguagem que tem sua própria seção aparecem naquela seção.

O capítulo inclui listagens de 14 programas de exemplo completos, cada um em uma linguagem diferente. Esses programas não são descritos neste capítulo; o objetivo é ilustrar a aparência de programas nessas linguagens. Leitores familiarizados com qualquer das linguagens imperativas comumente utilizadas devem ser capazes de ler e entender a maioria do código desses programas, exceto aqueles em Lisp, COBOL e Smalltalk. (Uma função Scheme semelhante ao exemplo com Lisp está discutida no Capítulo 15.) O mesmo problema é resolvido pelos programas em Fortran, ALGOL 60, PL/I, Basic, Pascal, C, Perl, Ada, Java, JavaScript e C#. Note que a maioria das linguagens contemporâneas nessa lista oferece suporte para vetores dinâmicos, mas, devido à simplicidade do problema de exemplo, não as utilizamos nos programas. Além disso, no programa da Fortran 95, evitamos usar os recursos que poderiam ter evitado o uso de laços completamente, em parte para manter o programa simples e legível e em parte apenas para ilustrar a estrutura básica de laços da linguagem.

A Figura 2.1 é um gráfico da genealogia das linguagens de alto nível discutidas neste capítulo.

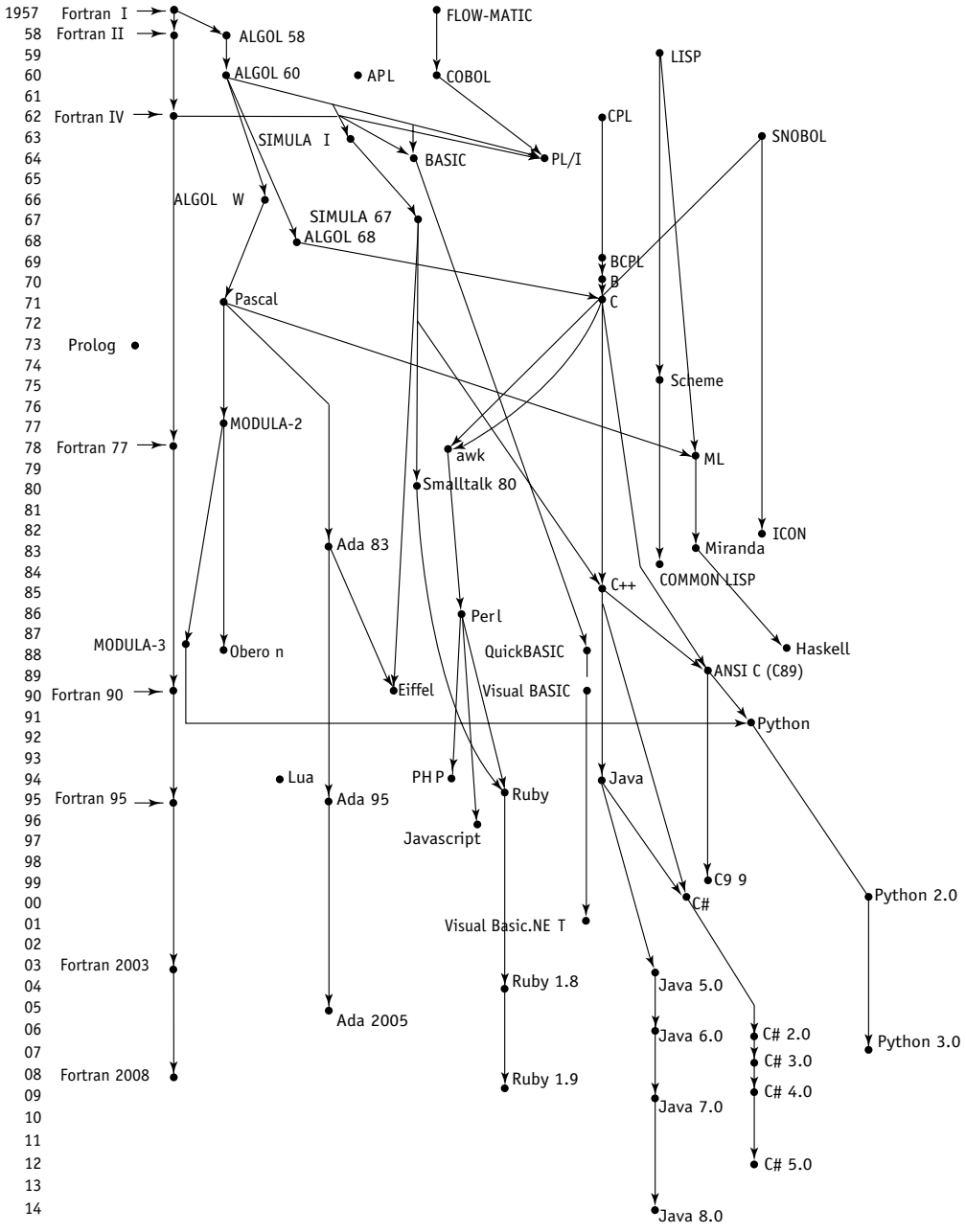


FIGURA 2.1
Genealogia das principais linguagens de programação de alto nível.

2.1 PLANKALKÜL DE ZUSE

A primeira linguagem de programação discutida neste capítulo é incomum em diversos aspectos. Primeiro, ela nunca foi implementada. Segundo, apesar de desenvolvida em 1945, sua descrição não foi publicada até 1972. Como poucas pessoas a conheciam, algumas de suas capacidades não apareceram em outras linguagens até 15 anos após seu desenvolvimento.

2.1.1 Perspectiva histórica

Entre 1936 e 1945, o cientista alemão Konrad Zuse construiu uma série de computadores complexos e sofisticados a partir de relés eletromecânicos. No início de 1945, o bombardeio aliado havia destruído todos, exceto um de seus últimos modelos, o Z4; então, ele se mudou para um vilarejo na Bavária chamado Hinterstein, e os membros de seu grupo de pesquisa se separaram.

Trabalhando sozinho, Zuse embarcou em uma jornada para desenvolver uma linguagem para expressar computações para o Z4, um projeto iniciado em 1943, como proposta de sua tese de doutorado. Ele chamou essa linguagem de Plankalkül, que significa *cálculo de programas*. Em um extenso manuscrito datado de 1945, mas não publicado até 1972 (Zuse, 1972), Zuse definiu Plankalkül e escreveu algoritmos na linguagem para resolver uma ampla variedade de problemas.

2.1.2 Visão geral da linguagem

Plankalkül era extraordinariamente completa, com alguns de seus recursos mais avançados na área de estruturas de dados. O tipo de dados mais simples em Plankalkül era o bit. Tipos numéricos inteiros e de ponto flutuante eram construídos a partir do tipo bit. O tipo ponto flutuante usava a notação de complemento de dois e o esquema de "bit oculto", atualmente usado para evitar o armazenamento do bit mais significativo da parte normalizada da fração de um valor de ponto flutuante.

Além dos tipos escalares usuais, Plankalkül incluía vetores e registros (denominados *structs* nas linguagens baseadas em C). Os registros podiam ser aninhados.

Apesar de a linguagem não ter um comando de desvio incondicional (goto) explícito, ela incluía uma sentença iterativa similar ao **`for`** de Ada. Tinha também o comando **`Fin`**, com um índice que especificava uma saída de um número de aninhamentos de iterações de um laço ou para o início de um novo ciclo iterativo. Plankalkül incluía uma sentença de seleção, mas não permitia o uso de uma cláusula do tipo senão (`else`).

Um dos recursos mais interessantes dos programas de Zuse era a inclusão de expressões matemáticas mostrando os relacionamentos atuais entre variáveis de programas. Essas expressões informavam o que deveria ser verdadeiro durante a execução nos pontos de código em que apareciam. Isso é bastante similar ao uso de asserções em Java e em semântica axiomática, discutida no Capítulo 3.

O manuscrito de Zuse continha programas de complexidade muito maior que qualquer outro escrito antes de 1945. Estavam incluídos programas para ordenar vetores de

números; testar a conectividade de um grafo dado; efetuar operações de inteiros e de ponto flutuante, incluindo a raiz quadrada; e realizar análise sintática em fórmulas lógicas que tinham parênteses e operadores em seis níveis diferentes de precedência. Talvez o mais excepcional fossem suas 49 páginas de algoritmos para jogar xadrez, um jogo no qual ele não era um especialista.

Se um cientista da computação tivesse encontrado a descrição de Zuse de Plankalkül no início dos anos 50, o único aspecto da linguagem que poderia ser um obstáculo para sua implementação, conforme a definição, seria sua notação. Cada sentença consistia em duas ou três linhas de código. A primeira linha era bastante parecida com as sentenças das linguagens atuais. A segunda, opcional, continha os índices das referências a vetores na primeira. O mesmo método para indicar índices era usado por Charles Babbage em programas para sua Máquina Analítica em meados do século XIX. A última linha de cada sentença em Plankalkül continha os nomes dos tipos para as variáveis mencionadas na primeira linha. Vista pela primeira vez, essa notação é bastante intimidadora.

A seguinte sentença de atribuição de exemplo, a qual atribui o valor da expressão `A[4] +1` para `A[5]`, ilustra essa notação. A linha rotulada `V` é para os índices e a `S` é para os tipos de dados. Nesse exemplo, `1.n` significa um inteiro de *n* bits:

```
  | A + 1 =>    A
V | 4           5
S | 1.n         1.n
```

Podemos apenas especular sobre a direção que o projeto de linguagens de programação teria tomado se o trabalho de Zuse tivesse sido amplamente conhecido em 1945 ou até mesmo em 1950. É também interessante considerar como seu trabalho teria sido diferente se ele o tivesse feito em um ambiente pacífico, cercado por outros cientistas, em vez de na Alemanha de 1945, praticamente isolado.

2.2 PSEUDOCÓDIGOS

Primeiro, note que a palavra *pseudocódigo* é usada neste capítulo com um sentido diferente de seu significado contemporâneo. Chamamos as linguagens discutidas nesta seção de pseudocódigos porque esse era o seu nome na época em que foram desenvolvidas e usadas (final dos anos 1940 e início dos anos 1950). Entretanto, elas não são pseudocódigos no sentido atual da palavra.

Os computadores que se tornaram disponíveis no final dos anos 1940 e no início dos anos 1950 eram muito menos usáveis do que os de atualmente. Além de lentas, não confiáveis, caras e com memórias extremamente pequenas, as máquinas daquela época eram difíceis de programar devido à falta de software de suporte.

Não existiam linguagens de programação de alto nível, nem mesmo linguagens de montagem, então a programação era feita em código de máquina, o que era tanto tedioso quanto passível de erros. Entre os problemas, existia o uso de códigos numéricos para especificar instruções. Por exemplo, uma instrução ADD poderia ser especificada pelo código 14, em vez de por um nome textual conotativo, mesmo se fosse composto por

apenas uma única letra. Isso tornava difícil ler os programas. Um problema mais sério era o endereçamento absoluto, que tornava as modificações de programas tediosas e passíveis de erros. Por exemplo, suponha que tenhamos um programa em linguagem de máquina armazenado na memória. Muitas das instruções se referem a outras posições dentro do programa, normalmente para referenciar dados ou indicar os alvos de instruções de desvio. Inserir uma instrução em qualquer posição do programa que não seja no final invalida a corretude de todas as instruções que referenciam endereços além do ponto de inserção, pois esses endereços devem ser incrementados para que exista espaço para a nova instrução. Para fazer a adição corretamente, todas as instruções que referenciam endereços após essa adição devem ser encontradas e modificadas. Um problema similar ocorre com a exclusão de uma instrução. Nesse caso, entretanto, as linguagens de máquina geralmente incluem uma instrução "sem operação" que pode substituir instruções excluídas, evitando o problema.

Esses são os problemas padrão das linguagens de máquina, que motivaram o desenvolvimento de montadores e de linguagens de montagem. Além disso, a maioria dos problemas de programação da época era numérica e exigia operações aritméticas de ponto flutuante e indexação de algum tipo para permitir a conveniência do uso de vetores. Nenhuma dessas capacidades, entretanto, estava incluída na arquitetura dos computadores do final dos anos 1940 e início dos anos 1950. Essas deficiências levaram ao desenvolvimento de linguagens de um nível mais alto.

2.2.1 Short Code

A primeira dessas novas linguagens, chamada de Short Code, foi desenvolvida por John Mauchly, em 1949, para o BINAC, um dos primeiros computadores eletrônicos com programas armazenados bem-sucedidos. Short Code foi posteriormente transferida para um UNIVAC I (o primeiro computador eletrônico comercial vendido nos Estados Unidos) e, por diversos anos, foi uma das principais maneiras de programar essas máquinas. Apesar de pouco ser conhecido sobre a linguagem Short Code original, já que sua descrição completa nunca foi publicada, um manual de programação para o UNIVAC I sobreviveu (Remington-Rand, 1952). É seguro supor que as duas versões eram bastante similares.

As palavras da memória do UNIVAC I tinham 72 bits, agrupados como 12 bytes de seis bits cada. A linguagem Short Code era composta de versões codificadas de expressões matemáticas que seriam avaliadas. Os códigos eram valores de pares de bytes e muitas equações podiam ser codificadas em uma palavra. Estavam incluídos os seguintes códigos de operação:

```
01 -    06 abs value  1n (n+2)nd power
02 )    07 +          2n (n+2)nd root
03 =    08 pause      4n if <= n
04 /    09 (          58 print and tab
```

As variáveis eram nomeadas com códigos de pares de bytes, assim como os locais a serem usados como constantes. Por exemplo, `X0` e `Y0` poderiam ser variáveis. A sentença

```
X0 = SQRT(ABS(Y0))
```

seria codificada em uma palavra como `00 X0 03 20 06 Y0`. O `00` inicial era usado como um espaçamento para preencher a palavra. Um fato interessante é que não existia um código para a multiplicação; ela era indicada apenas pela simples colocação de dois operandos um ao lado do outro, como na álgebra.

Os programas em Short Code não eram traduzidos para código de máquina. Em vez disso, a linguagem era implementada com um interpretador puro. Na época, esse processo era chamado de *programação automática*. Ele claramente simplificava o processo de programação, mas à custa do tempo de execução. A interpretação de Short Code era aproximadamente 50 vezes mais lenta que a do código de máquina.

2.2.2 Speedcoding

Em outros lugares eram desenvolvidos sistemas de interpretação para estender linguagens de máquina a fim de incluir operações de ponto flutuante. O sistema Speedcoding, desenvolvido por John Backus para o IBM 701, é um exemplo (Backus, 1954). O interpretador Speedcoding efetivamente convertia o 701 em uma calculadora virtual de ponto flutuante de três endereços. O sistema incluía pseudoinstruções para as quatro operações aritméticas em dados de ponto flutuante, assim como operações como a raiz quadrada, seno, arco tangente, exponenciação e logaritmo. Desvios condicionais e incondicionais e conversões de entrada e saída também faziam parte da arquitetura virtual. Para ter uma ideia das limitações de tais sistemas, considere que a memória usável restante, após carregar o interpretador, era de apenas 700 palavras e que a instrução de adição levava 4,2 milissegundos para ser executada. Em contrapartida, a linguagem Speedcoding incluía a inédita facilidade para incrementar os registradores de endereço automaticamente. Essa facilidade não apareceu em hardware até os computadores UNIVAC 1107, em 1962. Por causa desses recursos, a multiplicação de matrizes podia ser feita com 12 instruções Speedcoding. Backus afirmava que problemas que levariam duas semanas para serem programados em código de máquina poderiam ser programados em poucas horas com Speedcoding.

2.2.3 O sistema de "compilação" da UNIVAC

Entre 1951 e 1953, uma equipe liderada por Grace Hopper na UNIVAC desenvolveu uma série de sistemas de "compilação" chamados A-0, A-1 e A-2, que expandiam um pseudocódigo em subprogramas em código de máquina, da mesma maneira que as macros são expandidas em linguagem de montagem. O código-fonte do pseudocódigo para esses "compiladores" era ainda muito primitivo, apesar de já ser uma grande melhoria em relação ao código de máquina, pois reduzia os programas-fonte. Wilkes (1952), independentemente, sugeriu um processo similar.

2.2.4 Trabalhos relacionados

Outras maneiras de facilitar a tarefa de programação estavam sendo desenvolvidas mais ou menos na mesma época. Na Universidade de Cambridge, David J. Wheeler (1950) desenvolveu um método de usar blocos de endereços realocáveis para resolver, pelo menos parcialmente, o problema do endereçamento absoluto e, posteriormente, Maurice V. Wilkes (também em Cambridge) estendeu a ideia de projetar um programa em linguagem de montagem que poderia combinar sub-rotinas escolhidas e alocar armazenamento (Wilkes et al., 1951, 1957). Esse foi, de fato, um avanço importante e fundamental.

Devemos mencionar também que as linguagens de montagem, bastante diferentes dos pseudocódigos discutidos, evoluíram durante o início dos anos 1950. Entretanto, tiveram pouco impacto no projeto de linguagens de alto nível.

2.3 IBM 704 E FORTRAN

Certamente um dos maiores avanços na computação veio com a introdução do IBM 704, em 1954, em grande parte porque suas capacidades levaram ao desenvolvimento de Fortran. Não fossem a IBM com o 704 e o Fortran, haveria outra organização, com um computador similar e uma linguagem de alto nível relacionada. Entretanto, a IBM foi a primeira a ter tanto a visão quanto os recursos para bancar tais avanços.

2.3.1 Perspectiva histórica

Uma das principais razões pelas quais a lentidão dos sistemas de interpretação era tolerada no final da década de 1940 e até meados da década de 1950 era a falta de hardware de ponto flutuante nos computadores disponíveis. Todas as operações de ponto flutuante tinham de ser simuladas em software, um processo muito demorado. Como muito tempo do processador era gasto no processamento de software para ponto flutuante, a sobrecarga da interpretação e a simulação de indexação eram relativamente insignificantes. Enquanto as operações de ponto flutuante tivessem de ser feitas via software, a interpretação era uma despesa aceitável. Entretanto, muitos programadores da época nunca usaram sistemas de interpretação, preferindo a eficiência do código de linguagem máquina (ou de montagem) escrito à mão. O anúncio do sistema IBM 704, contendo tanto indexação quanto instruções de ponto flutuante em hardware, decretaram o fim da era da interpretação, ao menos para a computação científica. A inclusão de hardware de ponto flutuante removeu do esconderijo o custo da interpretação.

Apesar de Fortran levar o crédito de ser a primeira linguagem de alto nível compilada, a questão sobre quem merece o crédito por implementar a primeira linguagem desse tipo permanece aberta. Knuth e Pardo (1977) creditam a Alick E. Glennie o seu compilador de Autocode para o computador Manchester Mark I. Glennie desenvolveu o compilador no Fort Halstead, Royal Armaments Research Establishment, na Inglaterra. O compilador se tornou operacional em setembro de 1952. Entretanto, de acordo com John Backus (Wexelblat, 1981, p. 26), o Autocode de Glennie era de tão baixo nível e

orientado à máquina que não poderia ser considerado um sistema compilado. Backus dá o crédito a Laning e Zierler, do Instituto de Tecnologia de Massachusetts (MIT).

O sistema de Laning e Zierler (Laning and Zierler, 1954) foi o primeiro de tradução algébrica a ser implementado. Por algébrica, queremos dizer que o sistema traduzia expressões aritméticas, usava subprogramas codificados separadamente para computar funções transcendentais (por exemplo, seno e logaritmo) e incluía vetores. No verão de 1952, o sistema foi implementado como um protótipo experimental no computador Whirlwind do MIT, e, em uma forma mais usável, em maio de 1953. O tradutor gerava uma chamada à sub-rotina para codificar cada fórmula, ou expressão, no programa. A linguagem-fonte era fácil de ler e as instruções de máquina incluídas eram para desvios. Embora esse trabalho seja anterior ao do Fortran, ele nunca saiu do MIT.

Apesar desses trabalhos anteriores, a primeira linguagem de alto nível compilada de ampla aceitação foi Fortran. As subseções seguintes descrevem esse importante avanço.

2.3.2 Processo de projeto

Mesmo antes de o sistema 704 ser anunciado, em maio de 1954, já haviam sido iniciados os planos para Fortran. Em novembro de 1954, John Backus e seu grupo da IBM produziram um relatório intitulado "The IBM Mathematical FORmula TRANslating System: FORTRAN" (IBM, 1954). Esse documento descrevia a primeira versão de Fortran, à qual nos referimos como Fortran 0, antes de sua implementação. O documento também afirmava que forneceria a eficiência de programas codificados manualmente, com a facilidade de programação dos sistemas de interpretação de pseudocódigo. Em outra rajada de otimismo, afirmava que Fortran eliminaria os erros de codificação e o processo de depuração. Baseado nessa premissa, o primeiro compilador Fortran incluía pouca verificação de erros de sintaxe.

O ambiente no qual Fortran foi desenvolvida era o seguinte: (1) os computadores tinham memórias pequenas, eram lentos e relativamente não confiáveis, (2) o uso primário dos computadores era para computações científicas, (3) não existiam maneiras eficientes e efetivas de programar computadores, e (4) devido ao alto custo dos computadores comparado com o dos programadores, a velocidade do código objeto gerado era o objetivo principal dos primeiros compiladores Fortran. As características das primeiras versões de Fortran são diretamente oriundas desse ambiente.

2.3.3 Visão geral de Fortran I

Fortran 0 foi modificado durante o período de implementação, que começou em janeiro de 1955 e continuou até o lançamento do compilador, em abril de 1957. A linguagem implementada, que chamamos de Fortran I, é descrita no primeiro *Manual de Referência do Programador Fortran*, publicado em outubro de 1956 (IBM, 1956). Fortran I incluía formatação de entrada e saída, nomes de variáveis com até seis caracteres (eram apenas dois em Fortran 0), sub-rotinas definidas pelos usuários, apesar de não poderem ser compiladas separadamente, a sentença de seleção `If` e a sentença de repetição `Do`.

Todas as sentenças de controle de Fortran I eram baseadas em instruções do 704. Não fica claro se os projetistas do 704 ditaram o projeto das sentenças de controle da Fortran I ou se os de Fortran I sugeriram essas instruções para os do 704.

Não havia sentenças de tipo de dado na linguagem Fortran I. Variáveis cujos nomes começavam com `I`, `J`, `K`, `L`, `M` e `N` eram implicitamente de tipo inteiro e todas as outras eram implicitamente de ponto flutuante. A escolha das letras para essa convenção foi baseada no fato de que, na época, os cientistas e engenheiros usavam letras como índices de variável, normalmente *i*, *j* e *k*. Em um gesto de generosidade, os projetistas da Fortran inseriram mais três letras.

A afirmação mais audaciosa feita pelo grupo de desenvolvimento de Fortran, durante o projeto da linguagem, foi que o código de máquina produzido pelo compilador teria cerca de metade da eficiência do que poderia ser produzido manualmente.[1] Isso, mais do que qualquer coisa, fez os usuários em potencial ficarem céticos a seu respeito e frustrou muito do interesse na Fortran antes de seu lançamento. Para a surpresa de muitos, entretanto, o grupo de desenvolvimento de Fortran quase atingiu sua meta de eficiência. Grande parte do esforço de 18 trabalhadores/ano usada para construir o primeiro compilador foi gasta em otimização, e os resultados foram extraordinariamente eficazes.

O rápido sucesso de Fortran é mostrado pelos resultados de uma pesquisa feita em abril de 1958. Na época, aproximadamente metade do código que estava sendo escrito para o 704 era em Fortran, apesar do ceticismo do mundo da programação apenas um ano antes.

2.3.4 Fortran II

O compilador Fortran II foi distribuído na primavera de 1958. Ele corrigiu diversos problemas do sistema de compilação do Fortran I e adicionou recursos significativos à linguagem, o mais importante dos quais foi a compilação independente de sub-rotinas. Sem a compilação independente, quaisquer mudanças em um programa exigiam que ele todo fosse recompilado. A falta de compilação independente em Fortran I, agregada à pouca confiabilidade do 704, impôs uma restrição prática ao tamanho máximo dos programas, de cerca de 300 a 400 linhas (Wexelblat, 1981, p. 68). Programas mais extensos tinham pouca chance de serem compilados completamente antes da ocorrência de uma falha de máquina. A capacidade de incluir versões de subprogramas em linguagem de máquina previamente compilados abreviou o processo de compilação consideravelmente e possibilitou o desenvolvimento de programas muito maiores.

2.3.5 Fortrans IV, 77, 90, 95, 2003 e 2008

Um Fortran III foi desenvolvido, mas nunca amplamente distribuído. Fortran IV, entretanto, tornou-se uma das linguagens de programação mais utilizadas de seu tempo. Ela evoluiu no período de 1960 até 1962 e foi padronizada como Fortran 66 (ANSI, 1966),

[1] Na verdade, a equipe da Fortran acreditava que o código gerado pelo compilador não poderia ter menos da metade da velocidade do código de máquina manuscrito, senão a linguagem não seria adotada pelos usuários.

apesar de esse nome ser raramente usado. Fortran IV foi uma melhoria em relação a Fortran II em muitos pontos. Entre suas adições mais importantes estão as declarações de tipo explícitas para variáveis, uma construção `If` lógica e a capacidade de passar subprogramas como parâmetros para outros subprogramas.

Fortran IV foi substituída por Fortran 77, que se tornou o novo padrão em 1978 (ANSI, 1987a). Ela manteve a maioria dos recursos da Fortran IV e adicionou manipulação de cadeias de caracteres, sentenças de controle de laços lógicos e um `If` com uma cláusula opcional `Else`.

Fortran 90 (ANSI, 1992) era drasticamente diferente da Fortran 77. As adições mais significativas foram os vetores dinâmicos, os registros, os ponteiros, uma sentença de seleção múltipla e os módulos. Além disso, os subprogramas Fortran 90 podiam ser chamados recursivamente.

Um novo conceito incluído na definição de Fortran 90 foi o de remover alguns recursos da linguagem de versões anteriores. Embora Fortran 90 tivesse todos os recursos de Fortran 77, a definição da linguagem incluía uma lista de construções recomendadas para remoção na próxima versão da linguagem.

Fortran 90 incluía duas mudanças sintáticas simples que alteravam a aparência tanto de programas quanto da literatura de descrição da linguagem. Primeiro, o formato fixo obrigatório do código, que exigia o uso de posições específicas dos caracteres para partes específicas das sentenças, foi abandonado. Por exemplo, rótulos poderiam aparecer apenas nas primeiras cinco posições e as sentenças não poderiam começar antes da sétima posição. Essa formatação rígida de código era projetada para o uso com cartões perfurados. A segunda mudança foi na grafia oficial, de FORTRAN para Fortran. Essa alteração foi acompanhada pela mudança na convenção de usar apenas letras maiúsculas para palavras-chave e identificadores em programas Fortran. A nova convenção era que apenas a primeira letra das palavras-chave e dos identificadores deveria ter letra maiúscula.

Fortran 95 (INCITS/ISO/IEC, 1997) continuou a evolução da linguagem, mas apenas algumas mudanças foram feitas. Entre outras, uma nova construção de iteração, Forall, foi adicionada para facilitar a tarefa de paralelizar os programas Fortran.

Fortran 2003 (Metcalf et al., 2004) adicionou suporte à programação orientada a objetos, tipos derivados parametrizados, ponteiros para procedimentos e interoperabilidade com a linguagem de programação C.

A versão mais recente de Fortran, Fortran 2008 (ISO/IEC 1539-1, 2010), adicionou suporte para blocos para definir escopos locais, covetores, que fornecem um modelo de execução paralela, e a construção `DO CONCURRENT`, para especificar laços sem interdependências.

2.3.6 Avaliação

A equipe de desenvolvimento original de Fortran pensou no projeto da linguagem apenas como um prelúdio necessário para a tarefa crítica de projetar o tradutor. Além disso, a equipe nunca havia pensado que Fortran seria usada em computadores não fabricados pela IBM. Contudo, eles foram forçados a considerar a construção de compiladores Fortran para outras máquinas IBM porque o sucessor do 704, o 709, foi anunciado antes que

o compilador Fortran para o 704 fosse lançado. O efeito de Fortran no uso de computadores, além do fato de que todas as linguagens de programação subsequentes devem algo a ele, é impressionante, considerando os modestos objetivos de seus projetistas.

Um dos recursos de Fortran I – e de todos os seus sucessores antes de 1990 – que permitia compiladores altamente otimizados era o fato de os tipos e o armazenamento para todas as variáveis serem fixados antes da execução. Nenhuma nova variável ou espaço podia ser alocado durante a execução. A flexibilidade foi minimizada em prol da simplicidade e da eficiência. Isso eliminou a possibilidade de subprogramas recursivos e tornou difícil implementar estruturas de dados que crescem ou mudam de forma dinamicamente. É claro, os tipos de programas construídos na época do desenvolvimento das primeiras versões de Fortran eram simples e primariamente numéricos em sua natureza, se comparados aos projetos de software mais recentes. Assim, as perdas não eram muito significativas.

O sucesso global de Fortran é inegável: ele mudou substancialmente o modo de usar computadores. Isso, é claro, em grande parte por ter sido a primeira linguagem de alto nível amplamente utilizada. Em comparação com conceitos e linguagens desenvolvidos depois, as primeiras versões de Fortran sofreram de várias maneiras, como seria de se esperar. Afinal, não seria justo comparar o desempenho e o conforto de um Ford T modelo de 1910 com o desempenho e o conforto de um Ford Mustang 2015. Independentemente disso, apesar das inadequações de Fortran, o momento de alto investimento em software escrito na linguagem, entre outros fatores, o mantiveram em uso por quase 60 anos.

Alan Perlis, um dos projetistas de ALGOL 60, disse sobre Fortran, em 1978: "Fortran é a *língua franca* do mundo da computação. É a língua das ruas no melhor sentido da palavra. E ele tem sobrevivido e sobreviverá, porque se tornou uma parte extraordinariamente útil de um comércio vital" (Wexelblat, 1981, p. 161).

A seguir, temos um exemplo de um programa em Fortran 95:

```
! Programa de exemplo de Fortran 95
!  Entrada:    Um inteiro, List_Len, onde List_Len é menor que
!              100, seguido por valores inteiros List_Len
!  Saída:      O número de valores de entrada que são maiores
!              que a média de todos os valores de entrada
Implicit none
Integer Dimension(99) :: Int_List
Integer :: List_Len, Counter, Sum, Average, Result
Result= 0
Sum = 0
Read *, List_Len
If ((List_Len > 0) .AND. (List_Len < 100)) Then
! Lê dados de entrada em uma matriz e calcula a soma
   Do Counter = 1, List_Len
      Read *, Int_List(Counter)
      Sum = Sum + Int_List(Counter)
   End Do
```

```
! Calcula a média
   Average = Sum / List_Len
! Conta os valores maiores que a média
   Do Counter = 1, List_Len
        If (Int_List(Counter) > Average) Then
           Result = Result + 1
        End If
   End Do
! Imprimir o resultado
   Print *, 'Number of values > Average is:', Result
Else
   Print *, 'Error - list length value is not legal'
End If
End Program Example
```

2.4 PROGRAMAÇÃO FUNCIONAL: LISP

A primeira linguagem de programação funcional foi desenvolvida para o processamento de listas, uma necessidade que cresceu a partir das primeiras aplicações na área de Inteligência Artificial (IA).

2.4.1 O início da inteligência artificial e do processamento de listas

O interesse em IA se intensificou em meados dos anos 1950, em vários lugares. Parte se originou da linguística, parte da psicologia e parte da matemática. Os linguistas estavam interessados no processamento de linguagem natural. Os psicólogos, em modelar o armazenamento e a recuperação de informações humanas. Os matemáticos, em mecanizar certos processos inteligentes, como a prova de teoremas. Todas essas pesquisas chegaram à mesma conclusão: algum método deve ser desenvolvido para permitir aos computadores processar dados simbólicos em listas encadeadas. Na época, a maioria das computações era feita em dados numéricos armazenados em vetores.

O conceito de processamento de listas foi desenvolvido por Allen Newell, J. C. Shaw e Herbert Simon na RAND Corporation. Foi publicado pela primeira vez em um artigo clássico que descreve um dos primeiros programas de IA, o Logic Theorist,[2] e uma linguagem na qual poderia ser implementado (Newell e Simon, 1956). A linguagem, chamada IPL-I (Information Processing Language I), nunca foi implementada. A versão seguinte, IPL-II, foi implementada em um computador Johnniac da RAND. O desenvolvimento da IPL continuou até 1960, quando a descrição da IPL-V foi publicada (Newell e Tonge, 1960). O baixo nível da linguagem evitou sua popularização. Elas eram linguagens de montagem para um computador hipotético, implementadas com um

[2] Logic Theorist descobriu provas para teoremas do cálculo proposicional.

interpretador no qual foram incluídas instruções de processamento de listas. Outro fator que impediu as linguagens IPL de se tornarem populares foi sua implementação na obscura máquina Johnniac.

As contribuições das linguagens IPL foram o seu projeto baseado em listas e a sua demonstração de que o processamento de listas era viável e útil.

A IBM começou a se interessar por IA em meados dos anos 1950 e escolheu a prova de teoremas como área de demonstração. Na época, o projeto da Fortran ainda estava no meio do caminho. O alto custo do compilador para Fortran I convenceu a IBM de que seu processamento de listas deveria ser anexado a Fortran, em vez de na forma de uma nova linguagem. Então, a Fortran List Processing Language (FLPL) foi projetada e implementada como uma extensão de Fortran. A FLPL foi usada para construir um provador de teoremas para geometria plana, que era considerada a área mais fácil para a prova mecânica de teoremas.

2.4.2 O processo do projeto de Lisp

John McCarthy, do MIT, passou uma temporada de verão no Departamento de Pesquisa em Informação da IBM, em 1958. Seu objetivo era investigar computações simbólicas e desenvolver um conjunto de requisitos para fazê-las. Como área de problema de exemplo piloto, ele escolheu a diferenciação de expressões algébricas. A partir desse estudo, foi criada uma lista dos requisitos da linguagem. Entre eles estavam os métodos de controle de fluxo de funções matemáticas: recursão e expressões condicionais. A única linguagem de alto nível disponível na época, Fortran I, não tinha nenhuma das duas.

Outra necessidade que surgiu a partir da pesquisa sobre diferenciação simbólica foi a de alocar listas encadeadas dinamicamente e algum tipo de liberação implícita de listas abandonadas. McCarthy não permitiria que seu elegante algoritmo para diferenciação fosse inchado com sentenças explícitas de liberação.

Como a FLPL não suportava recursão, expressões condicionais, alocação dinâmica de armazenamento e alocação implícita, estava claro para McCarthy que era necessária uma nova linguagem.

Quando McCarthy retornou ao MIT, no final de 1958, ele e Marvin Minsky formaram o Projeto IA do MIT, com financiamento do Laboratório de Pesquisa para Eletrônica. O primeiro esforço importante foi produzir um sistema de software para o processamento de listas. Inicialmente, ele foi usado para implementar um programa proposto por McCarthy, chamado Advice Taker.[3] Essa aplicação se tornou o ímpeto para o desenvolvimento da linguagem de processamento de listas chamada Lisp. A primeira versão de Lisp é, algumas vezes, chamada de "Lisp puro", porque é uma linguagem puramente funcional. Na seção a seguir descreveremos o desenvolvimento de Lisp pura.

[3] Advice Taker representava as informações com sentenças escritas em uma linguagem formal e usava um processo de inferência lógica para decidir o que fazer.

2.4.3 Visão geral da linguagem

2.4.3.1 Estruturas de dados

Lisp pura tem apenas dois tipos de estruturas de dados: átomos e listas. Átomos são símbolos, que têm a forma de identificadores ou literais numéricos. O conceito de armazenar informações simbólicas em listas encadeadas é natural e era usado na IPL-II. Tais estruturas permitem a inserção e a exclusão em qualquer ponto – operações pensadas, na época, como parte necessária do processamento de listas. Foi determinado, entretanto, que os programas Lisp raramente precisavam dessas operações.

As listas são especificadas com a delimitação de seus elementos com parênteses. Listas simples, nas quais os elementos são restritos a átomos, têm o formato

```
(A B C D)
```

Estruturas de lista aninhadas também são especificadas com parênteses. Por exemplo,

```
(A (B C) D (E (F G)))
```

é composta de quatro elementos. O primeiro é o átomo `A`; o segundo é a sublista `(B C)`; o terceiro é o átomo `D`; o quarto é a sublista `(E (F G))`, que tem como segundo elemento a sublista `(F G)`.

Internamente, as listas são armazenadas como estruturas de lista simplesmente encadeadas, nas quais cada nó tem dois ponteiros e representa um elemento da lista. Um nó contendo um átomo tem o primeiro ponteiro apontando para alguma representação do átomo, como seu símbolo ou valor numérico, ou um ponteiro para uma sublista. Um nó para um elemento de sublista tem seu primeiro ponteiro apontando para o primeiro nó da sublista. Em ambos os casos, o segundo ponteiro de um nó aponta para o próximo elemento da lista. Uma lista é referenciada por um ponteiro para o seu primeiro elemento.

As representações internas das duas listas mostradas anteriormente são ilustradas na Figura 2.2. Note que os elementos de uma lista são mostrados horizontalmente. O último elemento de uma lista não tem sucessor, então sua ligação é `NIL`, representado na Figura 2.2 como uma linha diagonal no elemento. Sublistas são mostradas com a mesma estrutura.

2.4.3.2 Processos em programação funcional

Lisp foi projetada como uma linguagem de programação funcional. Todas as computações em um programa puramente funcional são realizadas por meio da aplicação de funções a argumentos. Nem as sentenças de atribuição nem as variáveis, abundantes em programas escritos em linguagens imperativas, são necessárias em programas escritos em linguagens funcionais. Além disso, processos repetitivos podem ser especificados com chamadas a funções recursivas, tornando as iterações (laços) desnecessárias. Esses conceitos básicos de programação funcional a tornam significativamente diferente da programação em uma linguagem imperativa.

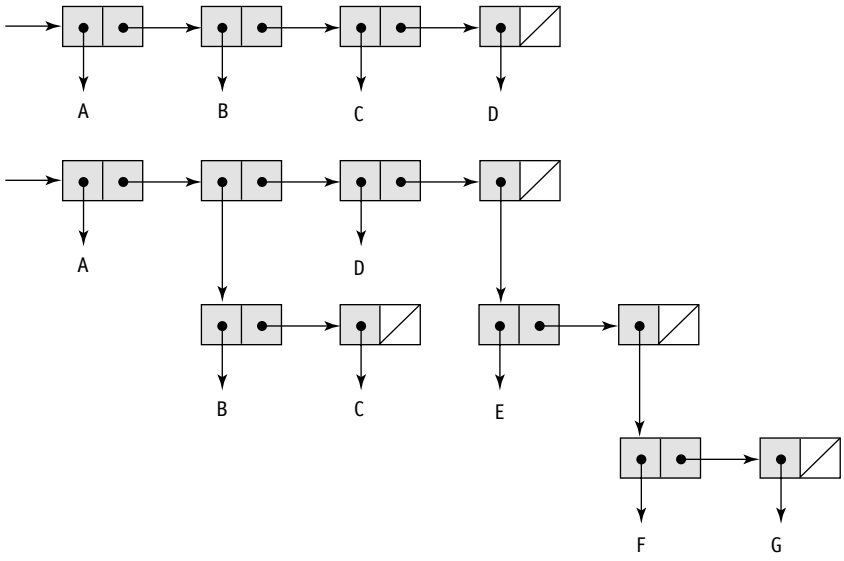


FIGURA 2.2
Representação interna de duas listas Lisp.

2.4.3.3 A sintaxe de Lisp

Lisp é muito diferente das linguagens imperativas, tanto porque é funcional quanto porque a aparência dos seus programas é muito diferente daquela de linguagens como Java ou C++. Por exemplo, a sintaxe de Java é uma mistura complicada de inglês e álgebra, enquanto a sintaxe de Lisp é um modelo de simplicidade. O código e os dados dos programas têm exatamente a mesma forma: listas dentro de parênteses. Considere mais uma vez a lista

```
(A B C D)
```

Quando interpretada como dados, ela é uma lista de quatro elementos. Se vista como código, é a aplicação da função chamada `A` para os três parâmetros `B`, `C` e `D`.

2.4.4 Avaliação

Lisp dominou completamente as aplicações de IA por um quarto de século. Muitas das causas da má reputação de Lisp foram eliminadas. Muitas das implementações contemporâneas são compiladas, e o código resultante é muito mais rápido do que executar o código-fonte em um interpretador. Além do seu sucesso em IA, Lisp foi pioneira na programação funcional, que se mostrou uma área extremamente ativa na pesquisa em linguagens de programação. Conforme mencionado no Capítulo 1, muitos pesquisadores de linguagens de programação acreditam que a programação funcional é uma estratégia muito melhor para o desenvolvimento de software do que a programação procedural com linguagens imperativas.

A seguir, temos um exemplo de um programa em Lisp:

```
; Função de exemplo em Lisp
; O código a seguir define uma função de predicado em Lisp
; que recebe duas listas como argumentos e retorna True
; se as duas listas forem iguais, e NIL (false) caso contrário
  (DEFUN equal_lists (lis1 lis2)
   (COND
     (ATOM lis1) (EQ lis1 lis2))
     ((ATOM lis2) NIL)
     ((equal_lists (CAR lis1) (CAR lis2))
                 (equal_lists (CDR lis1) (CDR lis2)))
     (T NIL)
   )
)
```

2.4.5 Dois descendentes de Lisp

Dois dialetos de Lisp são muito usados agora, Scheme e Common Lisp. Eles são discutidos brevemente nas subseções a seguir.

2.4.5.1 Scheme

A linguagem Scheme emergiu do MIT em meados dos anos 1970. Ela é caracterizada por seu tamanho diminuto, seu uso exclusivo de escopo estático (discutido no Capítulo 5) e seu tratamento de funções como entidades de primeira classe. As funções Scheme podem ser atribuídas a variáveis, passadas como parâmetros e retornadas como valores da aplicação de funções. Elas também podem ser elementos de listas. As primeiras versões de Lisp não forneciam todas essas capacidades, nem usavam escopo estático.

Como uma linguagem pequena, com sintaxe e semântica simples, Scheme é bastante adequada para aplicações educacionais, como cursos sobre programação funcional e introduções gerais à programação. Ela é descrita em alguns detalhes no Capítulo 15.

2.4.5.2 Common Lisp

Durante os anos 1970 e o início dos anos 1980, diversos dialetos diferentes de Lisp foram desenvolvidos e usados. Isso levou ao conhecido problema da falta de portabilidade entre os programas escritos nos vários dialetos. Common Lisp (Graham, 1996) foi criada em um esforço de corrigir essa situação. Ela foi projetada com a combinação dos recursos de vários dialetos de Lisp desenvolvidos no início dos anos 1980, incluindo Scheme, em uma única linguagem. Sendo um amálgama, Common Lisp é uma linguagem relativamente extensa e complexa. Sua base, entretanto, é Lisp pura; então, sua sintaxe, funções primitivas e natureza fundamental vêm dessa linguagem.

Reconhecendo a flexibilidade fornecida pelo escopo dinâmico e a simplicidade do escopo estático, Common Lisp permite ambos. O escopo padrão para variáveis é estático, mas, ao se declarar uma variável como **`special`** (especial), o escopo dela se torna dinâmico.

Common Lisp tem um grande número de tipos e estruturas de dados, incluindo registros, vetores, números complexos e cadeias de caracteres. Tem também uma forma de pacotes para modular coleções de funções e dados que fornecem controle de acesso.

Common Lisp é descrito em mais detalhes no Capítulo 15.

2.4.6 Linguagens relacionadas

A ML (*M*eta*L*anguage; Ullman, 1998) foi originalmente projetada nos anos 1980 por Robin Milner, na Universidade de Edimburgo, como uma metalinguagem para um sistema de verificação de programas chamado Logic for Computable Functions (LCF; Milner et al., 1990). ML é primariamente uma linguagem funcional, mas também oferece suporte para programação imperativa. Ao contrário do que ocorre em Lisp e Scheme, em ML o tipo de cada variável e expressão pode ser determinado em tempo de compilação. Os tipos são associados a objetos, em vez de nomes. Os tipos de nomes e expressões são deduzidos a partir de seu contexto.

Diferentemente de Lisp e de Scheme, ML não usa a sintaxe funcional baseada em parênteses que se originou com expressões lambda. Em vez disso, a sintaxe de ML lembra a das linguagens imperativas, como Java e C++.

Miranda foi desenvolvida por David Turner (1986) na Universidade de Kent, em Canterbury, na Inglaterra, no início dos anos 1980. Ela é parcialmente baseada nas linguagens ML, SASL e KRC. Haskell (Hudak e Fasel, 1992) é baseada em grande parte na Miranda. Como esta, é uma linguagem puramente funcional, sem variáveis e sentenças de atribuição. Outra característica distintiva de Haskell é o uso de avaliação tardia (*lazy evaluation*). Isso significa que nenhuma expressão é avaliada até que seu valor seja exigido. Isso leva a algumas capacidades surpreendentes na linguagem.

Caml (Cousineau et al., 1998) e seu dialeto que suporta programação orientada a objetos, OCaml (Smith, 2006), descendem de ML e de Haskell. Por fim, F# é uma linguagem tipada relativamente nova, baseada diretamente em OCaml. Ela (Syme et al., 2010) é uma linguagem .NET com acesso direto a toda a biblioteca .NET. Ser uma linguagem .NET também significa que ela pode interagir diretamente com qualquer outra linguagem .NET. F# suporta programação funcional e programação procedural, além de programação orientada a objetos.

ML, Haskell e F# são discutidas mais detalhadamente no Capítulo 15.

2.5 O PRIMEIRO PASSO EM DIREÇÃO À SOFISTICAÇÃO: ALGOL 60

ALGOL 60 influenciou muito as linguagens de programação subsequentes e é de importância central em qualquer estudo histórico de linguagens.

2.5.1 Perspectiva histórica

ALGOL 60 é o resultado de esforços para projetar uma linguagem de programação universal para aplicações científicas. No final de 1954, o sistema algébrico de Laning e

Zierler estava em operação há mais de um ano, e o primeiro relatório sobre Fortran havia sido publicado. Fortran se tornou uma realidade em 1957, e diversas outras linguagens de alto nível foram desenvolvidas. Entre essas, a mais notável era a IT, projetada por Alan Perlis no Carnegie Tech, e duas linguagens para os computadores UNIVAC, MATH-MATIC e UNICODE. A proliferação de linguagens fez com que o compartilhamento de programas entre usuários se tornasse difícil. Além disso, as novas linguagens estavam crescendo em torno de arquiteturas únicas, algumas para os computadores UNIVAC, outras para as máquinas da série IBM 700. Em resposta a esse florescimento de linguagens dependentes da máquina, diversos dos principais usuários de computadores nos Estados Unidos, incluindo o SHARE (o grupo de usuários científicos da IBM) e o USE (UNIVAC Scientific Exchange, o grupo de grande escala de usuários científicos da UNIVAC), submeteram uma petição à ACM (Association for Computing Machinery), em 10 de maio de 1957, a fim de formar um comitê para estudar e recomendar ações para uma linguagem de programação científica independente de máquina. Embora Fortran pudesse ser tal candidata, não se tornaria uma linguagem universal, já que era de propriedade única da IBM.

Anteriormente, em 1955, a GAMM (Sociedade de Matemática e Mecânica Aplicada, na sigla em alemão) formou um comitê para projetar uma linguagem algorítmica universal, independente de máquina. O desejo por essa nova linguagem era em parte devido ao medo dos europeus de serem dominados pela IBM. No final de 1957, entretanto, a aparição de diversas linguagens de alto nível nos Estados Unidos convenceu o subcomitê da GAMM de que seus esforços precisavam ser ampliados para incluir os americanos, e uma carta-convite foi enviada à ACM. Em abril de 1958, após Fritz Bauer, da GAMM, apresentar uma proposta formal à ACM, os dois grupos concordaram oficialmente em desenvolver uma linguagem conjunta.

2.5.2 Processo do projeto inicial

A GAMM e a ACM enviaram, cada uma, quatro membros para a primeira reunião de projeto. A reunião, realizada em Zurique de 27 de maio a 1º de junho de 1958, começou com os seguintes objetivos para a nova linguagem:

- A sintaxe da linguagem deve ser o mais próximo possível da notação padrão matemática e os programas devem ser legíveis, com poucas explicações adicionais.
- Deve ser possível usar a linguagem para a descrição de algoritmos em publicações impressas.
- Programas na nova linguagem devem ser mecanicamente traduzíveis em código de máquina.

O primeiro objetivo indica que a nova linguagem seria usada para programação científica, a principal área da aplicação de computadores na época. O segundo era algo inteiramente novo nos negócios em computação. O último é uma necessidade óbvia para qualquer linguagem de programação.

A reunião de Zurique foi bem-sucedida em produzir uma linguagem que atendesse aos objetivos levantados, mas o processo do projeto necessitava de inúmeros comprome-

timentos, tanto entre indivíduos quanto entre os dois lados do Atlântico. Em alguns casos, não era tanto em torno de grandes questões, mas em termos de esferas de influência. A questão de usar uma vírgula (o método europeu) ou um ponto (o método americano) para um ponto decimal é um exemplo.

2.5.3 Visão geral de ALGOL 58

A linguagem projetada na reunião em Zurique foi denominada Linguagem Algorítmica Internacional (IAL – International Algorithmic Language). Durante o projeto, foi sugerido que ela se chamasse ALGOL, do inglês ALGOrithmic Language, mas o nome foi rejeitado porque não refletia o escopo internacional do comitê. Durante o ano seguinte, entretanto, o nome foi mudado para ALGOL, e a linguagem ficou conhecida como ALGOL 58.

De muitas formas, ALGOL 58 é descendente de Fortran, o que é bastante natural. Ele generalizou muitos dos recursos de Fortran e adicionou novas construções e conceitos. Algumas das generalizações eram relativas ao objetivo de não vincular a linguagem a nenhuma máquina em particular, e outras eram tentativas de tornar a linguagem mais flexível e poderosa. Uma rara combinação de simplicidade e elegância emergiu desse esforço.

ALGOL 58 formalizou o conceito de tipo de dados, apesar de apenas variáveis que não fossem de ponto flutuante precisarem ser explicitamente declaradas. Ele adicionou a ideia de sentenças compostas, que a maioria das linguagens subsequentes incorporou. Alguns recursos de Fortran que foram generalizados são os seguintes: os identificadores podiam ter qualquer tamanho, em oposição à restrição de Fortran I de nomes de identificadores com até seis caracteres; qualquer número de dimensões de um vetor era permitido, diferentemente da limitação de Fortran I de até três dimensões; o limite inferior dos vetores podia ser especificado pelo programador, enquanto em Fortran ele era implicitamente igual a 1; sentenças de seleção aninhadas eram permitidas, o que não era o caso em Fortran I.

ALGOL 58 adquiriu o operador de atribuição de uma maneira um tanto incomum. Zuse utilizou a forma

expressão => variável

para a sentença de atribuição em Plankalkül. Apesar de Plankalkül não ter sido ainda publicada, alguns dos membros europeus do comitê da ALGOL 58 conheciam a linguagem. O comitê se interessou pela forma da sentença de Plankalkül, mas, por causa dos argumentos sobre as limitações do conjunto de caracteres,[4] o símbolo de maior foi mudado para dois pontos. Então, em grande parte por causa da insistência dos americanos, toda a sentença foi modificada para ser equivalente ao formato de Fortran

variável := expressão

Os europeus preferiam a forma oposta, mas ela seria o inverso de Fortran.

[4] As perfuradoras de cartão da época não incluíam o símbolo de maior que.

2.5.4 Recepção do relatório da ALGOL 58

Em dezembro de 1958, a publicação do relatório da ALGOL 58 (Perlis e Samelson, 1958) foi recebida com entusiasmo. Nos Estados Unidos, a nova linguagem foi vista mais como uma coleção de ideias para o projeto de linguagens de programação do que como uma linguagem padrão universal. Na verdade, o relatório de ALGOL 58 não pretendia ser um produto finalizado, mas um documento preliminar para discussão internacional. Independentemente disso, três grandes esforços de projeto e implementação foram feitos, usando o relatório como base. Na Universidade de Michigan, a linguagem MAD nasceu (Arden et al., 1961). O Grupo de Eletrônica Naval dos Estados Unidos produziu a linguagem NELIAC (Huskey et al., 1963). Na System Development Corporation, a linguagem JOVIAL foi projetada e implementada (Shaw, 1963). JOVIAL, um acrônimo para Jules' Own Version of the International Algebraic Language (Versão de Jules para a Linguagem Internacional Algébrica), representa a única linguagem baseada em ALGOL 58 a atingir uso amplo. (Jules era Jules I. Schwartz, um dos projetistas da JOVIAL). A JOVIAL se tornou bastante usada porque foi a linguagem científica oficial para a Força Aérea Americana por um quarto de século.

O restante da comunidade da computação nos Estados Unidos não foi tão gentil com a nova linguagem. Em um primeiro momento, tanto a IBM quanto seu maior grupo de usuários científicos, o SHARE, pareciam ter abraçado ALGOL 58. A IBM iniciou uma implementação logo após o relatório ter sido publicado, e o SHARE formou um subcomitê, SHAREL IAL, para estudar a linguagem. Logo a seguir, o subcomitê recomendou que a ACM padronizasse ALGOL 58 e que a IBM a implementasse para toda a série de computadores IBM 700. O entusiasmo durou pouco. Na primavera de 1959, tanto a IBM quanto o SHARE, com sua experiência com Fortran, já haviam tido decepções e despesas suficientes para iniciar uma nova linguagem, tanto em termos de desenvolver e usar os compiladores de primeira geração quanto de treinar os usuários na nova linguagem e persuadi-los a usá-la. Em meados de 1959, tanto a IBM quanto o SHARE haviam desenvolvido um interesse próprio por Fortran, tomando a decisão de mantê-la como *a* linguagem científica para as máquinas IBM da série 700, abandonando ALGOL 58.

2.5.5 O processo do projeto da ALGOL 60

Durante 1959, houve intenso debate em torno de ALGOL 58, tanto na Europa quanto nos Estados Unidos. Um grande número de modificações e adições foi publicado no *ALGOL Bulletin* europeu e na *Communications of the ACM*. Um dos eventos mais importantes de 1959 foi a apresentação do trabalho do comitê de Zurique na Conferência Internacional de Processamento de Informação, na qual Backus introduziu sua notação para descrever a sintaxe de linguagens de programação, posteriormente conhecida como BNF (Backus-Naur Form). A BNF é descrita em detalhes no Capítulo 3.

Em janeiro de 1960, foi realizada a segunda reunião da ALGOL, desta vez em Paris. O objetivo da reunião era debater as 80 sugestões que haviam sido formalmente submetidas para consideração. Peter Naur da Dinamarca havia se envolvido enormemente no desenvolvimento de ALGOL, apesar de não ser membro do grupo de Zurique. Foi Naur quem criou e publicou o *ALGOL Bulletin*. Ele passou bastante tempo estudando o artigo de Backus que apresentava a BNF e decidiu que ela deveria ser usada para descrever formalmente os resultados da reunião de 1960. Após ter feito algumas mudanças relati-

vamente pequenas na BNF, ele escreveu uma descrição da nova linguagem proposta em BNF e entregou para os membros do grupo de 1960 no início da reunião.

2.5.6 Visão geral de ALGOL 60

Apesar de a reunião de 1960 ter durado apenas seis dias, as modificações feitas na ALGOL 58 foram drásticas. Entre os mais importantes avanços estavam os seguintes:

- Foi introduzido o conceito de estrutura de bloco. Isso permitiu aos programadores localizarem partes de programas introduzindo novos ambientes de dados ou escopos.
- Duas formas diferentes de passagem de parâmetros a subprogramas foram permitidas: por valor e por nome.
- Foi permitido aos procedimentos serem recursivos. A descrição da ALGOL 58 não era clara em relação a essa questão. Note que, apesar da recursão ser nova para as linguagens imperativas, Lisp já fornecia funções recursivas em 1959.
- Vetores dinâmicos na pilha foram permitidos. Um vetor dinâmico na pilha é aquele para o qual a faixa ou faixas de índices são especificados por variáveis, de forma que seu tamanho é determinado no momento em que o armazenamento é alocado, o que acontece quando a declaração é alcançada durante a execução. Vetores dinâmicos na pilha são discutidos em detalhe no Capítulo 6.

Diversos recursos que poderiam ter gerado um grande impacto no sucesso ou na falha da linguagem foram propostos e rejeitados. O mais importante deles eram sentenças de entrada e saída com formatação, omitidas porque se pensava que seriam muito dependentes de máquina.

O relatório da ALGOL 60 foi publicado em maio de 1960 (Naur, 1960). Algumas ambiguidades ainda permaneceram na descrição da linguagem, e uma terceira reunião foi marcada para abril de 1962, em Roma, para resolver esses problemas. Nessa reunião, o grupo tratou apenas dos problemas; nenhuma adição à linguagem foi permitida. Os resultados foram publicados sob o título "Revised Report on the Algorithmic Language ALGOL 60" (Backus et al., 1963).

2.5.7 Avaliação

De algumas formas, ALGOL 60 foi um grande sucesso; de outras, um imenso fracasso. Foi bem-sucedida em se tornar, quase imediatamente, a única maneira formal aceitável de comunicar algoritmos na literatura da computação – e permaneceu assim por mais de 20 anos. Todas as linguagens de programação imperativas, desde 1960, devem algo ao ALGOL 60. De fato, muitas são descendentes diretos ou indiretos, como PL/I, SIMULA 67, ALGOL 68, C, Pascal, Ada, C++, Java e C#.

O esforço de projeto de ALGOL 58/ALGOL 60 incluiu uma longa lista de primeiras vezes. Essa foi a primeira vez que um grupo internacional tentou projetar uma linguagem de programação. Ela foi a primeira linguagem projetada para ser independente de

máquina. Foi também a primeira linguagem cuja sintaxe foi descrita formalmente. O sucesso do uso do formalismo BNF iniciou diversos campos importantes na ciência da computação: linguagens formais, teoria de análise sintática e projeto de compilador baseado em BNF. Finalmente, a estrutura da ALGOL 60 afetou as arquiteturas de máquina. O exemplo mais contundente disso é que uma extensão da linguagem foi usada como a linguagem de sistema de uma série de computadores de grande escala, as máquinas Borroughs B5000, B6000 e B7000, projetadas com uma pilha de hardware para implementar eficientemente a estrutura de bloco e os subprogramas recursivos da linguagem.

Do outro lado da moeda, ALGOL 60 nunca atingiu um uso disseminado nos Estados Unidos. Mesmo na Europa, onde era mais popular, nunca se tornou a linguagem dominante. Existem diversas razões para essa baixa aceitação. Por um lado, alguns dos recursos se mostraram flexíveis demais – e fizeram com que o entendimento da linguagem fosse mais difícil e a implementação ineficiente. O melhor exemplo disso é o método de passagem de parâmetros por nome para os subprogramas, explicado no Capítulo 9. As dificuldades para implementar ALGOL 60 foram evidenciadas por Rutishauser em 1967, que disse que poucas implementações (se é que alguma) incluíam a linguagem ALGOL 60 completa (Rutishauser, 1967, p. 8).

A falta de sentenças de entrada e saída na linguagem foi outra razão para sua falta de aceitação. A entrada e saída dependente de implementação fez os programas terem uma portabilidade ruim para outros computadores.

Ironicamente, uma das mais importantes contribuições à ciência da computação associada a ALGOL 60, a BNF, também foi um fator. Apesar de a BNF ser agora considerada uma maneira simples e elegante de descrição de sintaxe, em 1960 ela parecia estranha e complicada.

Finalmente, apesar de haver muitos outros problemas, o forte estabelecimento de Fortran entre os usuários e a falta de suporte da IBM foram provavelmente os fatores mais importantes para que a ALGOL 60 não tivesse seu uso disseminado.

O esforço da ALGOL 60 nunca foi realmente completo, no sentido de que ambiguidades e obscuridades sempre fizeram parte da descrição da linguagem (Knuth, 1967).

A seguir, é mostrado um exemplo de um programa em ALGOL:

```
comment Programa de exemplo de ALGOL 60
 Entrada: Um inteiro, listlen, onde listlen é menor que
          100, seguido por valores inteiros listlen.
 Saída:   O número de valores de entrada que são maiores
          que a média de todos os valores de entrada.
begin
  integer array intlist [1:99];
  integer listlen, counter, sum, average, result;
  sum := 0;
  result := 0;
  readint (listlen);
  if (listlen > 0) ^ (listlen < 100) then
    begin
```

```
comment Lê os dados de entrada em um vetor e calcula sua média;
    for counter := 1 step 1 until listlen do
      begin
      readint (intlist[counter]);
      sum := sum + intlist[counter]
      end;
comment Calcula a média;
    average := sum / listlen;
comment Conta os valores maiores que a média;
    for counter := 1 step 1 until listlen do
      if intlist[counter] > average
       then result := result + 1;
comment Imprimir o resultado;
    printstring("The number of values > average is:");
    printint (result)
    end
  else
    printstring ("Error-input list length is not legal";
end
```

2.6 INFORMATIZAÇÃO DE REGISTROS COMERCIAIS: COBOL

A história de COBOL é, de certa forma, o oposto da de ALGOL 60. Apesar de ter sido mais usada por 65 anos, ela teve pouco efeito no projeto de linguagens subsequentes, exceto por PL/I. Talvez ainda seja a linguagem mais amplamente utilizada,[5] embora seja difícil precisar isso. Talvez a razão mais importante para COBOL ter uma influência pequena é que poucos tentaram projetar uma nova linguagem de negócios desde sua aparição. Isso ocorreu em parte porque COBOL atendem bem as necessidades de sua área de aplicação. Outra razão é que uma grande parcela da computação de negócios nos últimos 30 anos ocorreu em pequenas empresas. E, nelas, há pouco desenvolvimento de software. Em vez disso, muito do software usado é comprado como pacotes de prateleira para várias aplicações gerais.

2.6.1 Perspectiva histórica

O início de COBOL é, de certa forma, similar ao de ALGOL 60. A linguagem também foi projetada por um comitê de pessoas que se reuniam em períodos relativamente curtos. Na época, em 1959, o estado da computação de negócios era similar ao estado da computação científica quando Fortran foi projetada. Uma linguagem compilada para aplicações de negócios, chamada FLOW-MATIC, havia sido implementada em 1957, mas pertencia à UNIVAC e foi projetada para os computadores dessa empresa. Outra

[5]No final dos anos 1990, em um estudo associado ao problema Y2K, foi estimado que havia aproximadamente 800 milhões de linhas de COBOL em uso regular nos quase 57 Km² de Manhattan.

linguagem, a AIMACO, era usada pela Força Aérea Americana, mas era apenas uma pequena variação de FLOW-MATIC. A IBM havia projetado uma linguagem de programação para aplicações comerciais, chamada COMTRAN (COMmercial TRANslator), mas ela nunca foi implementada. Diversos outros projetos de linguagens foram planejados.

2.6.2 FLOW-MATIC

Vale a pena discutir brevemente as origens de FLOW-MATIC, principal precursora de COBOL. Em dezembro de 1953, Grace Hopper, na Remington-Rand UNIVAC, escreveu uma proposta profética. Ela sugeria que "programas matemáticos deveriam ser escritos em notação matemática, programas de processamento de dados deviam ser escritos em sentenças em inglês" (Wexelbal, 1981, p. 16). Infelizmente, em 1953 era impossível convencer não programadores de que poderia ser feito um computador para entender palavras em inglês. Somente em 1955 uma proposta similar teve a possibilidade de ser patrocinada pela UNIVAC, e mesmo assim precisou de um protótipo para convencer a gerência. Parte do processo de venda envolvia compilar e executar um pequeno programa, primeiro usando palavras-chave em inglês, depois palavras-chave em francês e, então, palavras-chave em alemão. Essa demonstração foi considerada notável pela gerência da UNIVAC e foi providencial para a aceitação da proposta de Hopper.

2.6.3 O processo do projeto de COBOL

A primeira reunião formal sobre uma linguagem comum para aplicações de negócios, patrocinada pelo Departamento de Defesa, ocorreu no Pentágono, em 28 e 29 de maio de 1959 (exatamente um ano após a reunião da ALGOL em Zurique). O consenso do grupo era que a linguagem, na época chamada CBL (Common Business Language), deveria ter certas características gerais. A maioria concordou que ela deveria usar o inglês o máximo possível, apesar de alguns terem argumentado a favor de uma notação mais matemática. A linguagem deveria ser fácil de utilizar, mesmo ao custo de ser menos poderosa, de forma a aumentar a base daqueles que poderiam programar computadores. Somando-se ao fato de tornar a linguagem mais fácil, acreditava-se que o uso de inglês poderia permitir aos gerentes ler os programas. Finalmente, o projeto não deveria ser restringido pelos problemas de sua implementação.

Uma das preocupações recorrentes na reunião era que os passos para criar essa linguagem universal deveriam ser dados rapidamente, já que um monte de trabalho estava sendo feito para criar outras linguagens de negócios. Além das existentes, a RCA e a Sylvania estavam trabalhando em suas próprias linguagens para aplicações de negócios. Estava claro que, quanto mais tempo levasse para produzir uma linguagem universal, mais difícil seria para que ela se tornasse amplamente usada. Dessa forma, foi decidido que deveria existir um rápido estudo sobre as linguagens existentes. Para essa tarefa, foi formado o Short Range Committee.

Foram tomadas decisões iniciais para separar as sentenças da linguagem em duas categorias – descrições de dados e operações executáveis – e para que as sentenças des-

sas duas categorias ficassem em partes diferentes dos programas. Um dos debates do Short Range Committee foi sobre a inclusão de índices. Muitos membros do comitê argumentaram que índices seriam muito complexos para as pessoas que trabalhavam com processamento de dados, as quais, pensava-se, não ficariam à vontade com uma notação matemática. Argumentos similares giravam em torno de incluir ou não expressões aritméticas. O relatório final do Short Range Committee, concluído em dezembro de 1959, descrevia a linguagem que mais tarde foi chamada de COBOL 60.

A especificação da linguagem de COBOL 60, publicada pela Agência de Impressão do Governo Americano (Government Printing Office) em abril de 1960 (Department of Defense, 1960), foi descrita como "inicial". Versões revisadas foram publicadas em 1961 e 1962 (Department of Defense, 1961, 1962). A linguagem foi padronizada pelo Instituto Nacional de Padrões dos Estados Unidos (ANSI – American National Standards Institute) em 1968. As três revisões seguintes foram padronizadas pelo ANSI em 1974, 1985 e 2002. A linguagem continua a evoluir até hoje.

2.6.4 Avaliação

A linguagem COBOL originou diversos conceitos inovadores, alguns dos quais apareceram em outras linguagens. Por exemplo, o verbo `DEFINE` de COBOL 60 foi a primeira construção para macros de uma linguagem de alto nível. E o mais importante: estruturas de dados hierárquicas (registros), que apareceram em Plankalkül, foram primeiro implementadas em COBOL. Os registros têm sido incluídos na maioria das linguagens imperativas projetadas desde então. COBOL também foi a primeira linguagem a permitir nomes realmente conotativos, pois permitia nomes longos (até 30 caracteres) e caracteres conectores de palavras (hifens).

De modo geral, a divisão de dados (*data division*) é a parte forte do projeto de COBOL, enquanto a divisão de procedimentos (*procedure division*) é relativamente fraca. Cada variável é definida em detalhes na divisão de dados, incluindo o número de dígitos decimais e a localização do ponto decimal esperado. Registros de arquivos também são descritos com esse nível de detalhes, assim como o são as linhas a serem enviadas a uma impressora, tornando COBOL ideal para imprimir relatórios contábeis. Talvez a fraqueza mais importante da divisão de procedimentos original de COBOL tenha sido sua falta de funções. Versões de COBOL anteriores ao padrão 1974 também não permitiam subprogramas com parâmetros.

Nosso comentário final sobre COBOL: foi a primeira linguagem de programação cujo uso foi obrigatório pelo Departamento de Defesa americano (DoD). Essa obrigatoriedade veio após seu desenvolvimento inicial, já que COBOL não foi especificamente projetada para o DoD. Apesar de seus méritos, COBOL provavelmente não teria sobrevivido sem essa obrigatoriedade. O desempenho ruim dos primeiros compiladores simplesmente tornava a linguagem muito cara. Finalmente, é claro, os compiladores se tornaram mais eficientes e os computadores ficaram mais rápidos e baratos, com memórias muito maiores. Juntos, esses fatores permitiram o sucesso de COBOL, dentro e fora do DoD. Sua aparição levou à mecanização eletrônica da contabilidade, um avanço importante por qualquer medida.

A seguir está um exemplo de programa em COBOL. Ele lê um arquivo chamado `BAL-FWD-File`, que contém informação de inventário sobre certa coleção de itens. Entre outras coisas, cada registro inclui o número atual de itens (`BAL-ON-HAND`) e o ponto

de reposição (BAL-REORDER-POINT). O ponto de reposição é o número limite de itens para a solicitação de novos itens. O programa produz uma lista de itens que devem ser reorganizados como um arquivo chamado REORDER-LISTING.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PRODUCE-REORDER-LISTING.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. DEC-VAX.
OBJECT-COMPUTER. DEC-VAX.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT BAL-FWD-FILE ASSIGN TO READER.
    SELECT REORDER-LISTING ASSIGN TO LOCAL-PRINTER.
DATA DIVISION.
FILE SECTION.
FD  BAL-FWD-FILE
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 80 CHARACTERS.

01  BAL-FWD-CARD.
    02 BAL-ITEM-NO          PICTURE IS 9(5).
    02 BAL-ITEM-DESC        PICTURE IS X(20).
    02 FILLER               PICTURE IS X(5).
    02 BAL-UNIT-PRICE       PICTURE IS 999V99.
    02 BAL-REORDER-POINT    PICTURE IS 9(5).
    02 BAL-ON-HAND          PICTURE IS 9(5).
    02 BAL-ON-ORDER         PICTURE IS 9(5).
    02 FILLER               PICTURE IS X(30).
FD  REORDER-LISTING
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 132 CHARACTERS.

01  REORDER-LINE.
    02 RL-ITEM-NO           PICTURE IS Z(5).
    02 FILLER               PICTURE IS X(5).
    02 RL-ITEM-DESC         PICTURE IS X(20).
    02 FILLER               PICTURE IS X(5).
    02 RL-UNIT-PRICE        PICTURE IS ZZZ.99.
    02 FILLER               PICTURE IS X(5).
    02 RL-AVAILABLE-STOCK   PICTURE IS Z(5).
    02 FILLER               PICTURE IS X(5).
    02 RL-REORDER-POINT     PICTURE IS Z(5).
    02 FILLER               PICTURE IS X(71).

WORKING-STORAGE SECTION.
01  SWITCHES.
    02 CARD-EOF-SWITCH      PICTURE IS X.
01  WORK-FIELDS.
    02 AVAILABLE-STOCK      PICTURE IS 9(5).
```

```
PROCEDURE DIVISION.
000-PRODUCE-REORDER-LISTING.
    OPEN INPUT BAL-FWD-FILE.
    OPEN OUTPUT REORDER-LISTING.
    MOVE "N" TO CARD-EOF-SWITCH.
    PERFORM 100-PRODUCE-REORDER-LINE
        UNTIL CARD-EOF-SWITCH IS EQUAL TO 'Y".
    CLOSE BAL-FWD-File.
    CLOSE REORDER-LISTING.
    STOP RUN.

100-PRODUCE-REORDER-LINE.
    PERFORM 110-READ-INVENTORY-RECORD.
    IF CARD-EOF-SWITCH IS NOT EQUAL TO "Y"]
        PERFORM 120-CALCULATE-AVAILABLE-STOCK
        IF AVAILABLE-STOCK IS LESS THAN BAL-REORDER-POINT
            PERFORM 130-PRINT-REORDER-LINE.

110-READ-INVENTORY-RECORD.
    READ BAL-FWD-FILE RECORD
        AT END
            MOVE "Y" TO CARD-EOF-SWITCH.

120-CALCULATE-AVAILABLE-STOCK.
ADD BAL-ON-HAND BAL-ON-ORDER
    GIVING AVAILABLE-STOCK.

130-PRINT-REORDER-LINE.
    MOVE SPACE               TO REORDER-LINE.
    MOVE BAL-ITEM-NO         TO RL-ITEM-NO.
    MOVE BAL-ITEM-DESC       TO RL-ITEM-DESC.
    MOVE BAL-UNIT-PRICE      TO RL-UNIT-PRICE.
    MOVE AVAILABLE-STOCK     TO RL-AVAILABLE-STOCK.
    MOVE BAL-REORDER-POINT   TO RL-REORDER-POINT.
    WRITE REORDER-LINE.
```

2.7 O INÍCIO DO COMPARTILHAMENTO DE TEMPO: BASIC

Basic (Mather e Waite, 1971) é outra linguagem de programação que teve amplo uso, mas pouco reconhecimento. Como COBOL, foi ignorada pelos cientistas da computação. Além disso, como COBOL, em suas primeiras versões a linguagem era deselegante e incluía apenas um conjunto deficiente de sentenças de controle.

No final dos anos 1970, início dos anos 1980, Basic era muito popular em microcomputadores, resultado direto de duas das principais características das suas versões iniciais. Era de fácil aprendizado para iniciantes, especialmente para os leigos em in-

formática, e seus dialetos menores podiam ser implementados em computadores com memórias muito pequenas.[6] Quando a capacidade dos microcomputadores cresceu e outras linguagens foram implementadas, o uso do Basic diminuiu. Uma forte revitalização no seu uso surgiu com a aparição do Visual Basic (Microsoft, 1991), no início dos anos 1990.

2.7.1 Processo de projeto

Basic (Beginner's All-purpose Symbolic Instruction Code) foi originalmente projetado na Faculdade de Dartmouth (agora Universidade de Dartmouth), em New Hampshire, por dois matemáticos, John Kemeny e Thomas Kurtz, que, no início dos anos 1960, desenvolveram compiladores para uma variedade de dialetos de Fortran e ALGOL 60. Seus estudantes de ciências básicas tinham pouca dificuldade para aprender ou usar essas linguagens em seus estudos. No entanto, Dartmouth era primariamente uma instituição da área de humanas, na qual os estudantes de ciências básicas e engenharia eram apenas cerca de 25%. Na primavera de 1963, foi decidido que seria projetada uma linguagem especialmente voltada para os estudantes da área de humanas. Essa nova linguagem usaria terminais como o método de acesso a computadores. Os objetivos do sistema eram os seguintes:

1. Deveria ser fácil para estudantes que não eram de ciências básicas aprender e usar.
2. Deveria ser "prazerosa e amigável".
3. Deveria agilizar os deveres de casa.
4. Deveria permitir acesso livre e privado.
5. Deveria considerar o tempo do usuário mais importante que o tempo do computador.

O último objetivo era um conceito revolucionário. Ele era baseado, ao menos parcialmente, na crença de que os computadores se tornariam mais baratos com o passar do tempo, o que aconteceu.

A combinação do segundo, terceiro e quarto objetivos levou ao aspecto de compartilhamento de tempo de Basic. Somente com acesso individual, por terminais usados por numerosos usuários simultaneamente, esses objetivos poderiam ser alcançados no início dos anos 1960.

No verão de 1963, Kemeny começou a trabalhar no compilador para a primeira versão de Basic, usando acesso remoto a um computador GE 225. O projeto e a codificação do sistema operacional para Basic começaram no final de 1963. Às 4 horas da manhã de 1º de maio de 1964, o primeiro programa usando Basic com compartilhamento de tempo foi digitado e executado. Em junho, o número de terminais no sistema aumentou para 11 – e, no fim do ano, para 20.

[6] Alguns microcomputadores antigos incluíam interpretadores Basic que residiam em 4.096 bytes de memória ROM.

2.7.2 Visão geral da linguagem

A versão original de Basic era muito pequena e, estranhamente, não era interativa: não havia nenhuma maneira de um programa em execução obter dados de entrada do usuário. Os programas eram digitados, compilados e executados de um modo parcialmente orientado a lotes. Basic original tinha apenas 14 tipos diferentes de sentenças e um único tipo de dados – ponto flutuante. Como se acreditava que poucos dos usuários-alvo iriam reconhecer a diferença entre tipos inteiros e de ponto flutuante, o tipo foi chamado de *numbers* (números). De modo geral, era uma linguagem muito limitada, apesar de bastante fácil de aprender.

2.7.3 Avaliação

O aspecto mais importante de Basic original foi o fato de ser a primeira linguagem amplamente adotada que era usada por meio de terminais conectados a um computador remoto.[7] Os terminais haviam recém-começado a ficar disponíveis na época. Antes disso, em sua maioria, os programas eram inseridos nos computadores por meio de cartões perfurados ou por fitas de papel.

Grande parte do projeto de Basic veio de Fortran, com alguma influência da sintaxe de ALGOL 60. Posteriormente, ele cresceu de diversas maneiras, com pouco ou nenhum esforço para padronizá-lo. O Instituto Nacional de Padrões dos Estados Unidos publicou um padrão mínimo para Basic, chamado de Minimal Basic (ANSI, 1978b), mas ele representava apenas o mínimo essencial dos recursos da linguagem. Na verdade, Basic original era bastante semelhante ao Minimal Basic.

Apesar de parecer surpreendente, a Digital Equipment Corporation (DEC) usou uma versão um tanto elaborada de Basic, chamada de Basic-PLUS, para escrever partes significativas de seu maior sistema operacional para os minicomputadores PDP-11, RSTS, nos anos 1970.

Basic foi criticado, entre outras coisas, pela estrutura deficiente dos programas escritos na linguagem. Pelo critério de avaliação discutido no Capítulo 1, mais especificamente em relação à legibilidade e à confiabilidade, a linguagem, de fato, é muito limitada. As primeiras versões não foram projetadas para serem usadas em programas sérios de qualquer tamanho significativo e não deveriam ter sido usadas para tal. As versões subsequentes eram bem mais adequadas para tais tarefas.

O ressurgimento de Basic nos anos 1990 foi motivado pela aparição do Visual Basic (VB). Em grande parte, o VB se tornou amplamente usado porque oferecia uma maneira simples de construir interfaces gráficas do usuário (GUIs) – daí o nome Visual Basic. Quando .NET surgiu, trouxe consigo uma nova versão de VB, VB.NET. Apesar de ter se distanciado das versões iniciais de VB, substituiu rapidamente a linguagem mais antiga. Talvez a diferença mais importante entre VB e a versão .NET é que a segunda suporta completamente a programação orientada a objetos.

[7] Inicialmente, Lisp foi usada por meio de terminais, mas não foi amplamente empregada no início dos anos 1960.

A seguir, temos um exemplo de um programa em Basic:

```
REM  Programa de exemplo de Basic
REM  Entrada:  Um inteiro, listlen, onde listlen é menor
REM            que 100, seguido por valores inteiros listlen
REM  Saída:    O número de valores de entrada que são maiores
REM            que a média de todos os valores de entrada
  DIM intlist(99)
  result = 0
  sum = 0
  INPUT listlen
  IF ((listlen > 0) AND (listlen < 100)) THEN
REM  Lê os dados de entrada em um vetor e calcula sua soma
    FOR counter = 1 TO listlen
      INPUT intlist(counter)
      sum = sum + intlist(counter)
    NEXT counter
REM  Calcula a média
    average = sum / listlen
REM  Conta o número de valores de entrada maiores que a média
    FOR counter = 1 TO listlen
      IF intlist(counter) > average
        THEN result = result + 1
    NEXT counter
REM  Imprimir o resultado
    PRINT "The number of values that are > average is:";
          result
  ELSE
    PRINT "Error-input list length is not legal"
  END IF
END
```

2.8 TUDO PARA TODOS: PL/I

PL/I representa a primeira tentativa em grande escala de projetar uma linguagem que pudesse ser usada por um amplo espectro de áreas de aplicação. Todas as linguagens anteriores e a maioria das subsequentes se concentravam em uma área de aplicação específica, como aplicações científicas, de inteligência artificial ou de negócios.

2.8.1 Perspectiva histórica

Como Fortran, PL/I foi desenvolvida como um produto da IBM. No início dos anos 1960, os usuários de computadores na indústria haviam se estabelecido em dois campos separados e bastante diferentes: aplicações científicas e aplicações de negócios. Do ponto de vista da IBM, os programadores científicos poderiam usar os computadores IBM 7090 de grande porte ou os 1620 de pequeno porte. Tais programadores utilizavam

Projeto de usuário e projeto de linguagens

ALAN COOPER

Autor do *best-seller About Face: The Essentials of User Interface Design*, Alan Cooper tem importante participação no projeto da que pode ser considerada a linguagem que mais se preocupa com o projeto de interface com o usuário, o Visual Basic. Para ele, tudo se resume ao objetivo de humanizar a tecnologia.

ALGUMAS INFORMAÇÕES BÁSICAS

Como você começou? Sou um desistente do ensino médio formado em tecnologia de programação em uma faculdade da Califórnia. Meu primeiro emprego foi como programador na America President Lines (uma das companhias mais antigas de transporte marítimo dos Estados Unidos), em San Francisco. Exceto por alguns meses aqui e ali, me mantive autônomo.

Qual é o seu trabalho atual? Fundador e presidente da Cooper, a empresa que humaniza a tecnologia (www.cooper.com).

Qual é ou foi seu trabalho favorito? Consultor de design de interação.

Você é muito conhecido nas áreas de projeto de linguagem e de interface do usuário. O que você pensa a respeito de projetar linguagens versus projetar software *versus* projetar qualquer outra coisa? É basicamente o mesmo no mundo do software: conheça seu usuário.

SOBRE AS VERSÕES INICIAIS DO WINDOWS

Nos anos 1980, você começou a usar o Windows e ficou entusiasmado com suas qualidades: o suporte para a interface gráfica do usuário e as bibliotecas ligadas dinamicamente (DLLs), que permitem a criação de ferramentas que se autoconfiguram. O que você tem a dizer sobre as partes do Windows que ajudou a dar forma? Fiquei bastante impressionado com a inclusão pela Microsoft do suporte para multitarefas no Windows. Isso incluía realocação dinâmica e comunicação interprocessos.
O MSDOS.exe era o programa de interpretação de comandos (*shell*) nos primeiros lançamentos do Windows. Era um programa terrível, e eu acreditava que poderia melhorá-lo muito. Em meu tempo livre, comecei a escrever um interpretador de comandos melhor. Eu o chamei de Tripod. O interpretador de comandos original da Microsoft, o MSDOS.exe, era um dos principais obstáculos para o sucesso inicial do Windows. O Tripod tentava resolver o problema sendo mais fácil de usar e configurar.

Quando você teve essa ideia? No final de 1987, quando eu estava entrevistando um cliente corporativo, a estratégia-chave de projeto para o Tripod surgiu em minha cabeça. Como o gerente de sistemas de informação explicou sua necessidade de criar e publicar uma ampla variedade de soluções de interpretação de comandos para sua base de dados heterogênea, eu me dei conta de que o problema era a falta de tal interpretador de comandos ideal. Cada usuário precisaria de seu interpretador de comandos pessoal, configurado para suas necessidades e no seu nível de conhecimento. Em um instante, percebi a solução para o problema do projeto do interpretador de comandos: deveria ser um conjunto de construção de interpretadores de comandos, uma ferramenta na qual cada usuário seria capaz de construir exatamente o interpretador necessário para um misto único de aplicações e de treinamento.

O que é tão atraente na ideia de um interpretador de comandos que pode ser personalizado? Em vez de dizer aos usuários qual interpretador de comandos é o ideal, eles podem projetar sua própria versão ideal e personalizada. Assim, um programador poderia criar um interpretador de comandos poderoso e abrangente, mas também um tanto perigoso, ao passo que um gerente de tecnologia da informação criaria um interpretador de comandos para dar a um atendente, expondo apenas as poucas ferramentas específicas que esse atendente usa.

Como você passou de escritor de um interpretador de comandos a colaborador da Microsoft? Tripod e Ruby são a mesma coisa. Depois de assinar um acordo com Bill Gates, troquei o nome do protótipo de Tripod para Ruby. Usei o protótipo de Ruby como os protótipos devem ser usados: um modelo descartável para construir código de alta qualidade. Foi isso que fiz. A Microsoft pegou a versão de distribuição de Ruby e adicionou QuickBasic a ela, criando VB. Todas aquelas inovações originais estavam em Tripod/Ruby.

RUBY COMO INCUBADORA PARA O VISUAL BASIC

"O MSDOS.exe era o programa de interpretação de comandos (shell) para os primeiros lançamentos do Windows. Era um programa terrível, e eu acreditava que poderia melhorá-lo drasticamente. Em meu tempo livre, comecei a escrever um melhor interpretador de comandos."

Fale sobre seu interesse nas primeiras versões do Windows e o tal recurso chamado DLL. As DLLs eram um recurso do sistema operacional. Elas permitiam que um programador construísse objetos de código para serem ligados em tempo de execução, em vez de somente em tempo de compilação. Isso me permitiu inventar as partes dinamicamente extensíveis do VB, em que os controles podem ser adicionados por terceiros.

O produto Ruby continha muitos avanços significativos em projeto de software, mas dois deles se destacam como excepcionalmente bem-sucedidos. Como mencionei, a capacidade de ligação dinâmica do Windows sempre me intrigou, mas ter as ferramentas e saber o que fazer com elas são coisas diferentes. Com Ruby, finalmente encontrei dois usos para a ligação dinâmica, e o programa original continha ambas. Primeiro, a linguagem era tanto instalável quanto poderia ser estendida. Segundo, a paleta de componentes poderia ser adicionada de maneira dinâmica.

A sua linguagem em Ruby foi a primeira a ter uma biblioteca de ligação dinâmica e ser ligada a um *front-end* visual? Pelo que sei, sim.

Usando um exemplo simples, o que isso permitia a um programador fazer com seu programa? Comprar um controle, como um controle de grade, de uma empresa qualquer, instalar em seu computador e ter o controle de grade como se fosse parte da linguagem, incluindo o *front-end* de programação visual.

Por que o chamam de "pai do Visual Basic"? Ruby vinha com uma pequena linguagem, voltada apenas para a execução de cerca de uma dúzia de comandos simples que um interpretador de comandos precisa. Entretanto, essa linguagem era implementada como uma cadeia de DLLs, as quais poderiam ser instaladas em tempo de execução. O analisador sintático interno identificaria um verbo e então o passaria para a cadeia de DLLs até uma delas confirmar que poderia processar o verbo. Se todas as DLLs passassem, havia um erro de sintaxe. A partir de nossas primeiras discussões, tanto a Microsoft quanto eu tínhamos gostado da ideia de aumentar a linguagem, possivelmente até mesmo substituindo-a por uma linguagem "real". C era a candidata mais mencionada, mas a Microsoft aproveitou essa interface dinâmica para substituir nossa pequena linguagem de interpretação de comandos pelo Quick-Basic. Esse novo casamento de linguagem com um *front-end* visual era estático e permanente, e, apesar de a interface dinâmica original ter possibilitado o casamento, ela foi perdida no processo.

COMENTÁRIOS FINAIS SOBRE NOVAS IDEIAS

No mundo da programação e das ferramentas de programação, incluindo linguagens e ambientes, que projetos mais lhe interessam? Tenho interesse em projetar ferramentas de programação para ajudar os usuários e não os programadores.

Que regra, citação famosa ou ideia de projeto devemos sempre manter em mente? Pontes não são construídas por engenheiros. São construídas por serralheiros.

Da mesma forma, programas de software não são construídos por engenheiros. São construídos por programadores.

dados em formato de ponto flutuante e vetores extensivamente. Fortran era a principal linguagem, apesar de alguma linguagem de montagem também ser usada. Eles tinham seu próprio grupo de usuários, chamado SHARE, e pouco contato com qualquer um que trabalhasse em aplicações de negócios.

Para aplicações de negócios, as pessoas usavam os computadores da IBM de grande porte 7080 ou os de pequeno porte 1401. Elas precisavam de tipos de dados decimais e de cadeias de caracteres, assim como de recursos elaborados e eficientes para entrada e saída. Além disso, usavam COBOL, apesar de, no início de 1963, quando a história de PL/I começou, a conversão de linguagem de montagem para COBOL não estar concluída. Essa categoria de usuários também tinha seu próprio grupo de usuários, chamado GUIDE, e raramente mantinha contato com usuários científicos.

No início de 1963, os planejadores da IBM perceberam uma mudança nessa situação. Os dois grupos amplamente separados estavam se aproximando um do outro, o que se pensava que causaria problemas. Os cientistas começaram a obter grandes arquivos de dados para serem processados. Esses dados exigiam recursos de entrada e saída mais sofisticados e eficientes. Os profissionais das aplicações de negócios começaram a usar análise de regressão para construir sistemas de administração de informações, os quais exigiam dados de ponto flutuante e vetores. Começou a ficar claro que as instalações de computação logo necessitariam de dois computadores e pessoal técnico separados, dando suporte a duas linguagens de programação muito diferentes.[8]

Como seria de se esperar, essas percepções levaram ao conceito de projetar um computador universal capaz de realizar tanto operações de ponto flutuante quanto aritmética decimal – e, dessa forma, suportar tanto aplicações científicas quanto comerciais. Nasceu então o conceito da linha de computadores IBM System/360. Com isso, veio a ideia de uma linguagem de programação que pudesse ser usada tanto para aplicações comerciais quanto para aplicações científicas. Para atender a tais aplicações foram adicionados recursos para suportar a programação de sistemas e para o processamento de listas. Assim, a nova linguagem viria para substituir Fortran, COBOL, Lisp e as aplicações de sistemas das linguagens de montagem.

2.8.2 Processo de projeto

O esforço de projeto começou quando a IBM e o SHARE formaram o Comitê de Desenvolvimento de Linguagem Avançada do Projeto Fortran do SHARE, em outubro de 1963. Esse novo comitê rapidamente se encontrou e formou um subcomitê denominado Comitê 3 x 3, assim chamado porque era formado por três membros da IBM e três do SHARE. O Comitê 3 x 3 se encontrava três ou quatro dias, semana sim, semana não, para projetar a linguagem.

Como ocorreu com o Short Range Committee para COBOL, o projeto inicial foi planejado para ser concluído em um tempo excepcionalmente curto. Aparentemente, independentemente da abrangência de um esforço de projeto de linguagem, no início dos anos 1960, acreditava-se que ele poderia ser feito em três meses. A primeira versão de PL/I, chamada de Fortran VI, supostamente deveria estar concluída em dezembro, menos de três meses após a formação do comitê. Ele obteve extensões de prazo em duas

[8]Na época, as grandes instalações de computador exigiam pessoal de manutenção de hardware e de software de sistema em tempo integral.

ocasiões, mudando a data de finalização para janeiro e, posteriormente, para o final de fevereiro de 1964.

O conceito inicial de projeto definia que a nova linguagem seria uma extensão de Fortran IV, mantendo a compatibilidade. Mas esse objetivo foi abandonado rapidamente, assim como o nome Fortran VI. Até 1965, a linguagem era conhecida como NPL (New Programming Language). O primeiro relatório publicado sobre a NPL foi apresentado na reunião do grupo SHARE em março de 1964. Uma descrição mais completa foi apresentada em abril, e a versão que seria implementada foi publicada em dezembro de 1964 (IBM, 1964) pelo grupo de compiladores no Laboratório Hursley da IBM na Inglaterra, escolhido para a implementação. Em 1965, o nome foi trocado para PL/I para evitar a confusão do nome NPL com o National Physical Laboratory da Inglaterra. Se o compilador tivesse sido desenvolvido fora do Reino Unido, o nome poderia ter permanecido NPL.

2.8.3 Visão geral da linguagem

Talvez a melhor descrição em uma única sentença da linguagem PL/I é que ela incluía o que eram consideradas as melhores partes de ALGOL 60 (recursão e estrutura de bloco), Fortran IV (compilação separada com comunicação por meio de dados globais) e COBOL 60 (estruturas de dados, entrada e saída e recursos para a geração de relatórios), além de uma extensa coleção de novas construções, todas unidas de maneira improvisada. Como PL/I não é mais uma linguagem popular, não tentaremos, mesmo que brevemente, discutir todos os seus recursos ou mesmo suas construções mais controversas. Em vez disso, vamos mencionar algumas das contribuições aos conhecimentos acerca de linguagens de programação.

PL/I foi a primeira linguagem a ter os seguintes recursos:

- Os programas podiam criar subprogramas de execução concorrente. Embora isso fosse uma boa ideia, ela foi mal desenvolvida em PL/I.
- Era possível detectar e manipular 23 tipos diferentes de exceções ou erros em tempo de execução.
- Era permitida a utilização de subprogramas recursivamente, mas tal mecanismo podia ser desabilitado, o que permitia uma ligação mais eficiente para subprogramas não recursivos.
- Ponteiros foram incluídos como um tipo de dados.
- Porções de uma matriz podiam ser referenciadas. Por exemplo, a terceira linha de uma matriz poderia ser referenciada como se fosse um vetor unidimensional.

2.8.4 Avaliação

Quaisquer avaliações de PL/I devem começar reconhecendo a ambição do esforço de projeto. Retrospectivamente, parece ingenuidade pensar que tantas construções poderiam ser combinadas com sucesso. Entretanto, tal julgamento deve levar em consideração que havia pouca experiência em projeto de linguagens na época. De modo geral, o projeto de PL/I era baseado na premissa de que qualquer construção útil que pudesse ser implementada deveria ser incluída, com poucas preocupações sobre como um pro-

gramador poderia entender e usar efetivamente tal coleção de construções e recursos. Edsger Dijkstra, em sua Palestra do Prêmio Turing (Dijkstra, 1972), fez uma das críticas mais contundentes a respeito da complexidade de PL/I: "Eu não consigo entender como poderemos manter o crescimento de nossos programas de maneira firme, dentro de nossas capacidades intelectuais, quando simplesmente a complexidade e a irregularidade da linguagem de programação – nossa ferramenta básica, vejam só – já fogem de nosso controle intelectual".

Além de enfrentar o problema da complexidade por conta de seu tamanho, PL/I ainda incluía diversas construções que atualmente são consideradas deficientes. Entre essas estavam os ponteiros, o tratamento de exceções e a concorrência, embora devamos dizer que, em todos os casos, essas construções não haviam aparecido em nenhuma linguagem anterior.

Em termos de uso, PL/I deve ser considerada bem-sucedida, ao menos parcialmente. Nos anos 1970, ela desfrutou de um uso significativo, tanto em aplicações comerciais quanto científicas. Ela também foi bastante usada como ferramenta de ensino em faculdades, principalmente em formatos que eram subconjuntos da linguagem, como PL/C (Cornell, 1977) e PL/CS (Conway e Constable, 1976).

A seguir, temos um exemplo de um programa em PL/I:

```
/* PROGRAMA DE EXEMPLO DE PL/I
 ENTRADA:   UM INTEIRO, LISTLEN, ONDE LISTLEN É MENOR QUE
            100, SEGUIDO POR VALORES INTEIROS LISTLEN.
 SAÍDA:     O NÚMERO DE VALORES DE ENTRADA QUE SÃO MAIORES
            QUE A MÉDIA DE TODOS OS VALORES DE ENTRADA */
PLIEX: PROCEDURE OPTIONS (MAIN);
  DECLARE INTLIST (1:99) FIXED.
  DECLARE (LISTLEN, COUNTER, SUM, AVERAGE, RESULT) FIXED;
  SUM = 0;
  RESULT = 0;
  GET LIST (LISTLEN);
  IF (LISTLEN > 0) & (LISTLEN < 100) THEN
    DO;
/* LÊ OS DADOS DE ENTRADA EM UM VETOR E CALCULA SUA SOMA */
    DO COUNTER = 1 TO LISTLEN;
      GET LIST (INTLIST (COUNTER));
      SUM = SUM + INTLIST (COUNTER);
    END;
/* CALCULA A MÉDIA */
    AVERAGE = SUM / LISTLEN;
/* CONTA O NÚMERO DE VALORES QUE SÃO MAIORES QUE A MÉDIA */
    DO COUNTER = 1 TO LISTLEN;
      IF INTLIST (COUNTER) > AVERAGE THEN
        RESULT = RESULT + 1;
    END;
/* IMPRIMIR O RESULTADO */
    PUT SKIP LIST ('THE NUMBER OF VALUES > AVERAGE IS:');
    PUT LIST (RESULT);
    END;
```

```
ELSE
    PUT SKIP LIST ('ERROR-INPUT LIST LENGTH IS ILLEGAL');
END PLIEX;
```

2.9 DUAS DAS PRIMEIRAS LINGUAGENS DINÂMICAS: APL E SNOBOL

A estrutura desta seção é diferente das outras, pois as linguagens discutidas aqui são muito diferentes. Nem APL nem SNOBOL tiveram muita influência sobre as principais linguagens posteriores.[9] Alguns dos recursos interessantes da APL são discutidos mais adiante.

Tanto em relação à aparência quanto ao propósito, APL e SNOBOL são bastante diferentes. Elas compartilham, entretanto, duas características fundamentais: tipagem dinâmica e alocação dinâmica de armazenamento. Nas duas linguagens as variáveis são essencialmente não tipadas. Uma variável adquire um tipo quando lhe atribuem um valor, ou seja, quando ela assume o tipo desse valor. O armazenamento, por sua vez, é alocado a uma variável apenas quando lhe é atribuído um valor, já que antes disso não existe uma maneira de saber a quantidade de armazenamento necessário.

2.9.1 Origens e características de APL

APL (Brown et al., 1988) foi projetada por volta de 1960 por Kenneth E. Iverson, na IBM. Ela não foi originalmente projetada para ser uma linguagem de programação implementada, mas um veículo para descrever arquiteturas de computadores. APL foi descrita inicialmente no livro do qual obtém seu nome, *A Programming Language* (Iverson, 1962). Em meados dos anos 1960, a primeira implementação de APL foi desenvolvida na IBM.

APL tem um grande número de operadores poderosos, especificados com um grande número de símbolos, os quais criaram um problema para os implementadores. Inicialmente, APL foi usada por meio de terminais de impressão IBM. Tais terminais tinham elementos de impressão especiais e opcionais que forneciam o estranho conjunto de caracteres exigido pela linguagem. Uma das razões pelas quais APL tem tantos operadores é que ela fornece um grande número de operações unitárias em vetores. Por exemplo, a transposição de qualquer matriz é feita com um único operador. A grande coleção de operadores oferece alta expressividade, mas também faz os programas APL serem de difícil leitura. Portanto, as pessoas consideraram APL uma linguagem mais bem usada para programação "descartável". Apesar de os programas serem escritos rapidamente, eles devem ser descartados após o uso, porque são de difícil manutenção.

APL existe há 50 anos e ainda está sendo usada, embora não amplamente. Além disso, ela não mudou muito durante sua vida.

[9] Contudo, elas tiveram certa influência sobre algumas linguagens (J é baseada em APL, ICON é baseada em SNOBOL e AWK é parcialmente baseada em SNOBOL).

2.9.2 Origens e características de SNOBOL

SNOBOL (pronuncia-se "snowball"; Griswold et al., 1971) foi projetada no início dos anos 1960 por três pessoas no Laboratório Bell: D. J. Farber, R. E. Griswold e I. P. Polonsky (Farber et al., 1964). Ela foi projetada especificamente para processamento de texto. O centro de SNOBOL é uma coleção de operações poderosas para o casamento de padrões de cadeias. Uma de suas primeiras aplicações foi a escrita de editores de texto. Como a natureza dinâmica de SNOBOL a torna mais lenta que as linguagens alternativas, ela não é mais usada para tais programas. Entretanto, ainda é uma linguagem viva e com suporte, usada para uma variedade de tarefas de processamento de textos em diferentes áreas de aplicação.

2.10 O COMEÇO DA ABSTRAÇÃO DE DADOS: SIMULA 67

Apesar de SIMULA 67 nunca ter atingido um amplo uso e ter tido pouco impacto nos programadores e na computação de sua época, algumas das construções que introduziu a tornam historicamente importante.

2.10.1 Processo de projeto

Dois noruegueses, Kristen Nygaard e Ole-Johan Dahl, desenvolveram a linguagem SIMULA I entre 1962 e 1964, no Centro de Computação Norueguês (NCC), em Oslo. Eles estavam inicialmente interessados em usar computadores para simulação, mas também trabalhavam em pesquisa operacional. SIMULA I foi projetada exclusivamente para a simulação de sistemas e implementada pela primeira vez no final de 1964 em um computador UNIVAC 1107.

Quando a implementação de SIMULA I estava concluída, Nygaard e Dahl começaram os esforços para estender a linguagem, adicionando novos recursos e modificando algumas das construções existentes para torná-la útil para aplicações de propósito geral. O resultado desse trabalho foi SIMULA 67, cujo projeto foi apresentado publicamente pela primeira vez em março de 1967 (Dahl e Nygaard, 1967). Discutiremos apenas SIMULA 67, apesar de alguns de seus recursos interessantes também estarem disponíveis em SIMULA I.

2.10.2 Visão geral da linguagem

SIMULA 67 é uma extensão de ALGOL 60, com sua estrutura de bloco e suas sentenças de controle. A principal deficiência de ALGOL 60 (e de outras linguagens da época) para aplicações de simulação era o projeto de seus subprogramas. As simulações precisam de subprogramas que possam ser reiniciados na posição em que foram previamente parados. Subprogramas com esse tipo de controle são denominados **corrotinas**, porque o chamador e os subprogramas chamados têm um relacionamento de certa forma igualitário, em vez do mestre/escravo rígido existente na maioria das linguagens imperativas.

Para oferecer suporte a corrotinas em SIMULA 67, foi desenvolvida a construção de classes. Essa foi uma evolução importante, pois o conceito de abstração de dados

começou com ela, e a abstração de dados fornece a base da programação orientada a objetos.

É interessante notar que o importante conceito de abstração de dados não foi desenvolvido e atribuído à classe `construct` até 1972, quando Hoare (1972) reconheceu a conexão.

2.11 PROJETO ORTOGONAL: ALGOL 68

ALGOL 68 foi fonte de diversas ideias novas no projeto de linguagens, algumas adotadas por outras linguagens. Incluímo-lo aqui por essa razão, apesar de ele nunca ter atingido um uso amplo nem na Europa, nem nos Estados Unidos.

2.11.1 Processo de projeto

O desenvolvimento da família ALGOL não terminou quando o relatório revisado sobre ALGOL 60 apareceu em 1962, apesar de demorar seis anos até que a iteração de projeto seguinte fosse publicada. A linguagem resultante, ALGOL 68 (van Wijngaarden et al.,1969), era drasticamente diferente de suas predecessoras.

Uma das inovações mais interessantes da ALGOL 68 foi em relação a um de seus critérios de projeto primários: ortogonalidade. Lembre-se de nossa discussão sobre ortogonalidade, no Capítulo 1. O uso da ortogonalidade resultou em diversos recursos inovadores em ALGOL 68, um deles descrito na seção seguinte.

2.11.2 Visão geral da linguagem

Um resultado importante da ortogonalidade em ALGOL 68 foi a inclusão dos tipos de dados definidos pelo usuário. Linguagens anteriores, como Fortran, incluíam apenas algumas estruturas de dados básicas. PL/I incluiu um número maior de estruturas de dados (o que a tornou mais difícil de aprender e de implementar), mas ela ainda não podia fornecer uma estrutura de dados apropriada para cada necessidade.

A estratégia de ALGOL 68 para as estruturas de dados era fornecer alguns tipos primitivos e permitir que o usuário os combinasse para definir um grande número de estruturas. Esse recurso de fornecer tipos de dados definidos pelo usuário foi usado por todas as principais linguagens imperativas projetadas desde então. Os tipos de dados definidos pelo usuário são valiosos, porque permitem projetar abstrações de dados que se encaixem nos problemas em particular de uma maneira muito consistente. Todos os aspectos dos tipos de dados são discutidos no Capítulo 6.

ALGOL 68 também introduziu vetores dinâmicos que serão chamados de *implícitos dinâmicos do monte* no Capítulo 5. Vetor dinâmico é aquele no qual a declaração não especifica os limites dos índices. Atribuições a um vetor dinâmico fazem com que o armazenamento necessário seja alocado em tempo de execução. Em ALGOL 68, os vetores dinâmicos são chamados de vetores **`flex`**.

2.11.3 Avaliação

ALGOL 68 inclui um número significativo de recursos. Seu uso da ortogonalidade, que alguns podem considerar excessivo, foi revolucionário.

Contudo, ele repetiu um dos erros da ALGOL 60, e esse foi um fator importante para sua popularidade limitada. A linguagem foi descrita com uma metalinguagem elegante e concisa, porém desconhecida. Antes que alguém pudesse ler o documento que descrevia a linguagem (van Wijngaarden et al., 1969), tinha de aprender a nova metalinguagem, chamada de gramáticas de van Wijngaarden, que eram bem mais complexas que a BNF. Para piorar, os projetistas inventaram uma coleção de palavras para explicar a gramática e a linguagem. Por exemplo, as palavras-chave eram chamadas de *indicativos*, a extração de subcadeias era chamada de *redução* e o processo de execução de um subprograma era chamado de *coerção de desprocedimento*, a qual poderia ser *obediente*, *firme* ou alguma outra coisa.

É natural avaliar o contraste do projeto de PL/I com o de ALGOL 68, pois surgiram com poucos anos de diferença. ALGOL 68 conseguiu ter uma boa facilidade de escrita por meio do princípio da ortogonalidade: alguns conceitos primitivos e o uso irrestrito de alguns mecanismos de combinação. PL/I conseguiu ter uma boa facilidade de escrita com a inclusão de um grande número de construções fixas. ALGOL 68 estendeu a simplicidade elegante de ALGOL 60, enquanto PL/I simplesmente reuniu os recursos de diversas linguagens para atingir seus objetivos. É claro, precisamos lembrar que o objetivo da linguagem PL/I era fornecer uma ferramenta unificada para uma ampla classe de problemas, enquanto ALGOL 68 se concentrava em uma classe: aplicações científicas. PL/I atingiu uma aceitação muito maior que ALGOL 68, principalmente pelos esforços promocionais da IBM e pelos problemas de entendimento e de implementação da ALGOL 68. A implementação era difícil para ambas as linguagens, mas PL/I tinha os recursos da IBM para aplicar na construção de um compilador. ALGOL 68 não desfrutava de tal benefício.

2.12 ALGUNS DOS PRIMEIROS DESCENDENTES DE ALGOLS

Todas as linguagens imperativas devem algo de seu projeto a ALGOL 60 e/ou a ALGOL 68. Esta seção discute alguns dos primeiros descendentes dessas linguagens.

2.12.1 Simplicidade por meio do projeto: Pascal

2.12.1.1 Perspectiva histórica

Niklaus Wirth (Wirth é pronunciado "Virt") era membro do Grupo de Trabalho 2.1 da IFIP (International Federation of Information Processing – Federação Internacional de Processamento de Informações), criado para continuar o desenvolvimento de ALGOL, em meados dos anos 1960. Em agosto de 1965, Wirth e C. A. R. ("Tony") Hoare contribuíram para esse esforço ao apresentar ao grupo uma proposta modesta de adições e modificações a ALGOL 60 (Wirth e Hoare, 1966). A maioria do grupo rejeitou a proposta como um aprimoramento muito pequeno em relação a ALGOL 60. Em vez disso, uma revisão muito mais complexa foi realizada – o que, no fim das contas, tornou-se ALGOL 68. Wirth, assim como outros membros desse grupo, não acreditava que o relatório de

ALGOL 68 deveria ser publicado, devido à complexidade tanto da linguagem quanto da metalinguagem usada para descrevê-la. Tal posicionamento se mostrou válido, já que os documentos de ALGOL 68 (e, dessa forma, a linguagem) foram tidos como desafiadores para a comunidade de computação.

A versão de Wirth e Hoare de ALGOL 60 foi denominada ALGOL-W. Ela foi implementada na Universidade de Stanford e usada basicamente como uma ferramenta educacional, mas apenas em algumas universidades. As principais contribuições de ALGOL-W foram o método de passagem de parâmetros por valor-resultado e a sentença **`case`** para seleção múltipla. O método valor-resultado é uma alternativa ao método por nome de ALGOL 60. Ambos são discutidos no Capítulo 9. A sentença **`case`** é discutida no Capítulo 8.

O esforço de projeto seguinte de Wirth, novamente baseado em ALGOL 60, foi seu maior sucesso: Pascal.[10] A definição de Pascal publicada originalmente apareceu em 1971 (Wirth, 1971). Essa versão foi modificada no processo de implementação e é descrita em Wirth (1973). Os recursos geralmente creditados a Pascal vieram de linguagens anteriores. Por exemplo, os tipos de dados definidos pelo usuário foram introduzidos em ALGOL 68, a sentença **`case`** em ALGOL-W, e os registros de Pascal são similares aos da COBOL e da linguagem PL/I.

2.12.1.2 Avaliação

O maior impacto de Pascal se deu no ensino de programação. Em 1970, a maioria dos estudantes de ciência da computação, engenharia e ciências básicas começava seus estudos em programação com Fortran, apesar de algumas universidades usarem PL/I, linguagens baseadas em PL/I e ALGOL-W. Em meados dos anos 1970, Pascal se tornou a linguagem mais usada para esse propósito. Isso era bastante natural, já que ele foi especificamente projetado para ensinar programação. Só depois dos anos 1990 Pascal deixou de ser a linguagem mais usada para o ensino de programação em faculdades e universidades.

Pascal foi projetada como uma linguagem de ensino, portanto não tinha diversos recursos essenciais para muitos tipos de aplicação. O melhor exemplo disso é a impossibilidade de escrever um subprograma que recebe como parâmetro um vetor de tamanho variável. Outro exemplo é a falta de quaisquer capacidades de compilação de arquivos separados. Essas deficiências levaram a muitos dialetos não padronizados, como o Turbo Pascal.

A popularidade de Pascal, tanto para o ensino de programação quanto para outras aplicações, era baseada em sua excepcional combinação de simplicidade com expressividade. Apesar de existirem algumas inseguranças na linguagem, Pascal é, mesmo assim, relativamente segura, especialmente quando comparada com Fortran ou C. Em meado dos anos 1990, a popularidade de Pascal entrou em declínio, tanto na indústria quanto nas universidades, principalmente devido à escalada de Modula-2, Ada e C++, todas as quais incluíam recursos que não estavam disponíveis nela.

A seguir, temos um exemplo de um programa em Pascal:

[10] Pascal foi assim chamada em homenagem a Blaise Pascal, filósofo e matemático francês do século XVII, que inventou a primeira máquina de somar mecânica, em 1642 (entre outras coisas).

```
{Programa de exemplo em Pascal
 Entrada:   Um inteiro, listlen, onde listlen é menor que
            100, seguido por valores inteiros listlen
 Saída:     O número de valores de entrada que são maiores
            que a média de todos os valores de entrada }
program pasex (input, output);
  type intlisttype = array [1..99] of integer;
  var
    intlist : intlisttype;
    listlen, counter, sum, average, result : integer;
  begin
  result := 0;
  sum := 0;
  readln (listlen);
  if ((listlen > 0) and (listlen < 100)) then
    begin
{ Lê os dados de entrada em um vetor e calcula sua soma }
  for counter := 1 to listlen do
    begin
    readln (intlist[counter]);
    sum := sum + intlist[counter]
    end;
{ Calcula a média }
    average := sum / listlen;
{ Conta o número de valores de entrada que são maiores que a média }
    for counter := 1 to listlen do
      if (intlist[counter] > average) then
        result := result + 1;
{ Imprimir o resultado }
    writeln ('The number of values > average is:',
             result)
    end { of the then clause of if (( listlen > 0 ... }
  else
    writeln ('Error-input list length is not legal')
end.
```

2.12.2 Uma linguagem de sistemas portável: C

Assim como Pascal, C contribuiu pouco para a coleção de recursos de linguagem conhecidos anteriormente, mas é usada há longo tempo. Apesar de originalmente projetada para a programação de sistemas, a linguagem C é bastante adequada para uma variedade de aplicações.

2.12.2.1 Perspectiva histórica

Os ancestrais de C incluem CPL, BCPL, B e ALGOL 68. CPL foi desenvolvida na Universidade de Cambridge, no início dos anos 1960. BCPL é uma linguagem de sistemas

simples, também desenvolvida em Cambridge, desta vez por Martin Richards, em 1967 (Richards, 1969).

O primeiro trabalho no sistema operacional UNIX foi feito no fim dos anos 1960 por Ken Thompson, nos Laboratórios da Bell (Bell Labs). A primeira versão foi escrita em linguagem de montagem. A primeira linguagem de alto nível implementada no UNIX foi B, que era baseada em BCPL. B foi projetada e implementada por Thompson em 1970.

Nem BCPL nem B são linguagens tipadas, o que é estranho entre as linguagens de alto nível, apesar de ambas serem de muito mais baixo nível do que Java, por exemplo. Ser não tipada significa que todos os dados são considerados palavras de máquina, o que leva a muitas complicações e inseguranças. Por exemplo, existe o problema de especificar pontos flutuantes em vez de aritméticas de inteiros em uma expressão. Em uma implementação de BCPL, as variáveis que atuavam como operandos de uma operação de ponto flutuante eram precedidas por pontos. Variáveis que eram usadas como operandos não precedidas por pontos eram consideradas inteiros. Uma alternativa a isso seria o uso de símbolos diferentes para os operadores de ponto flutuante.

Esse e diversos outros problemas levaram ao desenvolvimento de uma nova linguagem tipada, baseada em B. Originalmente chamada de NB, mas posteriormente denominada C, ela foi projetada e implementada por Dennis Ritchie no Bell Labs, em 1972 (Kernighan e Ritchie, 1978). Em alguns casos por meio de BCPL, e em outros diretamente, a linguagem C foi influenciada pela ALGOL 68. Isso é visto em suas sentenças **`for`** e **`switch`**, em seus operadores de atribuição e em seu tratamento de ponteiros.

O único "padrão" para C em sua primeira década e meia foi o livro de Kernighan e Ritchie (1978).[11] Ao longo desse período, a linguagem evoluiu lentamente, com diferentes implementadores adicionando recursos. Em 1989, a ANSI produziu uma descrição oficial de C (ANSI, 1898), incluindo muitos dos recursos que os implementadores haviam incorporado à linguagem. Esse padrão foi atualizado em 1999 (ISO, 1999). Essa versão posterior inclui algumas alterações significativas na linguagem. Entre elas estão um tipo de dados complexo, um tipo de dados booleano e comentários no estilo de C++- (//). Vamos nos referir à versão de 1989, chamada por muito tempo de ANSI C, como C89, e à versão de 1999 como C99.

2.12.2.2 Avaliação

A linguagem C tem sentenças de controle adequadas e recursos para a utilização de estruturas de dados que permitem seu uso em muitas áreas de aplicação. Ela também tem um rico conjunto de operadores que fornecem alto grau de expressividade.

Uma das principais razões pelas quais a linguagem C é tão admirada como odiada é sua falta de uma verificação de tipos completa. Por exemplo, em versões anteriores ao C99, podiam ser escritas funções para as quais os parâmetros não eram verificados em relação ao tipo. Aqueles que gostam de C apreciam a flexibilidade; aqueles que não gostam o acham muito inseguro. Uma das grandes razões para o grande aumento da sua

[11] Essa linguagem é muitas vezes referida como "K & R C".

popularidade nos anos 1980 foi o fato de que um compilador para a linguagem era parte do amplamente usado sistema operacional UNIX. Essa inclusão no UNIX forneceu um compilador bastante bom e essencialmente livre, que estava disponível para os programadores em muitos tipos de computadores.

A seguir, temos um exemplo de um programa em C:

```
/* Programa de exemplo em C
 Entrada:  Um inteiro, listlen, onde listlen é menor que
           100, seguido por valores inteiros listlen
 Saída:    O número de valores de entrada que são maiores
           que a média de todos os valores de entrada */
int main (){
  int intlist[99], listlen, counter, sum, average, result;
  sum = 0;
  result = 0;
  scanf("%d", &listlen);
  if ((listlen > 0) && (listlen < 100)) {
/* Lê os dados de entrada em um vetor e calcula sua soma */
    for (counter = 0; counter < listlen; counter++) {
      scanf("%d", &intlist[counter]);
      sum += intlist[counter];
    }
/* Calcula a média */
    average = sum / listlen;
/* Conta o número de valores de entrada que são maiores que a média */
    for (counter = 0; counter < listlen; counter++)
      if (intlist[counter] > average) result++;
/* Imprimir o resultado */
    printf("Number of values > average is:%d\n", result);
  }
  else
    printf("Error-input list length is not legal\n");
}
```

2.13 PROGRAMAÇÃO BASEADA EM LÓGICA: PROLOG

Em síntese, a programação lógica é o uso de uma notação lógica formal para comunicar processos computacionais a um computador. O cálculo de predicados é a notação usada nas linguagens de programação lógica atuais.

A programação em linguagens de programação lógica é não procedural. Os programas não exprimem exatamente *como* um resultado deve ser computado, mas descrevem a forma necessária e/ou as características dele. O que é preciso para fornecer essa capacidade em linguagens de programação lógica é uma maneira concisa de disponibilizar ao computador tanto a informação relevante quanto um processo de inferência para computar os resultados desejados. O cálculo de predicados fornece a forma básica de comunicação com o computador. E o método de prova, chamado **resolução**, desenvolvido inicialmente por Robinson (1965), fornece a técnica de inferência.

2.13.1 Processo de projeto

Durante o início dos anos 1970, Alain Colmerauer e Phillippe Roussel, do Grupo de Inteligência Artificial da Universidade de Aix-Marseille, com Robert Kowalski, do Departamento de Inteligência Artificial da Universidade de Edimburgo, desenvolveram o projeto fundamental de Prolog. Os componentes primários de Prolog são um método para a especificação de proposições de cálculo de predicados e uma implementação de uma forma restrita de resolução. Tanto o cálculo de predicados quanto a resolução são descritos no Capítulo 16. O primeiro interpretador Prolog foi desenvolvido em Marselha, em 1972. A versão da linguagem que foi implementada é descrita em Roussel (1975). O nome Prolog vem de *programming logic* (programação lógica).

2.13.2 Visão geral da linguagem

Os programas em Prolog consistem em coleções de sentenças. Prolog tem apenas alguns tipos de sentenças, mas elas podem ser complexas.

Um uso comum de Prolog é como um tipo de base de dados inteligente. Essa aplicação fornece um esquema simples para discutir a linguagem.

A base de dados de um programa Prolog consiste em dois tipos de sentenças: fatos e regras. Os seguintes são exemplos de sentenças de fatos:

```
mother(joanne, jake).
father(vern, joanne).
```

Essas sentenças afirmam que `joanne` é a mãe (`mother`) de `jake`, e que `vern` é o pai (`father`) de `joanne`.

Um exemplo de regra é

```
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

Essa sentença diz que se pode inferir que `X` é o avô (`grandparent`) de `Z`, se for verdade que `X` é o pai (`parent`) de `Y` e que `Y` é o pai (`parent`) de `Z`, para alguns valores específicos das variáveis `X`, `Y` e `Z`.

A base de dados Prolog pode ser consultada interativamente com sentenças-objetivo. Este é um exemplo:

```
father(bob, darcie).
```

Ele pergunta se `bob` é o pai (`father`) de `darcie`. Quando tal consulta, ou objetivo, é apresentada para o sistema Prolog, este usa seu processo de resolução para tentar determinar a verdade da sentença. Se conseguir concluir que o objetivo é verdadeiro, ele mostra “true”. Se não puder prová-lo, mostra “false”.

2.13.3 Avaliação

Nos anos 1980, um grupo relativamente pequeno de cientistas da computação acreditava que a programação lógica era a melhor alternativa para escapar da complexidade das linguagens imperativas e do problema de produzir a imensa quantidade de software confiável necessária. Até agora, existem duas grandes razões pelas quais a programação

lógica não se tornou mais usada. Primeiro, como acontece com outras estratégias não imperativas, os programas escritos em linguagens lógicas têm se mostrado ineficientes se comparados aos programas imperativos equivalentes. Segundo, foi determinado que essa estratégia é eficiente para apenas algumas áreas de aplicação: certos tipos de sistemas de gerenciamento de bancos de dados e áreas de IA.

Existe um dialeto de Prolog que oferece suporte à programação orientada a objetos: Prolog++ (Moss, 1994). Programação lógica e Prolog são descritos mais detalhadamente no Capítulo 16.

2.14 O MAIOR ESFORÇO DE PROJETO DA HISTÓRIA: ADA

A linguagem Ada é resultado do mais extenso e caro esforço de projeto de uma linguagem de programação da história. Os parágrafos a seguir descrevem brevemente a evolução de Ada.

2.14.1 Perspectiva histórica

A linguagem Ada foi desenvolvida para o Departamento de Defesa dos Estados Unidos (DoD) – logo, o estado do ambiente de computação do DoD foi fundamental para determinar sua forma. Em 1974, mais de metade das aplicações de computadores no DoD eram sistemas embarcados. Um sistema embarcado é aquele no qual o hardware do computador está incorporado ao dispositivo que controla ou para o qual fornece serviços. Os custos de software estavam crescendo rapidamente, principalmente devido à crescente complexidade dos sistemas. Mais de 450 linguagens de programação estavam em uso para projetos do DoD e nenhuma delas era padronizada por esse órgão. Cada fornecedor militar podia definir uma linguagem nova e diferente para cada contrato.[12] Devido a essa proliferação de linguagens, os aplicativos de software raramente eram reutilizados. Além disso, nenhuma ferramenta de software havia sido criada (porque elas são normalmente dependentes de linguagens). Muitas linguagens excelentes estavam sendo usadas, mas nenhuma era adequada para aplicações de sistemas embarcados. Por essas razões, em 1974, o Exército, a Marinha e a Força Aérea dos Estados Unidos propuseram, independentemente, o desenvolvimento de uma única linguagem de alto nível para sistemas embarcados.

2.14.2 Processo de projeto

Ao notar esse amplo interesse, em janeiro de 1975, Malcolm Currie, diretor de pesquisa e engenharia de defesa, formou o Grupo de Trabalho para Linguagem de Alto Nível (HOLWG – High-Order Language Working Group), inicialmente liderado pelo tenente-coronel William Whitaker, da Força Aérea. O HOLWG tinha representantes de todos os serviços militares americanos e colaboradores da Grã-Bretanha, da França e da então Alemanha Ocidental. Seu objetivo inicial era:

[12] Esse resultado foi amplamente devido ao uso difundido da linguagem assembly para sistemas embarcados, assim como ao fato de a maioria dos sistemas embarcados usar processadores especializados.

- Identificar os requisitos para uma nova linguagem de alto nível para o DoD.
- Avaliar linguagens existentes para determinar se existia uma candidata viável.
- Recomendar a adoção ou implementação de um conjunto mínimo de linguagens de programação.

Em abril de 1975, o HOLWG produziu o documento de requisitos inicial (Strawman – Homem de Palha) para a nova linguagem (Departament of Defense, 1975a). Ele foi distribuído para órgãos militares, agências federais, representantes selecionados da indústria e das universidades e colaboradores interessados na Europa.

O documento Strawman foi seguido pelo Woodenman (Homem de Madeira) em agosto de 1975 (Departament of Defense, 1975b), pelo Tinman (Homem de Estanho) em janeiro de 1976 (Departament of Defense, 1976), Ironman (Homem de Ferro) em janeiro de 1977 (Departament of Defense, 1977) e, finalmente, pelo Steelman (Homem de Aço) em junho de 1978 (Departament of Defense, 1978).

Após um processo tedioso, as propostas submetidas para a linguagem foram reduzidas a quatro finalistas, todas baseadas em Pascal. Em maio de 1979, a proposta de projeto da Cii Honeywell/Bull foi escolhida entre os quatro finalistas como o projeto que seria usado. A equipe da Cii Honeywell/Bull, única concorrente estrangeira, era liderada por Jean Ichbiah.

Na primavera de 1979, Jack Cooper, do Comando Material da Marinha, sugeriu um nome para a nova linguagem, Ada, que foi então adotado. O nome homenageia Augusta Ada Byron (1815-1851), condessa de Lovelace, matemática e filha do poeta Lord Byron. Ela é comumente reconhecida como a primeira programadora do mundo. Augusta trabalhou com Charles Babbage em seus computadores mecânicos, as Máquinas Diferenciais e Analíticas, escrevendo programas para diversos processos numéricos.

O projeto e as razões fundamentais para a linguagem Ada foram publicados pela ACM na *SIGPLAN Notices* (ACM, 1979) e distribuídos para mais de 10 mil pessoas. Um teste público e uma conferência de avaliação foram realizados em outubro de 1979, em Boston, com representantes de mais de cem organizações dos Estados Unidos e da Europa. Em novembro, mais de 500 relatórios sobre a linguagem, de 15 países, haviam sido recebidos. A maioria sugeria pequenas modificações, em vez de mudanças drásticas e rejeições completas. Com base nos relatórios sobre a linguagem, a versão seguinte da especificação de requisitos, o documento Stoneman (Homem de Pedra) (Departament of Defense, 1980a), foi lançado em fevereiro de 1980.

Uma versão revisada do projeto da linguagem foi finalizada em julho de 1980 e aceita com o nome MIL-STD 1815, o *Manual de Referência da Linguagem Ada* padrão. O número 1815 foi escolhido por ser o ano de nascimento de Augusta Ada Byron. Outra versão revisada do *Manual de Referência da Linguagem Ada* foi lançada em julho de 1982. Em 1983, o Instituto Nacional de Padrões dos Estados Unidos padronizou Ada. Essa versão oficial "final" é descrita em Goos e Hartmanis (1983). O projeto da linguagem Ada foi então congelado por cinco anos.

2.14.3 Visão geral da linguagem

Esta seção descreve brevemente quatro das principais contribuições da linguagem Ada.

Nela, pacotes fornecem os meios para encapsular objetos de dados, especificações para tipos de dados e procedimentos. Isso, por sua vez, fornece o suporte para o uso de abstração de dados no projeto de programas.

A linguagem Ada inclui diversos recursos para o tratamento de exceções, os quais permitem que os programadores obtenham o controle após ter sido detectada a ocorrência de uma exceção, ou erros em tempo de execução, dentro de uma variedade de exceções possíveis.

Em Ada, as unidades de programas podem ser genéricas. Por exemplo, é possível escrever um procedimento de ordenação usando um tipo não especificado para os dados a serem ordenados. Tal procedimento genérico deve ser instanciado para um tipo específico, antes de poder ser usado, o que é feito com uma sentença que faz o compilador gerar uma versão do procedimento com o tipo informado. A disponibilidade de tais unidades genéricas aumenta a gama de unidades de programas que podem ser reutilizadas, em vez de duplicadas, pelos programadores.

A linguagem Ada também fornece a execução concorrente de unidades de programa especiais, chamadas tarefas, por meio do mecanismo *rendezvous*. *Rendezvous* é o nome de um método de sincronização e comunicação entre tarefas.

2.14.4 Avaliação

Talvez os aspectos mais importantes do projeto da linguagem Ada sejam:

- Como o projeto era competitivo, não existiam limites na participação.
- A linguagem Ada agrupa a maioria dos conceitos de engenharia de software e projeto de linguagem do final dos anos 1970. Apesar de ser possível questionar as abordagens concretas usadas para incorporar esses recursos, bem como a inclusão de um número muito grande deles em uma linguagem, a maioria das pessoas concorda que os recursos são valiosos.
- Apesar de a maioria das pessoas não ter imaginado isso, o desenvolvimento de um compilador para a linguagem Ada era uma tarefa difícil. Apenas em 1985, quase quatro anos após a conclusão do projeto da linguagem, é que compiladores Ada realmente usáveis começaram a aparecer.

A crítica mais séria em relação a Ada em seus primeiros anos foi que a linguagem era muito extensa e complexa. Em particular, Hoare (1981) afirmou que ela não deveria ser usada por quaisquer aplicações nas quais a confiabilidade fosse crítica, o que é precisamente o tipo de aplicações para as quais ela foi projetada. Por outro lado, outros a consideraram o ápice do projeto de linguagens de sua época. De fato, até mesmo Hoare, no fim das contas, amenizou sua visão da linguagem.

A seguir, temos um exemplo de um programa escrito em Ada:

```
-- Programa de exemplo em Ada
-- Entrada:    Um inteiro, List_Len, onde List_Len é menor
--             que 100, seguido por valores inteiros listlen
-- Saída:      O número de valores de entrada que são maiores
--             que a média de todos os valores de entrada
```

```
with Ada.Text_IO, Ada.Integer.Text_IO;
use Ada.Text_IO, Ada.Integer.Text_IO;
procedure Ada_Ex is
  type Int_List_Type is array (1..99) of Integer;
  Int_List : Int_List_Type;
  List_Len, Sum, Average, Result : Integer;
begin
  Result:= 0;
  Sum := 0;
  Get (List_Len);
  if (List_Len > 0) and (List_Len < 100) then
-- Lê dados de entrada em uma matriz e calcula a soma
    for Counter := 1 .. List_Len loop
      Get (Int_List(Counter));
      Sum := Sum + Int_List(Counter);
    end loop;
-- Calcula a média
    Average := Sum / List_Len;
-- Conta o número de valores que são maiores que a média
    for Counter := 1 .. List_Len loop
      if Int_List(Counter) > Average then
        Result:= Result+ 1;
      end if;
    end loop;
-- Imprimir o resultado
    Put ("The number of values > average is:");
    Put (Result);
    New_Line;
  else
    Put_Line ("Error-input list length is not legal");
  end if;
end Ada_Ex;
```

2.14.5 Ada 95 e Ada 2005

Dois dos mais importantes novos recursos de Ada 95 são brevemente descritos nos parágrafos seguintes. No restante deste livro, vamos usar o nome Ada 83 para a versão original e Ada 95 (seu nome atual) para a versão posterior, quando for importante distinguir as duas. Em discussões sobre recursos comuns a ambas, usaremos o nome Ada. O padrão da linguagem Ada 95 é definido em ARM (1995).

O mecanismo de derivação de tipos de Ada 83 é estendido em Ada 95 para permitir adições de novos componentes aos herdados de uma classe base. Isso permite a herança, um ingrediente-chave em linguagens de programação orientadas a objetos. A vinculação dinâmica de chamadas a definições de subprogramas é realizada por meio do despacho de subprogramas, o qual é baseado no valor de marcação de tipos derivados por tipos com a amplitude de classes (*classwide types*). Esse recurso possibilita o uso de polimorfismo, outro recurso importante da programação orientada a objetos.

O mecanismo de *rendezvous* de Ada 83 fornecia apenas um modo pesado e ineficiente de compartilhar dados entre processos concorrentes. Era necessário introduzir uma tarefa para controlar o acesso aos dados compartilhados. Os objetos protegidos de Ada 95 oferecem uma alternativa atraente. Os dados compartilhados são encapsulados em uma estrutura sintática que controla todos os acessos aos dados por *rendezvous* ou por chamadas a subprogramas.

Acredita-se que a popularidade de Ada 95 diminuiu porque o Departamento de Defesa parou de exigir seu uso em sistemas de software militares. Existiram, é claro, outros fatores que impediram o crescimento de sua popularidade. O mais importante foi a ampla aceitação de C++ para programação orientada a objetos, o que ocorreu antes do lançamento de Ada 95.

Houve várias adições a Ada 95 para chegar a Ada 2005. Entre elas estavam as interfaces, semelhantes às de Java, mais controle dos algoritmos de agendamento e interfaces sincronizadas.

Ada é amplamente usada na aviônica comercial e de defesa, no controle de tráfego aéreo e no transporte por estradas de ferro, entre outras áreas.

2.15 PROGRAMAÇÃO ORIENTADA A OBJETOS: SMALLTALK

Smalltalk foi a primeira linguagem de programação que ofereceu suporte completo à programação orientada a objetos. Logo, é uma parte importante de qualquer discussão sobre a evolução das linguagens de programação.

2.15.1 Processo de projeto

Os conceitos que levaram ao desenvolvimento de Smalltalk se originaram na tese de doutorado de Alan Kay, no final dos anos 1960, na Universidade de Utah (Kay, 1969). Kay teve a excepcional capacidade de prever a disponibilidade futura de computadores de mesa poderosos. Lembre-se de que os primeiros sistemas de microcomputadores não foram comercializados até meados dos anos 1970, e eles estavam apenas remotamente relacionados às máquinas vislumbradas por Kay, as quais deveriam executar 1 milhão ou mais de instruções por segundo e conter diversos megabytes de memória. Tais máquinas, na forma de estações de trabalho, tornaram-se disponíveis apenas no início dos anos 1980.

Kay acreditava que os computadores de mesa seriam usados por não programadores e, dessa forma, precisariam de capacidades muito poderosas de interação homem-máquina. Os computadores do final dos anos 1960 eram orientados a lotes e usados exclusivamente por programadores profissionais e cientistas. Para o uso de não programadores, Kay determinou que um computador deveria ser altamente interativo e usar interfaces gráficas sofisticadas com os usuários. Alguns dos conceitos gráficos vieram da experiência do LOGO de Seymour Paper, em que gráficos eram usados para ajudar crianças no uso de computadores (Papert, 1980).

Kay originalmente vislumbrou um sistema que chamou de Dynabook, pensado como um processador de informações gerais. Ele era baseado em parte na linguagem Flex, que Kay havia ajudado a projetar. Flex era baseada primariamente em SIMULA 67. O Dynabook usava o paradigma de uma mesa de trabalho típica, na qual existem diversos papéis, alguns parcialmente cobertos. A folha de cima é o foco da atenção,

enquanto as outras estão temporariamente fora de foco. A visualização do Dynabook modelaria essa cena, usando janelas de tela para representar diversas folhas de papel na área de trabalho. O usuário interagiria com tal visualização tanto por meio de toques de tecla quanto tocando na tela com os dedos. Após o projeto preliminar do Dynabook lhe render um doutorado, o objetivo de Kay se tornou ver tal máquina construída.

Kay conseguiu entrar no Centro de Pesquisa da Xerox em Palo Alto (Xerox PARC – Xerox Palo Alto Research Center) e apresentar suas ideias sobre o Dynabook. Isso o levou a ser empregado lá e, subsequentemente, aajudar no nascimento do Learning Research Group (Grupo de Pesquisa em Aprendizagem) da Xerox. A primeira atribuição do grupo era projetar uma linguagem para suportar o paradigma de programação de Kay e implementá-lo no melhor computador pessoal possível. Esses esforços resultaram em um Dynabook "Interino", composto de uma estação de trabalho Xerox Alto e do software Smalltalk-72. Juntos, eles formaram uma ferramenta de pesquisa para desenvolvimento futuro. Numerosos projetos de pesquisa foram conduzidos com esse sistema, incluindo experimentos para ensinar programação a crianças. Com os experimentos, vieram desenvolvimentos adicionais, levando a uma sequência de linguagens que culminaram com Smalltalk-80. A linguagem cresceu, assim como o poder do hardware no qual ela residia. Em 1980, tanto a linguagem quanto o hardware da Xerox praticamente coincidiam com o projeto original de Alan Kay.

2.15.2 Visão geral da linguagem

O mundo Smalltalk é povoado apenas por objetos, de constantes inteiras até grandes sistemas de software complexos. Toda a computação em Smalltalk é feita pela mesma técnica uniforme: o envio de uma mensagem a um objeto para invocar um de seus métodos. Uma resposta a uma mensagem é um objeto, o qual retorna a informação requisitada ou simplesmente notifica o chamador de que o processamento solicitado foi concluído. A diferença fundamental entre uma mensagem e uma chamada de subprograma é esta: uma mensagem é enviada para um objeto de dados, especificamente para um dos métodos definidos para o objeto. O método chamado é então executado, geralmente modificando os dados do objeto para o qual a mensagem foi enviada; uma chamada a subprograma é uma mensagem para o código de um subprograma. Normalmente, os dados a serem processados pelo subprograma são enviados a ele como parâmetro.[13]

Em Smalltalk, abstrações de objetos são classes, bastante similares às de SIMULA 67. Instâncias da classe podem ser criadas e são, então, os objetos do programa.

A sintaxe de Smalltalk é diferente daquelas da maioria das outras linguagens de programação, em grande parte devido ao uso de mensagens, em vez de expressões lógicas e aritméticas e sentenças de controle convencionais. Uma das construções de controle de Smalltalk é ilustrada no exemplo da próxima subseção.

2.15.3 Avaliação

Smalltalk teve um grande papel na promoção de dois aspectos distintos da computação: interfaces gráficas do usuário e programação orientada a objetos. O sistema de janelas,

[13]Evidentemente, uma chamada de método também pode passar dados para serem processados pelo método chamado.

que é agora o método dominante de interfaces com o usuário em sistemas de software, cresceu a partir dele. Atualmente, as metodologias de projeto de software e as linguagens de programação mais significativas são orientadas a objetos. Apesar de a origem de algumas das ideias de linguagens orientadas a objetos ser SIMULA 67, elas alcançaram a maturidade nem Smalltalk. É claro que o impacto de Smalltalk no mundo da computação é extenso e terá vida longa.

A seguir, temos um exemplo de uma definição de classe em Smalltalk:

```
"Programa de Exemplo em Smalltalk"
"A seguir está uma definição de classe, instanciações que podem de-
senhar polígonos equiláteros de qualquer número de lados"
class name                    Polygon
superclass                    Object
instance variable names       ourPen
numSides
sideLength
"Métodos de Classe"
  "Cria uma instância"
  new
     ^ super new getPen

  "Obtém uma caneta para desenhar o polígono"
  getPen
     ourPen <- Pen new defaultNib: 2

  "Métodos de instância"
  "Desenha um polígono"
  draw
     numSides timesRepeat: [ourPen go: sideLength;
                            turn: 360 // numSides]

  "Configura o tamanho dos lados"
  length: len
  sideLength <- len

  "Configura o número de lados"
  sides: num
     numSides <- num
```

2.16 COMBINAÇÃO DE RECURSOS IMPERATIVOS E ORIENTADOS A OBJETOS: C++

As origens da linguagem C foram discutidas na Seção 2.1, as de Simula 67 foram discutidas na Seção 2.10 e as de Smalltalk foram discutidas na Seção 2.15. C++ complementa os recursos da linguagem C, emprestados de Simula 67, para oferecer suporte aos recursos em que Smalltalk foi pioneira. C++ evoluiu a partir de C, com uma série de modificações para melhorar seus recursos imperativos e adicionar construções para dar suporte à programação orientada a objetos.

2.16.1 Processo de projeto

O primeiro passo de C em direção a C++ foi dado por Bjarne Stroustrup, no Bell Labs, em 1980. As modificações iniciais em C incluíam a adição de verificação de tipos e a conversão de parâmetros de funções e classes, as quais estavam relacionadas às de SIMULA 67 e de Smalltalk. Também estavam incluídas classes derivadas, controle de acesso público/privado de componentes herdados, métodos construtores e destrutores e classes amigas (*friend classes*). Durante 1981, foram adicionadas funções internalizadas (*inline functions*), parâmetros padrão e a sobrecarga do operador de atribuição. A linguagem resultante foi chamada de C com Classes e é descrita em Stroustrup (1983).

É útil considerar alguns dos objetivos de C com Classes. O primário era fornecer uma linguagem na qual os programas pudessem ser organizados da mesma forma que em SIMULA 67 – ou seja, com classes e herança. Um segundo objetivo importante era que deveriam existir penalidades pequenas ou nenhuma em termos de desempenho a C. Por exemplo, a verificação de faixa de índices de vetores não era considerada devido a uma desvantagem significativa de desempenho, em relação a C, que poderia resultar disso. Um terceiro objetivo do C com Classes era que poderia ser usado para quaisquer aplicações para as quais se poderia usar C; portanto, praticamente nenhum dos recursos de C seriam removidos, nem aqueles considerados inseguros.

Em 1984, a linguagem foi estendida com inclusão de métodos virtuais, que fornecem vinculação dinâmica de chamadas de métodos a definições de métodos específicos, nomes de métodos e sobrecarga de operadores e tipos de referência. Essa versão da linguagem foi chamada de C++ e é descrita em Stroustrup (1984).

Em 1985, apareceu a primeira implementação: um sistema chamado Cfront, que transformava programas C++ em programas C. Essa versão de Cfront e a versão de C++ que ela implementava foram denominadas Release 1.0. Essa versão é descrita em Stroustrup (1986).

Entre 1985 e 1989, C++ continuou a evoluir, baseada nas reações dos usuários em relação à primeira implementação distribuída. A versão seguinte foi chamada Release 2.0. Sua implementação de Cfront foi lançada em junho de 1989. Os recursos mais importantes adicionados a C++ Release 2.0 foram o suporte para herança múltipla (classes com mais de uma classe pai) e classes abstratas, entre algumas outras melhorias. Classes abstratas são descritas no Capítulo 12.

O Release 3.0 de C++ evoluiu entre 1989 e 1990. Ele adicionou *templates*, que fornecem tipos parametrizados, e tratamento de exceções. A versão atual de C++, padronizada em 1998, é descrita pela ISO (1998).

Em 2002, a Microsoft lançou sua plataforma de computação .NET, que incluía uma nova versão de C++, chamada de Managed C++ (ou C++ Gerenciada) ou MC++. MC++ estende C++ para fornecer acesso às funcionalidades do framework .NET. As adições incluem propriedades, *delegates*, interfaces e um tipo de referência para objetos coletados por um coletor de lixo. Propriedades são discutidas no Capítulo 11. *Delegates* são brevemente discutidos na introdução a C#, na Seção 2.19. Como .NET não suporta herança múltipla, MC++ também não o faz.

2.16.2 Visão geral da linguagem

Como C++ tem tanto funções quanto métodos, ele suporta a programação procedural e a orientada a objetos.

Operadores em C++ podem ser sobrecarregados – ou seja, o usuário pode criar novos operadores para os já existentes em tipos definidos por ele. Métodos em C++ também podem ser sobrecarregados, e isso significa que o usuário pode definir mais de um método com o mesmo nome, desde que os números ou tipos dos parâmetros sejam diferentes.

A vinculação dinâmica em C++ é fornecida por métodos virtuais. Esses métodos definem operações dependentes do tipo usando métodos sobrecarregados dentro de uma coleção de classes relacionadas por herança. Um ponteiro para um objeto de uma classe A pode apontar também para objetos de classes que têm a classe A como ancestral. Quando esse ponteiro aponta para um método virtual sobrecarregado, o do tipo atual é escolhido dinamicamente.

Tanto métodos quanto classes podem ser usados como *templates*, e isso significa que eles podem ser parametrizados. Por exemplo, um método pode ser escrito como um método com *template* de forma a permitir que ele tenha versões para uma variedade de tipos de parâmetros. As classes desfrutam da mesma flexibilidade.

C++ suporta herança múltipla. As construções de tratamento de exceção de C++ são discutidas no Capítulo 14.

2.16.3 Avaliação

C++ rapidamente se tornou (e se mantém) uma linguagem amplamente utilizada. Um fator para sua popularidade é a disponibilidade de compiladores bons e baratos. Outro é que ele é quase completamente compatível com C (o que significa que, com poucas alterações, programas em C podem ser compilados como programas C++) e, na maioria das implementações, é possível vincular código em C++ com código em C — e, assim, para muitos programadores que já conhecem C é relativamente fácil aprender C++. Por último, na época em que C++ apareceu, quando a programação orientada a objetos começou a despertar amplo interesse, era a única linguagem disponível conveniente para grandes projetos de software comercial.

Pelo lado negativo, como C++ é uma linguagem muito extensa e complexa, ela sofre deficiências similares àquelas da linguagem PL/I. C++ herdou muitas das inseguranças de C, tornando-se menos segura que linguagens como Ada e Java.

2.16.4 Uma linguagem relacionada: Objective-C

Objective-C (Kochan, 2009) é outra linguagem híbrida, com recursos imperativos e orientados a objetos. Ela foi projetada por Brad Cox e Tom Love, no início dos anos 1980. Inicialmente, consistia em C, mais as classes e a passagem de mensagem de Smalltalk. Entre as linguagens de programação criadas pela adição de suporte para pro-

gramação orientada a objetos em uma linguagem imperativa, Objective-C é a única a usar sintaxe Smalltalk para esse suporte.

Depois que Steve Jobs deixou a Apple e fundou a NeXT, licenciou Objective-C e ela foi usada para escrever o software de sistema de computador NeXT. A NeXT também lançou seu compilador de Objective-C, junto com o ambiente de desenvolvimento NeXTstep e uma biblioteca de utilitários. Depois que o projeto NeXT falhou, a Apple comprou a NeXT e usou Objective-C para escrever o MAC OS X. Objective-C é a linguagem de todo software de iPhone, o que explica o rápido aumento da sua popularidade depois que esse smartphone apareceu.

Uma característica que Objective-C herdou de Smalltalk é a vinculação dinâmica de mensagens a objetos. Isso significa que não existe nenhuma verificação estática de mensagens. Se uma mensagem é enviada para um objeto e o objeto não pode responder à mensagem, ela é desconhecida até o momento da execução, quando uma exceção é lançada.

Em 2006, a Apple anunciou Objective-C 2.0, que adicionou uma forma de coleta de lixo e uma nova sintaxe para declarar propriedades. Infelizmente, a coleta de lixo não é suportada pelo sistema de *run-time* do iPhone.

Objective-C é um superconjunto restrito de C; portanto, todas as inseguranças desta linguagem estão presentes em Objective-C.

2.17 UMA LINGUAGEM ORIENTADA A OBJETOS BASEADA NO PARADIGMA IMPERATIVO: JAVA

Os projetistas de Java começaram com C++, removeram algumas construções, modificaram outras e adicionaram poucas mais. A linguagem resultante fornece muito do poder e da flexibilidade de C++, mas em uma linguagem menor, mais simples e mais segura. Desde o projeto inicial, Java cresceu consideravelmente.

2.17.1 Processo de projeto

Java, como muitas linguagens de programação, foi projetado para uma aplicação que parecia não ter uma linguagem existente satisfatória. Em 1990, a Sun Microsystems determinou que havia a necessidade de uma linguagem de programação para dispositivos eletrônicos embarcados para o consumidor, como torradeiras, fornos de micro-ondas e sistemas interativos de TV. Confiabilidade era um dos objetivos primários de tal linguagem. Pode não parecer que a confiabilidade seja um fator importante no software para um forno de micro-ondas. Se um forno tem um problema de software, isso provavelmente não representará um grave risco para ninguém nem levará a grandes casos na justiça. Entretanto, se o sistema de software de determinado modelo contiver erros que forem descobertos após 1 milhão de unidades terem sido fabricadas e vendidas, um *recall* teria custos significativos. Logo, a confiabilidade é uma característica importante do software em produtos eletrônicos para o consumidor.

Após considerar C e C++, foi decidido que nenhuma das duas linguagens seria satisfatória para desenvolver software para dispositivos eletrônicos para o consumidor. Apesar de C ser relativamente pequeno, ele não fornece suporte para programação orientada a objetos, o que era considerado uma necessidade. C++ suportava programação orientada a objetos, mas a equipe da Sun a julgava muito extensa e complexa, em parte porque também suportava programação procedural. Também se acreditava que nem C nem C++ forneciam o nível necessário de confiabilidade. Então, uma nova linguagem, posteriormente chamada Java, foi projetada. Seu projeto foi guiado pelo objetivo fundamental de fornecer simplicidade e confiabilidade maiores do que as fornecidas por C++.

Apesar de o ímpeto inicial de Java serem os eletrônicos para consumidores, nenhum dos produtos nos quais ela foi usada em seus primeiros anos foi comercializado. A partir de 1993, quando a World Wide Web se tornou bastante usada, e devido aos novos navegadores gráficos, descobriu-se que Java era uma ferramenta útil de programação para a Web. Em particular, os *applets* Java, programas relativamente pequenos, interpretados em navegadores Web e cuja saída podia ser incluída e mostrada em documentos Web, rapidamente se tornaram populares da metade para o fim da década de 1990. Nos primeiros anos de popularidade de Java, a Web era a aplicação mais comum.

A equipe de projeto de Java era liderada por James Gosling, que havia projetado o editor emacs do UNIX e o sistema de janelas NeWS.

2.17.2 Visão geral da linguagem

Conforme já mencionado, Java é baseada em C++, mas foi projetada para ser menor, mais simples e mais confiável. Como C++, Java tem tanto classes quanto tipos primitivos. Vetores em Java são instâncias de uma classe pré-definida, enquanto em C++ eles não são – apesar de muitos usuários C++ construírem classes que encapsulam vetores para adicionar recursos como verificação de índice de faixa, o que é implícito em Java.

Java não tem ponteiros, mas seus tipos de referência fornecem algumas das capacidades de ponteiros. Essas referências são usadas para apontar instâncias de classes. Todos os objetos são alocados no monte. As referências são sempre implicitamente desreferenciadas, quando necessário. Dessa forma, elas se comportam mais como variáveis escalares normais.

Java tem um tipo booleano chamado **`boolean`**, usado principalmente para as expressões de controle de suas sentenças de controle (como **`if`** e **`while`**). Diferentemente de C e C++, expressões aritméticas não podem ser usadas para expressões de controle.

Uma diferença significativa entre Java e muitas de suas antecessoras que suportam orientação a objetos, incluindo C++, é não ser possível escrever subprogramas autocontidos em Java. Todos os subprogramas em Java são métodos definidos em classes. Além disso, os métodos podem ser chamados apenas por meio de uma classe ou objeto. Uma consequência disso é que, enquanto C++ oferece suporte tanto para programação orientada a objetos quanto procedural, Java oferece suporte apenas para programação orientada a objetos.

Outra diferença importante entre C++ e Java é que a primeira oferece suporte para herança múltipla diretamente em suas definições de classes. Java oferece suporte apenas para herança simples de classes, apesar de alguns dos benefícios da herança múltipla poderem ser obtidos pelo uso de interfaces.

Entre as construções C++ que não foram copiadas por Java estão as estruturas e as uniões.

Java inclui uma forma relativamente simples de controle de concorrência por meio de seu modificador **`synchronized`**, que pode aparecer em métodos e blocos. Em ambos, ele faz com que um bloqueio seja anexado. O bloqueio garante acesso ou execução mutuamente exclusiva. Em Java, é relativamente fácil criar processos concorrentes, chamados de *linhas de execução* (*threads*).

Java usa liberação implícita de armazenamento para seus objetos, geralmente chamada de **coleta de lixo**. Isso exime o programador da necessidade de remover explicitamente os objetos quando eles não forem mais necessários. Programas escritos em linguagens que não têm coleta de lixo sofrem de um problema chamado de vazamento de memória, ou seja, o armazenamento é alocado, mas nunca liberado. Isso pode levar ao consumo de todo o armazenamento disponível. A liberação de objetos é discutida em detalhes no Capítulo 6.

Diferentemente de C e C++, Java inclui coerções de tipo em atribuições (conversões de tipo implícitas) apenas se elas aumentarem o tipo (de um "menor" para um "maior"). Logo, coerções de **`int`** para **`float`** são feitas por meio do operador de atribuição, mas coerções de **`float`** para **`int`** não.

2.17.3 Avaliação

Os projetistas de Java acertaram em remover os recursos excessivos e/ou inseguros de C++. Por exemplo, a eliminação de metade das coerções de atribuição feitas em C++ é um passo à frente em direção a maior confiabilidade. A verificação de faixas de índices de acessos a vetores também torna a linguagem mais segura. A adição de concorrência melhora a gama de aplicações que podem ser escritas na linguagem, assim como as bibliotecas de classes para interfaces gráficas do usuário, acesso a bases de dados e redes.

A portabilidade de Java, ao menos em sua forma intermediária, é geralmente atribuída ao projeto da linguagem, mas na verdade essa atribuição não é correta. Qualquer linguagem pode ser traduzida para uma forma intermediária e "executada" em qualquer plataforma que tenha uma máquina virtual para essa forma intermediária. O preço desse tipo de portabilidade é o custo de interpretação, que tradicionalmente tem sido uma ordem de magnitude maior que a execução de código de máquina. A versão inicial do interpretador Java, chamado de Máquina Virtual Java (JVM), era ao menos 10 vezes mais lenta que os programas compilados em C equivalentes. Entretanto, muitos programas Java são agora traduzidos para código de máquina antes de serem executados, por meio de compiladores Just-in-Time (JIT). Isso torna a eficiência dos programas Java competitiva com a de programas escritos em linguagens compiladas de forma convencional, como C++, pelo menos quando a verificação de faixa de índices de vetores não é considerada.

O uso de Java aumentou mais rapidamente do que o de qualquer outra linguagem de programação. Inicialmente, isso ocorreu devido ao seu valor na programação de documentos Web dinâmicos. Claramente, uma das razões para a rápida ascensão de Java é que os programadores gostam de seu projeto. Alguns desenvolvedores pensavam que a linguagem C++ era muito extensa e complexa para ser prática e segura. Java ofereceu a eles uma alternativa com muito do poder d C++, mas em uma linguagem mais simples e

segura. Outro motivo é o fato de o sistema de compilação/interpretação para Java ser gratuito e fácil de obter na Web. Java é agora usada em uma variedade de áreas de aplicação.

A sua versão mais recente, Java SE8, apareceu em 2014. Desde a primeira versão, recursos significativos foram adicionais à linguagem. Entre eles estão uma classe de enumeração, genéricos, uma nova construção de iteração, expressões lambda e numerosas bibliotecas de classe.

A seguir, temos um exemplo de um programa em Java:

```
// Programa de exemplo em Java
// Entrada:   Um inteiro, listlen, onde listlen é menor
//            que 100, seguido por valores inteiros listlen
// Saída:     O número de valores de entrada que são maiores
//            que a média de todos os valores de entrada
import java.io.*;
class IntSort {
public static void main(String args[]) throws IOException {
  DataInputStream in = new DataInputStream(System.in);
  int listlen,
      counter,
      sum = 0,
      average,
      result = 0;
  int[] intlist = new int[99];
  listlen = Integer.parseInt(in.readLine());
  if ((listlen > 0) && (listlen < 100)) {
/* Lê os dados de entrada em um vetor e calcula sua soma */
    for (counter = 0; counter < listlen; counter++) {
      intlist[counter] =
              Integer.valueOf(in.readLine()).intValue();
      sum += intlist[counter];
    }
/* Calcula a média */
    average = sum / listlen;
/* Conta o número de valores que são maiores que a média */
    for (counter = 0; counter < listlen; counter++)
      if (intlist[counter] > average) result++;
/* Imprimir o resultado */
      System.out.println(
          "\Number of values > average is:" + result);
    } //** end of then clause of if ((listlen > 0) ...
    else System.out.println(
             "Error-input list length is not legal\n");
  } //** end of method main
} //** end of class IntSort
```

2.18 LINGUAGENS DE *SCRIPTING*

As linguagens de *scripting* evoluíram nos últimos 35 anos. As primeiras eram usadas por meio de uma lista de comandos, chamada de **script**, em um arquivo a ser interpre-

tado. A primeira dessas linguagens, chamada `sh` (de *shell*), começou como uma pequena coleção de comandos interpretados como chamadas a subprogramas de sistema que realizavam funções utilitárias, como gerenciamento de arquivos e filtragens de arquivos simples. A isso foram adicionadas variáveis, sentenças de controle de fluxo, funções e várias capacidades, e o resultado é uma linguagem de programação completa. Uma das mais poderosas e utilizadas dessas linguagens é a `ksh` (Bolsky e Korn, 1995), desenvolvida por David Korn no Bell Labs.

Outra linguagem de *scripting* é a `awk`, desenvolvida por Al Aho, Brian Kernighan e Peter Weinberger nos Laboratórios da Bell (Aho et al., 1988). A `awk` começou como uma linguagem de geração de relatórios, mas depois se tornou uma linguagem de propósito mais geral.

2.18.1 Origens e características de Perl

A linguagem Perl, desenvolvida por Larry Wall, era originalmente uma combinação de sh e awk. Perl cresceu significativamente desde seu início e é agora uma linguagem de programação poderosa, embora ainda um tanto primitiva. Apesar de ser geralmente chamada de linguagem de *scripting*, ela é mais parecida com uma linguagem imperativa típica, já que é sempre compilada, ao menos para uma linguagem intermediária, antes de ser executada. Além disso, ela tem todas as construções que a tornam aplicável a uma variedade de áreas de problemas computacionais.

Perl tem diversos recursos interessantes, dos quais apenas alguns são mencionados neste capítulo e discutidos mais adiante no livro.

Variáveis em Perl são estaticamente tipadas e implicitamente declaradas. Existem três espaços de nomes distintos para variáveis, denotados pelo primeiro caractere de nomes de variáveis. Todos os nomes de variáveis escalares começam com cifrão (`$`), todos os nomes de vetores começam com arroba (`@`) e todos os nomes de dispersões (*hashes*) – dispersões são brevemente descritas abaixo – começam com sinais de percentual (`%`). Essa convenção torna os nomes em programas mais legíveis que aqueles em qualquer outra linguagem de programação.

Perl inclui um grande número de variáveis implícitas. Algumas são usadas para armazenar parâmetros Perl, como o formato particular do caractere de nova linha ou caracteres que são usados na implementação. Variáveis implícitas são usadas como parâmetros padrão para funções pré-definidas e operandos padrão para alguns operadores. As variáveis implícitas têm nomes distintivos – embora misteriosos –, como `$!` e `@_`. Os nomes de variáveis implícitas, como os de variáveis definidas pelo usuário, usam os três espaços de nomes, logo `$!` é um escalar.

Os vetores em Perl têm duas características que os diferenciam de outros vetores das linguagens imperativas comuns. Primeiro, eles têm tamanho dinâmico, ou seja, podem crescer e encolher conforme necessário durante a execução. Segundo, os vetores podem ser esparsos, ou seja, pode haver espaços em branco entre os elementos. Esses espaços em branco não ocupam espaço na memória, e a sentença de iteração usada para vetores, **`foreach`**, itera sobre os elementos que faltam.

Perl inclui vetores associativos, chamados de **dispersões**. Essas estruturas de dados são indexadas por cadeias e são tabelas de dispersão (*hash tables*) implicitamente controladas. O sistema Perl fornece a função hash e aumenta o tamanho da estrutura conforme necessário.

Perl é uma linguagem poderosa, mas de certa forma perigosa. Seus tipos escalares armazenam tanto cadeias quanto números, normalmente armazenados em formato de ponto flutuante em precisão dupla. Dependendo do contexto, os números podem sofrer coerção para cadeias e vice-versa. Se uma string é usada em um contexto numérico e ela não puder ser convertida em um número, é usado zero e nenhuma mensagem de aviso ou erro é fornecida para o usuário. Isso pode levar a erros que não são detectados pelo compilador ou pelo sistema de tempo de execução. A indexação de vetores não pode ser verificada, porque não existem conjuntos de faixas de índices para os vetores. Referências a elementos não existentes retornam **undef**, interpretado como zero em contexto numérico. Então, não existe também detecção de erros no acesso a elementos de vetores.

O uso inicial de Perl foi como um utilitário do UNIX para processar arquivos de texto. Perl era e ainda é muito usada como uma ferramenta de administração de sistema em UNIX. Quando a World Wide Web apareceu, Perl atingiu grande utilização como uma linguagem CGI (Common Gateway Interface) para uso na Web, apesar de agora ser raramente usada para esse propósito. Ela é usada como linguagem de propósito geral para uma variedade de aplicações, como biologia computacional e inteligência artificial.

A seguir, temos um exemplo de um programa em Perl:

```
# Programa de exemplo em Perl
# Entrada:    Um inteiro, listlen, onde listlen é menor que
#             100, seguido por valores inteiros listlen
# Saída:      O número de valores de entrada que são maiores
#             que a média de todos os valores de entrada.
($sum, $result) = (0, 0);
$listlen = <STDIN>;
if (($listlen > 0) && ($listlen < 100)) {
# Lê os dados de entrada em um vetor e calcula sua soma
  for ($counter = 0; $counter < $listlen; $counter++) {
    $intlist[$counter] = <STDIN>;
  } #- end of for (counter ...
# Calcula a média
  $average = $sum / $listlen;
# Conta o número de valores que são maiores que a média
  foreach $num (@intlist) {
    if ($num > $average) { $result++; }
  } #- end of foreach $num ...
# Imprimir o resultado
  print "Number of values > average is: $result \n";
} #- end of if (($listlen ...
else {
  print "Error--input list length is not legal \n";
}
```

2.18.2 Origens e características de JavaScript

O uso da Web explodiu em meados dos anos 1990, após a aparição dos primeiros navegadores gráficos. A necessidade de computação associada a documentos HTML, os quais por si só eram completamente estáticos, rapidamente se tornou crítica. A compu-

tação no lado servidor era possível com o uso de CGI (Common Gateway Interface), que permitia aos documentos HTML requisitarem a execução de programas no servidor. Os resultados de tais computações retornavam ao navegador na forma de documentos HTML. A computação no navegador se tornou disponível com o advento dos *applets* Java. As duas abordagens foram substituídas em grande parte por novas tecnologias, primariamente linguagens de *scripting*.

JavaScript foi originalmente desenvolvida por Brendan Eich na Netscape. Seu nome original era Mocha. Depois, mudou para LiveScript. No final de 1995, LiveScript se tornou um projeto conjunto da Netscape com a Sun Microsystems e seu nome foi modificado para JavaScript. JavaScript passou por uma evolução extensa, da versão 1.0 para a versão 1.5, com a adição de muitos recursos e capacidades. Um padrão de linguagem para JavaScript foi desenvolvido no final dos anos 1990 pela Associação Europeia de Fabricantes de Computadores (ECMA – European Computer Manufacturers Association), chamado ECMA-262. Esse padrão também foi aprovado pela Organização Internacional de Padrões (ISO – International Standards Organization) como a ISO-16262. A versão da Microsoft de JavaScript é chamada de JScript .NET.

Apesar de um interpretador JavaScript poder ser embarcado em muitas aplicações, seu uso mais comum é em navegadores Web. Código JavaScript é embarcado em documentos HTML e interpretado pelo navegador quando os documentos são mostrados. Os principais usos de JavaScript na programação Web são a validação de dados de entrada de formulários e a criação de documentos HTML dinâmicos.

Apesar de seu nome, JavaScript é relacionada a Java apenas pelo uso de uma sintaxe similar. Java é fortemente tipada, mas JavaScript é dinamicamente tipada (consulte o Capítulo 5). As cadeias de caracteres e os vetores de JavaScript têm tamanho dinâmico. Assim, os índices de vetores não são verificados em relação a sua validade, apesar de isso ser obrigatório em Java. Java oferece suporte completo para programação orientada a objetos, mas JavaScript não oferece suporte para herança nem para vinculação dinâmica de chamadas a métodos.

Um dos usos mais importantes de JavaScript é na criação e modificação dinâmica de documentos HTML. JavaScript define uma hierarquia de objetos que casa com um modelo hierárquico de um documento HTML, definido pelo Document Object Model (DOM). Elementos de um documento HTML são acessados por meio desses objetos, fornecendo a base para o controle dinâmico dos elementos dos documentos.

A seguir, temos um script em JavaScript para o problema previamente solucionado em diversas linguagens neste capítulo. Note que assumimos que esse script será chamado a partir de um documento HTML e interpretado por um navegador Web.

```
// example.js
//   Entrada:  Um inteiro, listLen, onde listLen é menor
//             que 100, seguido por valores numéricos listLen
// Saída:      O número de valores de entrada que são maiores
//             que a média de todos os valores de entrada

var intList = new Array(99);
var listLen, counter, sum = 0, result = 0;

listLen = prompt (
        "Please type the length of the input list", "");
```

```
if ((listLen > 0) && (listLen < 100)) {

// Get the input and compute its sum
   for (counter = 0; counter < listLen; counter++) {
      intList[counter] = prompt (
                     "Please type the next number", "");
      sum += parseInt(intList[counter]);
   }

// Calcula a média
   average = sum / listLen;

// Conta o número de valores que são maiores que a média
   for (counter = 0; counter < listLen; counter++)
      if (intList[counter] > average) result++;

// Mostra os resultados
   document.write("Number of values > average is: ",
                  result, "<br />");
} else
   document.write(
       "Error - input list length is not legal <br />");
```

2.18.3 Origens e características de PHP

PHP (Tatroe et al., 2013) foi desenvolvida por Rasmus Lerdorf, membro do Grupo Apache, em 1994. Seu objetivo inicial era fornecer uma ferramenta para ajudar a rastrear os visitantes em seu site pessoal. Em 1995, ele desenvolveu um pacote chamado Personal Home Page Tools (Ferramentas para Páginas Pessoais), que foi a primeira versão distribuída publicamente de PHP. Originalmente, PHP era uma abreviação para Personal Home Page (Página Pessoal). Mais tarde, sua comunidade de usuários começou a usar o nome recursivo PHP: Hypertext Preprocessor (PHP: Processador de Hipertexto), o que levou o nome original à obscuridade. PHP é agora desenvolvida, distribuída e suportada como um produto de código aberto. Processadores PHP estão disponíveis na maioria dos servidores Web.

PHP é uma linguagem de *scripting*, do lado servidor, embutida em HTML e projetada para aplicações Web. O código PHP é interpretado no servidor Web quando um documento HTML no qual ele está embutido é requisitado por um navegador. O código PHP normalmente produz código HTML como saída, o qual o substitui no documento HTML. Logo, um navegador Web nunca vê código PHP.

PHP é similar a JavaScript em sua aparência sintática, na natureza dinâmica de suas cadeias e vetores e no uso de tipagem dinâmica. Os vetores em PHP são uma combinação dos vetores de JavaScript e das dispersões em Perl.

A versão original de PHP não oferecia suporte para programação orientada a objetos, mas isso foi adicionado na segunda versão. Entretanto, PHP não suporta classes abstratas ou interfaces, destrutores ou controles de acesso para membros de classes.

PHP permite um acesso simples aos dados de formulários HTML; logo, o processamento de formulários é fácil com PHP. Ele fornece suporte para muitos sistemas de gerenciamento de bancos de dados. Isso o torna uma linguagem útil para construir programas que precisam de acesso Web a bases de dados.

2.18.4 Origens e características de Python

Python (Lutz, 2013) é uma linguagem de *scripting* orientada a objetos, interpretada e relativamente recente. Seu projeto original foi feito por Guido van Rossum no Stichting Mathematisch Centrum, na Holanda, no início dos anos 1990. Seu desenvolvimento é feito agora pela Python Software Foundation. Python é usada para os mesmos tipos de aplicação que Perl: administração de sistemas, programação em CGI e outras tarefas computacionais relativamente pequenas. É um sistema de código aberto e está disponível para a maioria das plataformas de computação comuns. A implementação de Python está disponível em www.python.org, que fornece também informações sobre a linguagem.

A sintaxe de Python não é baseada diretamente em nenhuma linguagem comumente usada. Ela é uma linguagem com verificação de tipos, mas tipada dinamicamente. Em vez de vetores, inclui três tipos de estruturas de dados: listas; listas imutáveis, chamadas de **tuplas**; e dispersões, chamadas de **dicionários**. Existe uma coleção de métodos de lista, como inserir no final (`append`), inserir em uma posição arbitrária (`insert`), remover (`remove`) e ordenar (`sort`), assim como uma coleção de métodos para dicionários, como para obter chaves (`keys`), valores (`values`), para copiar (`copy`) e para verificar a existência de uma chave (`has_key`). Python também oferece suporte para compreensões de lista, originadas na linguagem Haskell. Compreensões de listas são discutidas na Seção 15.8.

Python é orientada a objetos, inclui as capacidades de casamento de padrões de Perl e tem tratamento de exceções. A coleta de lixo é usada para remover elementos da memória quando não são mais necessários.

O suporte para programação CGI, e para o processamento de formulários em particular, é fornecido pelo módulo `cgi`. Módulos que oferecem suporte a *cookies*, redes e acesso a bases de dados também estão disponíveis.

Python inclui suporte para concorrência com suas linhas de execução (*threads*) e suporte para programação de rede com seus soquetes. Tem também mais suporte para programação funcional que outras linguagens de programação não funcionais.

Um dos recursos mais interessantes de Python é que ela pode ser facilmente estendida por qualquer usuário. Os módulos que suportam as extensões podem ser escritos em qualquer linguagem compilada. Extensões podem adicionar funções, variáveis e tipos de objetos. Essas extensões são implementadas como adições ao interpretador Python.

2.18.5 Origens e características de Ruby

Ruby (Thomas et al., 2005) foi projetada por Yukihiro Matsumoto (também conhecido como Matz) no início dos anos 1990 e lançada em 1996. Desde então, tem evoluído continuamente. A motivação para Ruby foi a falta de satisfação de seu projetista com

Perl e Python. Apesar de tanto Perl quanto Python oferecerem suporte à programação orientada a objetos[14], nenhuma delas é uma linguagem puramente orientada a objetos, ao menos no sentido de cada uma ter tipos primitivos (não objetos) e aceitar o uso de funções.

O recurso característico primário de Ruby é que se trata de uma linguagem orientada a objetos pura, como Smalltalk. Cada valor de dados é um objeto e todas as operações são feitas por meio de chamadas a métodos. Os operadores em Ruby são os únicos mecanismos sintáticos para especificar chamadas a métodos para as operações correspondentes. Como são métodos, podem ser redefinidos. Todas as classes, pré-definidas ou definidas pelos usuários, são passíveis de extensão por herança.

Tanto as classes quanto os objetos em Ruby são dinâmicos, no sentido de que os métodos podem ser adicionados dinamicamente a ambos. Isso significa que tanto classes quanto objetos podem ter conjuntos de métodos em diferentes momentos durante a execução. Então, instanciações diferentes da mesma classe podem se comportar de maneira diferente. Coleções de métodos, dados e constantes podem ser incluídas na definição de uma classe.

A sintaxe de Ruby está relacionada à de Eiffel e à de Ada. Não existe necessidade de declarar variáveis, porque é usada a tipagem dinâmica. O escopo de uma variável é especificado em seu nome: uma variável cujo nome começa com uma letra tem escopo local; outra que começa com `@` é uma variável de instância, e aquela que começa com $ tem escopo global. Diversos recursos de Perl estão presentes em Ruby, incluindo variáveis implícitas com nomes patéticos, como `$_`.

Assim como acontece com Python, qualquer usuário pode estender e/ou modificar Ruby. Ela tem interesse cultural, pois é a primeira linguagem de programação projetada no Japão a ser usada de forma relativamente ampla nos Estados Unidos.

2.18.6 Origens e características de Lua

Lua[15] foi projetada no início dos anos 1990 por Roberto Ierusalimschy, Waldemar Celes e Luis Henrique de Figueiredo na Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), no Brasil. É uma linguagem de *scripting* que oferece suporte para programação procedural e funcional com a extensibilidade como um de seus objetivos primários. Entre as linguagens que influenciaram seu projeto estão Scheme, Icon e Python.

Lua é similar a JavaScript no sentido de não oferecer suporte para programação orientada a objetos, mas foi influenciada por ela. Ambas têm objetos que desempenham o papel tanto de classes quanto de objetos e herança baseada em protótipo, em vez de herança de classe. Entretanto, em Lua, a linguagem pode ser estendida para oferecer suporte à programação orientada a objetos.

Como em Scheme, as funções de Lua são valores de primeira classe. Além disso, a linguagem oferece suporte para *closures*. Essas capacidades permitem que ela seja usa-

[14] Na verdade, o suporte de Python para programação orientada a objetos é parcial.

[15] O nome Lua homenageia o satélite natural da Terra.

da para programação funcional. Também como Scheme, Lua tem apenas uma estrutura de dados, apesar de ser a tabela. As tabelas de Lua estendem os vetores associativos de PHP, os quais incluem os vetores presentes nas linguagens imperativas tradicionais. Referências a elementos de tabela podem tomar a forma de referências a vetores tradicionais, vetores associativos ou registros. Como as funções são valores de primeira classe, elas podem ser armazenadas em tabelas, e essas tabelas podem servir como espaços de nomes.

Lua usa coleta de lixo para seus objetos, os quais são todos alocados no monte. Ela usa tipagem dinâmica, como a maioria das outras linguagens de *scripting*.

Lua é uma linguagem pequena e relativamente simples, com apenas 21 palavras reservadas. A filosofia de projeto da linguagem é fornecer apenas o necessário e maneiras simples de estender a linguagem para permitir que ela se encaixe em uma variedade de áreas de aplicação. Grande parte de sua extensibilidade é derivada de sua estrutura de dados, a tabela, a qual pode ser personalizada por meio do conceito de metatabelas de Lua.

Lua pode ser usada como uma extensão de uma linguagem de *scripting* para outras linguagens. Como as primeiras implementações de Java, Lua é traduzida em um código intermediário e interpretada. Ela pode ser facilmente embarcada em outros sistemas, em parte por causa do tamanho pequeno de seu interpretador, que tem apenas cerca de 150 Kbytes.

2.19 A PRINCIPAL LINGUAGEM .NET: C#

C#, junto com a nova plataforma de desenvolvimento .NET,[16] foi anunciada pela Microsoft em 2000. Versões de produção de ambas foram lançadas em janeiro de 2002.

2.19.1 Processo de projeto

C# é baseada em C++ e Java, mas também inclui algumas ideias de Delphi e de Visual Basic. Seu projetista líder, Anders Hejlsberg, também projetou Turbo Pascal e Delphi, o que explica as partes Delphi da herança de C#.

O propósito de C# é fornecer uma linguagem para o desenvolvimento de software baseado em componentes, especificamente para tal desenvolvimento no framework .NET. Nesse ambiente, componentes de uma variedade de linguagens podem ser facilmente combinados para formarem sistemas. Todas as linguagens do .NET, incluindo C#, VB .NET, C++ gerenciado, F# e JScript .NET[17], usam o Sistema de Tipos Comum (CTS – Common Type System). O CTS fornece uma biblioteca de classes comum. Todos os tipos, nas cinco linguagens do .NET, herdam de uma classe raiz, `System.Object`. Compiladores que estão em conformidade com a especificação CTS criam objetos que podem ser combinados em sistemas de software. Todas as linguagens do .NET são com-

[16] O sistema de desenvolvimento .NET é discutido brevemente no Capítulo 1.

[17] Muitas outras linguagens têm sido modificadas para serem linguagens do .NET.

piladas na mesma forma intermediária, Intermediate Language (IL).[18] Diferentemente de Java, entretanto, a IL nunca é interpretada. Um compilador Just-in-Time é usado para converter IL em código de máquina antes de ela ser executada.

2.19.2 Visão geral da linguagem

Muitos especialistas acreditam que um dos avanços mais importantes de Java em relação a C++ está na exclusão de alguns recursos. Por exemplo, C++ oferece suporte para herança múltipla, ponteiros, estruturas, tipos **`enum`**, sobrecarga de operadores e uma sentença goto, mas Java não inclui nenhum desses recursos.[19] Os projetistas de C# discordaram dessas remoções, porque todos esses recursos, exceto a herança múltipla, foram incluídos na nova linguagem.

Entretanto, em diversos casos, a versão C# de um recurso C++ foi melhorada. Por exemplo, os tipos **`enum`** de C# são mais seguros, porque nunca são implicitamente convertidos para inteiros. Isso permite que sejam mais seguros em relação a tipos. O tipo **`struct`** foi modificado significativamente, resultando em uma construção verdadeiramente útil, enquanto em C++ tal construção é praticamente inútil. Os structs de C# são discutidos no Capítulo 12. C# dá um passo à frente ao melhorar a sentença **`switch`** usada em C, C++ e Java. O switch de C# é discutido no Capítulo 8.

Apesar de C++ incluir ponteiros para funções, eles compartilham a falta de segurança inerente aos ponteiros para variáveis de C++. C# inclui um novo tipo, os *delegates*, referências a subprogramas que são tanto orientadas a objetos quanto seguras em relação a tipos. *Delegates* são usados para implementar manipuladores de evento, controlando a execução de *threads* e *callbacks*.[20] *Callbacks* são implementadas em Java por meio de interfaces; em C++ são usados ponteiros para métodos.

Em C#, os métodos podem ter um número variável de parâmetros, desde que sejam do mesmo tipo. Isso é especificado pelo uso de um parâmetro formal do tipo vetor, precedido pela palavra reservada **`params`**.

Tanto C++ quanto Java usam sistemas de tipos distintos: um para tipos primitivos e outro para objetos. Além de ser confusa, essa distinção leva à necessidade de converter valores entre os dois sistemas – por exemplo, para colocar um valor primitivo em uma coleção que armazena objetos. C# faz a conversão entre valores dos dois sistemas de tipos de forma parcialmente implícita, por meio de operações de *boxing* e *unboxing*, discutidas em detalhes no Capítulo 12.[21]

Entre outros recursos de C# estão os vetores retangulares, não suportados pela maioria das linguagens de programação, e uma sentença **`foreach`**, usada para iterar em

[18] Inicialmente, a IL se chamava MSIL (Microsoft Intermediate Language), mas aparentemente muitas pessoas achavam esse nome longo demais.

[19] N. de R. T.: Tipos `enum` estão disponíveis em Java desde a versão 1.5, de 2004.

[20] Quando um objeto chama um método de outro objeto e precisa ser notificado quando esse método tiver concluído sua tarefa, o método chamado convoca seu chamador. Isso é conhecido como *callback*.

[21] Esse recurso foi adicionado à linguagem Java em sua versão 5.0.

vetores e objetos de coleção. Uma sentença **foreach** similar é encontrada em Perl, PHP e Java 5.0. Além disso, C# inclui propriedades, uma alternativa aos atributos de dados públicos. Propriedades são especificadas como atributos de dados com métodos de leitura e escrita, os quais são implicitamente chamados quando referências e atribuições são feitas aos atributos de dados associados.

C# evoluiu rápida e continuamente desde seu lançamento, em 2002. A versão mais recente é C# 5.0. Uma novidade de C# 5.0 são as funções assíncronas.

2.19.3 Avaliação

C# foi criada como um aprimoramento, tanto em relação a C++ quanto em relação a Java, como uma linguagem de programação de propósito geral. Apesar de ser possível argumentar que alguns de seus recursos são vistos como um retrocesso, C# inclui algumas construções que a movem à frente de suas antecessoras. Alguns de seus recursos certamente serão adotados por outras linguagens de programação do futuro.

A seguir, temos um exemplo de um programa em C#:

```
// Programa de exemplo em C#
// Entrada:  Um inteiro, listlen, onde listlen é menor que
//           100, seguido por valores inteiros listlen.
// Saída:    O número de valores de entrada que são maiores
//           que a média de todos os valores de entrada
using System;
public class Ch2example {
  static void Main() {
    int[] intlist;
    int listlen,
        counter,
        sum = 0,
        average,
        result = 0;
    intList = new int[99];
    listlen = Int32.Parse(Console.readLine());
    if ((listlen > 0) && (listlen < 100)) {
// Lê os dados de entrada em um vetor e calcula sua soma
      for (counter = 0; counter < listlen; counter++) {
        intList[counter] =
                       Int32.Parse(Console.readLine());
        sum += intList[counter];
      } //- end of for (counter ...
// Calcula a média
      average = sum / listlen;
// Conta o número de valores que são maiores que a média
      foreach (int num in intList)
        if (num > average) result++;
// Imprimir o resultado
```

```
      Console.WriteLine(
         "Number of values > average is:" + result);
    } //- end of if ((listlen ...
    else
      Console.WriteLine(
         "Error--input list length is not legal");
  } //- end of method Main
} //- end of class Ch2example
```

2.20 LINGUAGENS HÍBRIDAS DE MARCAÇÃO-PROGRAMAÇÃO

Uma linguagem híbrida de marcação-programação é uma linguagem de marcação na qual alguns dos elementos podem especificar ações de programação, como controle de fluxo e computação. As subseções a seguir apresentam duas linguagens híbridas, XSLT e JSP.

2.20.1 XSLT

eXtensible Markup Language (XML) é uma linguagem de metamarcação. Uma linguagem assim é usada para definir linguagens de marcação. Linguagens de marcação derivadas de XML são usadas para definir documentos de dados XML. Apesar de os documentos XML serem legíveis por humanos, eles são processados por computadores. Esse processamento algumas vezes consiste apenas em transformações para formatos alternativos que possam ser efetivamente visualizados ou impressos. Em muitos casos, as transformações são para HTML, que pode ser mostrada por um navegador Web. Em outros, os dados no documento são processados, como outras formas de arquivos de dados.

A transformação de documentos XML para HTML é especificada em outra linguagem de marcação, chamada de XSLT (eXtensible Stylesheet Language Transformations – Transformações em Linguagem de Folhas de Estilo Extensível) – www.w3.org/TR/XSLT. A XSLT pode especificar operações similares àquelas de programação. Logo, XSLT é uma linguagem híbrida de marcação-programação. XSLT foi definida pelo World Wide Web Consortium (W3C – Consórcio Web) no fim dos anos 1990.

Um processador XSLT é um programa que recebe como entrada um documento de dados XML e um XSLT (também especificado na forma de um documento XML). Nesse processamento, o documento de dados XML é transformado em outro documento XML,[22] por meio das transformações descritas no documento XSLT. XSLT especifica as transformações pela definição de *templates*, padrões de dados que podem ser encontrados pelo processador XSLT no arquivo XML de entrada. Associadas a cada *template* no documento XSLT estão suas instruções de transformação, as quais especificam como os dados que casam com os *templates* devem ser transformados antes de serem colocados no documento de saída. Logo, os *templates* (e seu processamento associado) agem como

[22] O documento de saída do processador XSLT também poderia ser em HTML ou em texto puro.

subprogramas que são "executados" quando o processador XSLT encontra um casamento de padrões nos dados do documento XML.

XSLT também tem construções de programação em um nível mais baixo. Por exemplo, uma construção de iteração é incluída, permitindo que partes repetidas do documento XML sejam selecionadas. Existe também um processo de ordenação. Essas construções de baixo nível são especificadas com *tags* XSLT, como `<for-each>`.

2.20.2 JSP

A parte principal de JSTL (Java Server Pages Standard Tag Library) é outra linguagem híbrida de marcação-programação, apesar de seu formato e propósito serem diferentes daqueles de XSLT. Antes de discutir JSTL, é necessário introduzir as ideias de *servlets* e de Java Server Pages (JSP). Um ***servlet*** é uma instância de uma classe Java que reside e é executada em um sistema de servidor Web. A execução de um *servlet* é solicitada por um documento de marcação mostrado em um navegador Web. A saída de um *servlet*, feita na forma de um documento HTML, é retornada para o navegador requisitante. Um programa executado no processo do servidor Web, chamado de ***servlet container***, controla a execução dos *servlets*. *Servlets* são comumente usados para o processamento de formulário e para o acesso a bases de dados.

JSP é uma coleção de tecnologias projetadas para oferecer suporte para documentos Web dinâmicos e suprir outras necessidades de processamento de documentos Web. Quando um documento JSP, normalmente um misto de HTML e Java, é solicitado por um navegador, o programa processador de JSP, que reside em um sistema servidor Web, converte o documento para um *servlet*. O código Java embarcado no documento é copiado para o *servlet*. O código HTML puro é copiado em sentenças de impressão Java que o exibem na saída no estado em que se encontram. A marcação JSTL no documento JSP é processada, conforme discutido no próximo parágrafo. O *servlet* produzido pelo processador JSP é executado pelo *servlet container*.

JSTL define uma coleção de elementos de ações XML que controlam o processamento do documento JSP no servidor Web. Tais elementos têm o mesmo formato de outros da HTML e da XML. Um dos elementos de ação de controle JSTL mais usados é `if`, que especifica uma expressão booleana como atributo.[23] O conteúdo do elemento `if` (o texto entre a *tag* de abertura (`<if>`) e a *tag* de fechamento (`</if>`)) é o código HTML que será incluído no documento de saída apenas se a expressão booleana for avaliada como verdadeira. O elemento `if` é relacionado ao comando de pré-processador `#if` de C/C++. O contêiner JSP processa as partes JSTL dos documentos JSP de maneira similar à forma pela qual processadores C/C++ processam programas C e C++. Os comandos do pré-processador são instruções para que ele especifique como o arquivo de saída deve ser construído a partir do arquivo de entrada. De maneira similar, os elementos de controle de ação de JSTL são instruções para o processador JSP sobre como construir o arquivo de saída XML a partir do arquivo de entrada XML.

[23] Na HTML, um atributo, que é incorporado à *tag* de abertura de um elemento, fornece mais informações sobre esse elemento.

Um uso comum do elemento `if` é para a validação de dados de formulários submetidos por um usuário de um navegador. Os dados de formulários são acessíveis pelo processador JSP e podem ser testados com o elemento `if` para garantir que são os dados esperados. Se não forem, o elemento `if` pode inserir uma mensagem de erro para o usuário no documento de saída.

Para controle de seleção múltipla, JSTL tem os elementos `choose`, `when` e `otherwise`. Ela também inclui o `forEach`, que itera sobre coleções, em geral valores de formulário de um cliente. O elemento forEach pode incluir atributos `begin`, `end` e `step` para controlar suas iterações.

RESUMO

Investigamos o desenvolvimento de várias linguagens de programação. Este capítulo fornece ao leitor um panorama das questões atuais do projeto de linguagens, preparando-o para uma discussão aprofundada sobre os recursos mais importantes das linguagens contemporâneas.

NOTAS BIBLIOGRÁFICAS

Talvez a fonte mais importante de informações históricas sobre o desenvolvimento das primeiras linguagens de programação seja *History of Programming Languages*, editada por Richard Wexelblat (1981). Ela contém a perspectiva histórica do desenvolvimento e do contexto de 13 linguagens de programação importantes, de acordo com seus projetistas. Um trabalho similar resultou em uma segunda conferência sobre "história", publicada como uma edição especial da *ACM SIGPLAN Notices* (ACM, 1993a). Nesse trabalho, são discutidas a história e a evolução de mais 13 linguagens de programação.

O artigo "Early Development of Programming Languages" (Knuth e Pardo, 1977), parte da *Encyclopedia of Computer Science and Technology*, é um trabalho excelente de 85 páginas que detalha o desenvolvimento de linguagens até Fortran. O artigo inclui programas de exemplo para demonstrar os recursos de muitas dessas linguagens.

Outro livro muito interessante é o *Programming Languages: History and Fundamentals*, de Jean Sammet (1969). É um trabalho de 785 páginas repleto de detalhes sobre 80 linguagens de programação dos anos 1950 e 1960. Sammet também publicou diversas atualizações para seu livro, como *Roster of Programming Languages for 1974–75* (1976).

QUESTÕES DE REVISÃO

1. Em que ano Plankalkül foi projetada? Em que ano foi publicado o projeto?
2. Cite duas estruturas de dados comuns incluídas na Plankalkül.
3. Como eram implementados os pseudocódigos do início dos anos 1950?

4. A linguagem Speedcoding foi inventada para resolver duas limitações significativas do hardware computacional do início dos anos 1950. Que limitações eram essas?
5. Por que a lentidão da interpretação dos programas era aceitável no início dos anos 1950?
6. Que recursos de hardware apareceram pela primeira vez no computador IBM 704 e afetaram fortemente a evolução das linguagens de programação? Explique por quê.
7. Em que ano foi iniciado o projeto do Fortran?
8. Qual era a área primária de aplicação dos computadores na época em que Fortran foi projetado?
9. Qual é a fonte de todas as sentenças de fluxo de controle de Fortran I?
10. Qual foi o recurso mais significativo adicionado a Fortran I para chegar a Fortran II?
11. Quais sentenças de controle de fluxo foram adicionadas a Fortran IV para chegar a Fortran 77?
12. Que versão de Fortran foi a primeira a ter quaisquer tipos de variáveis dinâmicas?
13. Que versão de Fortran foi a primeira a ter manipulação de cadeias de caracteres?
14. Por que os linguistas estavam interessados em inteligência artificial no final dos anos 1950?
15. Onde Lisp foi desenvolvida? Por quem?
16. De que maneira Scheme e Common Lisp são linguagens opostas?
17. Que dialeto de Lisp é usado para cursos introdutórios de programação em algumas universidades?
18. Quais são as duas organizações profissionais que projetaram ALGOL 60?
19. Em que versão de ALGOL a estrutura de bloco apareceu?
20. Que elemento de linguagem que faltava a ALGOL 60 fez com que suas chances de uso disseminado diminuíssem?
21. Que linguagem foi projetada para descrever a de ALGOL 60?
22. Em que linguagem de programação COBOL foi baseada?
23. Em que ano o processo de projeto de COBOL começou?
24. Que estrutura de dados que apareceu em COBOL foi originada em Plankalkül?
25. Que organização foi a maior responsável pelo sucesso inicial de COBOL (em termos de uso)?
26. Para que grupo de usuários foi destinada a primeira versão de Basic?
27. Por que Basic foi uma linguagem importante no início dos anos 1980?
28. PL/I foi projetada para substituir quais duas outras linguagens?
29. Para qual nova linha de computadores PL/I foi projetada?

30. Que recursos de SIMULA 67 são agora partes importantes de algumas linguagens orientadas a objetos?
31. Que inovação em estruturas de dados foi introduzida em ALGOL 68, geralmente creditada a Pascal?
32. Que critério de projeto foi usado extensivamente em ALGOL 68?
33. Que linguagem introduziu a sentença `case`?
34. Que operadores em C foram modelados a partir de operadores similares em ALGOL 68?
35. Cite duas características de C que a tornam menos segura que Pascal.
36. O que é uma linguagem não procedural?
37. Quais são os dois tipos de sentenças que compõem uma base de dados Prolog?
38. Qual é a área de aplicação primária para a qual Ada foi projetada?
39. Como são chamadas as unidades de programas concorrentes em Ada?
40. Que construção de Ada fornece suporte para tipos abstratos de dados?
41. O que compõe o mundo Smalltalk?
42. Quais são os três conceitos base para a programação orientada a objetos?
43. Por que C++ inclui os recursos de C que são sabidamente inseguros?
44. De qual linguagem Objective-C empresta sua sintaxe para chamadas de método?
45. Qual é a principal aplicação de Objective-C?
46. O que as linguagens Ada e COBOL têm em comum?
47. Qual foi a primeira aplicação para Java?
48. Que característica de Java é mais evidente em JavaScript?
49. Como o sistema de tipos de PHP e JavaScript diferem daquele de Java?
50. Que estrutura de vetor é incluída em C#, mas não em C, C++ ou Java?
51. Quais são as duas linguagens que a versão original de Perl pretendia substituir?
52. Para qual área de aplicação JavaScript é mais usada?
53. Qual é o relacionamento entre JavaScript e PHP, em termos de utilização?
54. A estrutura de dados primária de PHP é uma combinação de quais duas outras estruturas de dados de outras linguagens?
55. Que estrutura de dados Python usa em vez de vetores?
56. Que características Ruby compartilha com Smalltalk?

57. Que característica dos operadores aritméticos de Ruby os tornam únicos entre aqueles de outras linguagens?
58. Que estruturas de dados são construídas em Lua?
59. Lua é normalmente compilada, puramente interpretada ou impuramente interpretada?
60. Que deficiência da sentença **`switch`** de C é sanada com as mudanças feitas por C# a essa construção?
61. Qual é a plataforma primária na qual C# é usada?
62. Quais são as entradas para um processador XSLT?
63. Qual é a saída de um processador XSLT?
64. Que elemento de JSTL é relacionado a um subprograma?
65. Por que um documento JSP é convertido por um processador JSP?
66. Os *servlets* são executados?

PROBLEMAS

1. Quais recursos de Plankalkül você acha que teriam maior influência na Fortran 0 se os projetistas da Fortran estivessem familiarizados com Plankalkül?
2. Determine as capacidades do sistema 701 Speedcoding de Backus e compare-as com as de uma calculadora de mão programável.
3. Escreva uma breve história dos sistemas A-0, A-1 e A-2 projetados por Grace Hooper e seus associados.
4. Compare as facilidades da Fortran 0 com as do sistema de Laning e Zierler.
5. Qual dos três objetivos originais do comitê de projeto de ALGOL, em sua opinião, foi mais difícil de ser atingido naquela época?
6. Em sua opinião, qual é o erro de sintaxe mais comum em programas Lisp?
7. Lisp começou como uma linguagem funcional pura, mas gradualmente foi adquirindo mais recursos imperativos. Por quê?
8. Descreva em detalhes as três razões mais importantes, em sua opinião, para ALGOL 60 não ter se tornado uma linguagem amplamente usada.
9. Por que, em sua opinião, COBOL permite identificadores longos, enquanto Fortran e ALGOL não permitiam?
10. Descreva a maior motivação da IBM para desenvolver PL/I.

11. Era correta a interpretação da IBM na qual foi baseada sua decisão para desenvolver PL/I, dada a história dos computadores e os desenvolvimentos de linguagem desde 1964?
12. Descreva, em suas próprias palavras, o conceito de ortogonalidade no projeto de linguagens de programação.
13. Qual é a razão primária pela qual PL/I se tornou mais usada que ALGOL 68?
14. Quais são os argumentos a favor e contra a ideia de uma linguagem sem tipos?
15. Existem outras linguagens de programação lógica, além de Prolog?
16. Qual a sua opinião sobre o argumento de que as linguagens muito complexas também são muito perigosas, e que devemos manter todas as linguagens pequenas e simples?
17. Você acha que o projeto de linguagem por comitê é uma boa ideia? Argumente.
18. As linguagens continuam a evoluir. Que tipo de restrições você acha adequadas para mudanças em linguagens de programação? Compare suas respostas com a evolução de Fortran.
19. Construa uma tabela identificando todas as principais evoluções das linguagens, em que conste quando elas ocorreram, em quais linguagens apareceram primeiro e as identidades dos desenvolvedores.
20. Existiram algumas trocas públicas entre a Microsoft e a Sun a respeito do projeto de J++ e de C# da Microsoft e de Java da Sun. Leia alguns desses documentos, disponíveis nos respectivos sites das empresas na Web, e escreva uma análise das discordâncias existentes.
21. Nos últimos anos, as estruturas de dados evoluíram dentro de linguagens de *scripting* para substituir os vetores tradicionais. Explique a sequência cronológica desses avanços.
22. Cite duas razões para que a interpretação pura seja um método de implementação aceitável para diversas das linguagens de *scripting* recentes.
23. Por que, em sua opinião, aparecem novas linguagens de *scripting* mais frequentemente do que novas linguagens compiladas?
24. Faça uma breve descrição geral de uma linguagem híbrida de marcação-programação.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Para entender o valor dos registros em uma linguagem de programação, escreva um pequeno programa em uma linguagem baseada em C, usando um vetor de estruturas que armazenem informações de estudantes, incluindo o nome, a idade, a média das notas como um valor de ponto-flutuante e o nível do estudante em uma cadeia (por exemplo, "calouro", etc.). Escreva também o mesmo programa na mesma linguagem sem usar tais estruturas.
2. Para entender o valor da recursão em uma linguagem de programação, escreva um programa que implemente o algoritmo *quicksort*, primeiro usando recursão e então sem usar recursão.
3. Para entender o valor dos laços de iteração de contagem, escreva um programa que implemente multiplicação de matrizes usando construções de repetição baseadas em contagem. Então, escreva o mesmo programa usando apenas laços de repetição lógicos – por exemplo, laços **`while`**.

Esta página foi deixada em branco intencionalmente.

3

Descrição da sintaxe e da semântica

3.1 Introdução

3.2 O problema geral de descrever sintaxe

3.3 Métodos formais para descrever sintaxe

3.4 Gramáticas de atributos

3.5 Descrição do significado de programas: semântica dinâmica

Este capítulo começa definindo os termos *sintaxe* e *semântica*. Após a definição dos termos, é apresentada uma discussão detalhada sobre o método mais comum de descrever sintaxe, as gramáticas livres de contexto (também conhecidas como Forma de Backus-Naur). As derivações, as árvores sintáticas, a ambiguidade, as descrições de precedência e associatividade de operadores e a Forma de Backus-Naur estendida estão incluídas nessa discussão. A seguir, são discutidas as gramáticas de atributos, usadas para descrever tanto a sintaxe quanto a semântica estática de linguagens de programação. Na última seção, são introduzidos três métodos formais de descrição de semântica: operacional, axiomática e denotacional. Dada a inerente complexidade desses métodos, nossa discussão sobre eles é breve. É possível escrever um livro inteiro para um único desses métodos (como muitos autores já fizeram).

3.1 INTRODUÇÃO

A tarefa de fornecer uma descrição concisa e compreensível de uma linguagem de programação é difícil, mas essencial para o sucesso dela. ALGOL 60 e ALGOL 68 foram apresentadas pela primeira vez por meio de descrições formais concisas; em ambos os casos, as descrições não eram facilmente entendidas, parcialmente porque cada uma usava uma nova notação. Os níveis de aceitação de ambas as linguagens foram afetados em função disso. Em contrapartida, algumas linguagens foram afetadas por existirem muitos dialetos levemente diferentes, resultado de uma definição simples, porém informal e imprecisa.

Um dos problemas em descrever uma linguagem é a diversidade de pessoas que precisam entender a descrição. Entre essas estão os avaliadores iniciais, os implementadores e os usuários. A maioria das novas linguagens de programação está sujeita a um período de escrutínio pelos usuários em potencial, geralmente pessoas dentro da organização que emprega o projetista da linguagem, antes de seus projetos estarem completos. Esses são os avaliadores iniciais. O sucesso desse ciclo de sugestões depende muito da clareza da descrição.

Os implementadores de linguagens de programação devem ser capazes de determinar como as expressões, as sentenças e as unidades de programa de uma linguagem são formadas, e também os efeitos pretendidos, quando executadas. A dificuldade do trabalho dos implementadores é, em parte, determinada pela completude e pela precisão da descrição da linguagem.

Por fim, os usuários da linguagem devem ser capazes de determinar como codificar soluções de software ao se referirem a um manual de referência da linguagem. Livros-texto e cursos entram nesse processo, mas os manuais de linguagem normalmente são as únicas fontes impressas oficialmente reconhecidas sobre uma linguagem.

O estudo de linguagens de programação, como o estudo de linguagens naturais, pode ser dividido em exames acerca da sintaxe e da semântica. A **sintaxe** de uma linguagem de programação é a forma de suas expressões, sentenças e unidades de programa. Sua **semântica** é o significado dessas expressões, sentenças e unidades de programas. Por exemplo, a sintaxe de uma sentença `while` em Java é

```
while (expressão_booleana) sentença
```

A semântica desse formato de sentença é que, quando o valor atual da expressão booleana for verdadeiro, a sentença dentro da estrutura será executada. O controle retorna implicitamente para a expressão booleana, para repetir o processo. Se a expressão booleana é falsa, o controle é transferido para a sentença que vem depois da construção **`while`**.

Apesar de normalmente serem separadas para propósitos de discussão, a sintaxe e a semântica são bastante relacionadas. Em uma linguagem de programação bem projetada, a semântica deve seguir diretamente a partir da sintaxe; ou seja, a aparência de uma sentença deve sugerir o que ela realiza.

Descrever a sintaxe é mais fácil que descrever a semântica, especialmente porque uma notação aceita universalmente está disponível para a descrição da sintaxe, mas nenhuma ainda foi desenvolvida para descrever semântica.

3.2 O PROBLEMA GERAL DE DESCREVER SINTAXE

Uma linguagem, seja natural (como a língua inglesa) ou artificial (como Java), é um conjunto de cadeias de caracteres formadas a partir de um alfabeto. As cadeias de uma linguagem são chamadas de **sentenças**. As regras sintáticas de uma linguagem especificam quais cadeias de caracteres formadas a partir do alfabeto estão na linguagem. A língua inglesa, por exemplo, possui uma coleção de regras extensa e complexa para especificar a sintaxe de suas sentenças. Em comparação, mesmo as maiores e mais complexas linguagem de programação são sintaticamente simples.

Descrições formais da sintaxe de linguagens de programação, por questões de simplicidade, normalmente não incluem descrições das unidades sintáticas de nível mais baixo. Essas pequenas unidades são denominadas **lexemas**. A descrição dos lexemas pode ser dada por uma especificação léxica, normalmente separada da descrição sintática da linguagem. Os lexemas de uma linguagem de programação incluem seus literais numéricos, operadores e palavras especiais, entre outros. Você pode pensar nos programas como sendo cadeias de lexemas, em vez de caracteres.

Os lexemas são divididos em grupos – por exemplo, os nomes de variáveis, os métodos, as classes, e assim por diante, formam um grupo chamado de *identificadores*. Cada grupo de lexemas é representado por um nome, ou *token*. Logo, um **token** de uma linguagem é uma categoria de seus lexemas. Por exemplo, um identificador é um *token* que pode ter lexemas, ou instâncias, como `soma` e `total`. Em alguns casos, um *token* tem apenas um lexema possível. Por exemplo, o *token* para o símbolo da operação aritmética + tem apenas um lexema possível. Considere a seguinte sentença Java:

```
index = 2 * count + 17;
```

Os lexemas e *tokens* dessa sentença são

Lexemas	*Tokens*
`index`	identificador
`=`	sinal_de_igualdade
`2`	literal_inteiro
`*`	operador_de_multiplicação
`count`	identificador
`+`	operador_de_adição

`17`	literal_inteiro
`;`	ponto e vírgula

Os exemplos de descrições de linguagem deste capítulo são muito simples, e a maioria inclui descrições de lexemas.

3.2.1 Reconhecedores de linguagens

Em geral, as linguagens podem ser formalmente definidas de duas maneiras: **reconhecimento** e **geração** (apesar de nenhuma delas fornecer uma definição prática por si só para que as pessoas tentem entender ou usar uma linguagem de programação). Suponha que tenhamos uma linguagem L que usa um alfabeto Σ de caracteres. Para definir L formalmente, usando o método de reconhecimento, precisamos construir um mecanismo R, chamado de dispositivo de reconhecimento, capaz de ler cadeias de caracteres do alfabeto Σ. R pode indicar se determinada cadeia de entrada está ou não em L. Dessa forma, R aceitaria ou rejeitaria a cadeia fornecida. Tais dispositivos são como filtros, separando sentenças válidas das incorretamente formadas. Se R, ao ser alimentado por qualquer cadeia de caracteres em Σ, aceita-a apenas se ela estiver em L, então R é uma descrição de L. Como a maioria das linguagens úteis é, para todos os efeitos, infinita, esse processo pode parecer longo e ineficiente. Os dispositivos de reconhecimento, entretanto, não são usados para enumerar todas as sentenças de uma linguagem – eles têm um propósito diferente.

A parte de análise sintática de um compilador é um reconhecedor para a linguagem que o compilador traduz. Nesse papel, o reconhecedor não precisa testar todas as possíveis cadeias de caracteres de algum conjunto para determinar se cada uma está na linguagem. Em vez disso, ele precisa determinar se programas informados estão na linguagem. Dessa forma, o analisador sintático determina se os programas informados são sintaticamente corretos. A estrutura dos analisadores sintáticos, também conhecidos como *parsers*, é discutida no Capítulo 4.

3.2.2 Geradores de linguagens

Um gerador de linguagens é um dispositivo usado para gerar as sentenças de uma linguagem. Podemos pensar em um gerador com um botão que produz uma sentença da linguagem cada vez que é pressionado. Como a sentença em particular é produzida por um gerador quando o botão é pressionado, ele parece um dispositivo de utilidade limitada como descritor de linguagens. Entretanto, as pessoas preferem certas formas de geradores, em vez de reconhecedores, porque elas podem ser lidas e entendidas mais facilmente. Em contraste, a porção de verificação de sintaxe de um compilador (um reconhecedor de linguagens) não é uma descrição de linguagem tão útil para um programador, porque ela pode ser usada apenas em modo de tentativa e erro. Por exemplo, para determinar a sintaxe correta de determinada sentença usando um compilador, o programador pode apenas submeter uma versão especulada e esperar para ver se o com-

pilador a aceita. Em contrapartida, normalmente é possível determinar se a sintaxe de uma sentença está correta comparando-a com a estrutura do gerador.

Existe uma forte conexão entre dispositivos formais de geração e reconhecimento para a mesma linguagem. Essa foi uma das descobertas fundamentais na ciência da computação e levou a muito do que hoje é conhecido sobre linguagens formais e sobre a teoria de projeto de compiladores. Retornamos ao relacionamento entre geradores e reconhecedores na próxima seção.

3.3 MÉTODOS FORMAIS PARA DESCREVER SINTAXE

Esta seção discute os mecanismos formais de geração de linguagem, geralmente chamados de **gramáticas**, usados para descrever a sintaxe das linguagens de programação.

3.3.1 Forma de Backus-Naur e gramáticas livres de contexto

Do meio para o final dos anos 1950, dois homens, Noam Chomsky e John Backus, em esforços de pesquisa não relacionados, desenvolveram o mesmo formalismo de descrição de sintaxe, que se tornou o método mais usado para descrever a sintaxe de linguagens de programação.

3.3.1.1 Gramáticas livres de contexto

Em meados dos anos 1950, Noam Chomsky, um linguista notável (entre outras coisas), descreveu quatro classes de dispositivos geradores, ou gramáticas, que definem quatro classes de linguagens (Chomsky, 1956, 1959). Duas dessas classes, chamadas de *livres de contexto* e *regulares*, foram úteis para descrever a sintaxe de linguagens de programação. A forma dos *tokens* das linguagens de programação pode ser descrita por expressões regulares. A sintaxe de uma linguagem de programação completa, com pequenas exceções, pode ser descrita por gramáticas livres de contexto. Como Chomsky é um linguista, seu interesse principal era na natureza teórica das linguagens naturais. Ele não tinha interesse nas linguagens artificiais usadas para comunicação com computadores. Então, demorou um tempo até seu trabalho ser aplicado às linguagens de programação.

3.3.1.2 Origens da Forma de Backus-Naur

Logo após o trabalho de Chomsky em classes de linguagens, o grupo ACM-GAMM começou o projeto de ALGOL 58. Um artigo fundamental que descreve ALGOL 58 foi apresentado por John Backus, um proeminente membro do grupo ACM-GAMM, em uma conferência internacional em 1959 (Backus, 1959). Esse artigo introduziu uma nova notação formal para a especificação de sintaxe de linguagem de programação. Posteriormente, a nova notação foi ligeiramente modificada por Peter Naur para a descrição de ALGOL 60 (Naur, 1960). O método revisado de descrição de sintaxe ficou conhecido como **Forma de Backus-Naur**, ou simplesmente **BNF**.

A BNF é uma notação natural para descrever sintaxe. Na verdade, algo similar a ela era usado por Panini para descrever a sintaxe de sânscrito, muitas centenas de anos antes de Cristo (Ingerman, 1967).

Apesar de o uso da BNF no relatório da ALGOL 60 não ser imediatamente aceito pelos usuários de computadores, ela rapidamente se tornou, e ainda é, o método mais popular para descrever a sintaxe de linguagens de programação de maneira concisa.

É notável que a BNF seja praticamente idêntica aos dispositivos de geração para linguagens livres de contexto, chamadas de **gramáticas livres de contexto**. No restante deste capítulo, nos referimos às gramáticas livres de contexto simplesmente como gramáticas. Além disso, os termos BNF e gramática são usados como sinônimos.

3.3.1.3 Fundamentos

Uma **metalinguagem** é uma linguagem usada para descrever outra. A BNF é uma metalinguagem para linguagens de programação.

A BNF usa abstrações para estruturas sintáticas. Uma operação de atribuição simples em Java, por exemplo, poderia ser representada pela abstração <assign> (os sinais de menor e maior são geralmente usados para delimitar os nomes de abstrações). A definição propriamente dita de <assign> pode ser

<assign> → <var> = <expression>

O texto no lado esquerdo da seta, chamado de **lado esquerdo** (LHS – *left-hand side*), é a abstração definida. O texto no lado direito da seta é a definição de LHS. É chamado de **lado direito** (RHS – *right-hand side*) e consiste em um misto de *tokens*, lexemas e referências a outras abstrações. (Na verdade, os *tokens* também são abstrações.) A definição completa é chamada de **regra** ou **produção**. No exemplo, as abstrações <var> e <expression> devem ser definidas para que a definição de <assign> seja útil.

Essa regra em particular especifica que a abstração <assign> é definida como uma instância da abstração <var>, seguida pelo lexema =, seguido por uma instância da abstração <expression>. Uma sentença de exemplo cuja estrutura sintática é descrita por essa regra é

```
total = subtotal1 + subtotal2
```

As abstrações em uma descrição BNF, ou gramática, são chamadas de **símbolos não terminais**, ou simplesmente **não terminais**, e os lexemas e *tokens* das regras são chamados de **símbolos terminais**, ou simplesmente **terminais**. Uma descrição BNF, ou **gramática**, é uma coleção de regras.

Símbolos não terminais podem ter duas ou mais definições, representando duas ou mais formas sintáticas possíveis na linguagem. Várias definições podem ser escritas como uma única regra, separadas pelo símbolo |, que significa OU lógico. Por exemplo, uma sentença **`if`** em Java pode ser descrita com as regras

<if_stmt> → **if** (<logic_expr>) <stmt>
<if_stmt> → **if** (<logic_expr>) <stmt> **else** <stmt>

ou com a regra

<if_stmt> → **if** (<logic_expr>) <stmt>
| **if** (<logic_expr>) <stmt> **else** <stmt>

Nessas regras, <stmt> representa uma sentença única ou uma sentença composta.

Apesar de a BNF ser simples, é suficientemente poderosa para descrever praticamente toda a sintaxe das linguagens de programação. Em particular, pode descrever listas de construções similares, a ordem na qual as diferentes construções devem aparecer, estruturas aninhadas com qualquer profundidade e até expressar precedência e associatividade de operadores.

3.3.1.4 Descrição de listas

Listas de tamanho variável em matemática são geralmente escritas com reticências (. . .); 1, 2, . . . é um exemplo. A BNF não inclui reticências; portanto, um método alternativo é necessário para descrever listas de elementos sintáticos em linguagens de programação (por exemplo, uma lista de identificadores que aparecem em uma sentença de declaração de dados). Para a BNF, a alternativa é a recursão. Uma regra é **recursiva** se seu lado esquerdo aparece em seu lado direito. A seguinte regra ilustra como a recursão é usada para descrever listas:

<ident_list> → identifier
| identifier, <ident_list>

Essa regra define uma lista de identificadores (<ident_list>) como um *token* isolado (identificador) ou um identificador seguido por uma vírgula e outra instância de <ident_list>. A recursão é usada para descrever listas em muitas das gramáticas de exemplo neste capítulo.

3.3.1.5 Gramáticas e derivações

Uma gramática é um dispositivo de geração para definir linguagens. As sentenças da linguagem são geradas por meio de uma sequência de aplicação de regras, começando com um não terminal especial da gramática chamado de **símbolo inicial**. Essa sequência de aplicações de regras é denominada **derivação**. Em uma gramática para uma linguagem de programação completa, o símbolo inicial representa um programa completo e é chamado de <program>. A gramática simples mostrada no Exemplo 3.1 é usada para ilustrar derivações.

Exemplo 3.1 Uma gramática para uma pequena linguagem.

<program> → **begin** <stmt_list> **end**

<stmt_list> → <stmt>
| <stmt> ; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
| <var> – <var>
| <var>

A linguagem escrita pela gramática do Exemplo 3.1 tem apenas uma forma sentencial: atribuição. Um programa consiste na palavra especial **begin**, seguida por uma lista de sentenças separadas por ponto e vírgula, seguida da palavra especial **end**. Uma expressão é uma única variável ou duas variáveis separadas por um operador + ou −. Os únicos nomes de variáveis nessa linguagem são A, B e C.

A seguir, temos uma derivação de um programa nessa linguagem:

<program> => **begin** <stmt_list> **end**
=> **begin** <stmt> ; <stmt_list> **end**
=> **begin** <var> = <expression> ; <stmt_list> **end**
=> **begin** A = <expression> ; <stmt_list> **end**
=> **begin** A = <var> + <var> ; <stmt_list> **end**
=> **begin** A = B + <var> ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt_list> **end**
=> **begin** A = B + ; <stmt> **end**
=> **begin** A = B + C ; <var> = <expression> **end**
=> **begin** A = B + C ; B = <expression> **end**
=> **begin** A = B + C ; B = <var> **end**
=> **begin** A = B + C ; B = C **end**

Essa derivação, como todas, começa com o símbolo inicial, nesse caso <program>. O símbolo => é lido como "deriva". Cada cadeia sucessiva na sequência é derivada da cadeia anterior, substituindo um dos não terminais por uma das definições de não terminais. Cada uma das cadeias na derivação, inclusive <program>, é chamada de **forma sentencial**.

Nessa derivação, o não terminal substituído é sempre o mais à esquerda na forma sentencial anterior. Derivações que usam essa ordem de substituição são chamadas de **derivações mais à esquerda**. A derivação continua até que a forma sentencial não contenha mais nenhum não terminal. Essa forma sentencial, que consiste apenas em terminais, ou lexemas, é a sentença gerada.

Além de uma derivação poder ser mais à esquerda, ela também pode ser mais à direita ou em qualquer ordem que não seja mais à esquerda ou mais à direita. A ordem de derivação não tem efeito na linguagem gerada por uma gramática.

Ao se escolher lados direitos alternativos de regras para a substituição de não terminais na derivação, diferentes sentenças podem ser geradas. Por meio da escolha exaustiva de todas as combinações, a linguagem inteira pode ser gerada. Essa linguagem, como a maioria das outras, é infinita, então ninguém pode gerar *todas* as sentenças em tempo finito.

O Exemplo 3.2 apresenta uma gramática para parte de uma linguagem de programação típica.

Exemplo 3.2 Uma gramática para sentenças de atribuição simples.

<assign> → <id> = <expr>
<id> → A| B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

A gramática do Exemplo 3.2 descreve sentenças de atribuição cujo lado direito é composto de expressões aritméticas com operadores de multiplicação e adição, assim como parênteses. Por exemplo, a sentença

```
A = B * (A + C)
```

é gerada pela derivação mais à esquerda:

<assign> => <id> = <expr>
=> A = <expr>
=> A = <id> * <expr>
=> A = B * <expr>
=> A = B * (<expr>)
=> A = B * (<id> + <expr>)
=> A = B * (A + <expr>)
=> A = B * (A + <id>)
=> A = B * (A + C)

3.3.1.6 Árvores de análise sintática

Um dos recursos mais atraentes das gramáticas é que elas descrevem naturalmente a estrutura hierárquica estática das sentenças das linguagens que definem. Essas estruturas hierárquicas são chamadas de árvores de análise sintática (*parse trees*). Por exemplo, a árvore de análise sintática na Figura 3.1 mostra a estrutura da sentença de atribuição derivada anteriormente.

Cada nó interno de uma árvore de análise sintática é rotulado com um símbolo não terminal; cada folha é rotulada com um símbolo terminal. Cada subárvore de uma árvore de análise sintática descreve uma instância de uma abstração da sentença.

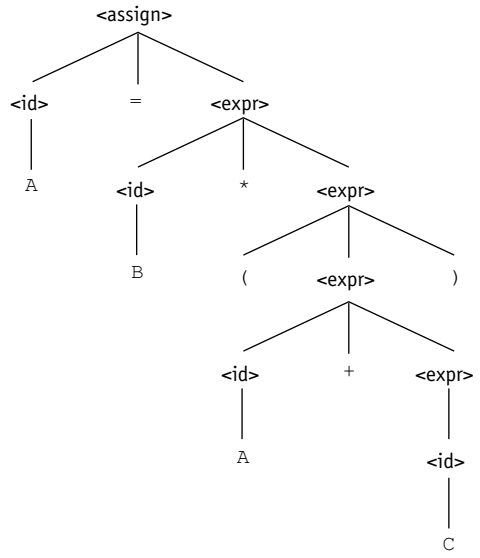


FIGURA 3.1
Uma árvore de análise sintática para a sentença simples `A = B * (A + C).`

3.3.1.7 Ambiguidade

Uma gramática que gera uma forma sentencial para a qual existem duas ou mais árvores de análise sintática é dita **ambígua**. Considere a gramática mostrada no Exemplo 3.3, uma pequena variação da gramática mostrada no Exemplo 3.2.

Exemplo 3.3 Uma gramática ambígua para sentenças de atribuição simples.

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
        |  <expr> * <expr>
        |  ( <expr>)
        |  <id>
```

A gramática do Exemplo 3.3 é ambígua porque a sentença

```
A = B + C * A
```

tem duas árvores de análise sintática distintas, conforme mostrado na Figura 3.2. A ambiguidade ocorre porque a gramática especifica um pouco menos de estrutura sintática que a gramática do Exemplo 3.2. Em vez de permitir que a árvore de análise sintática de uma expressão cresça apenas para a direita, essa gramática permite crescimento tanto para a esquerda quanto para a direita.

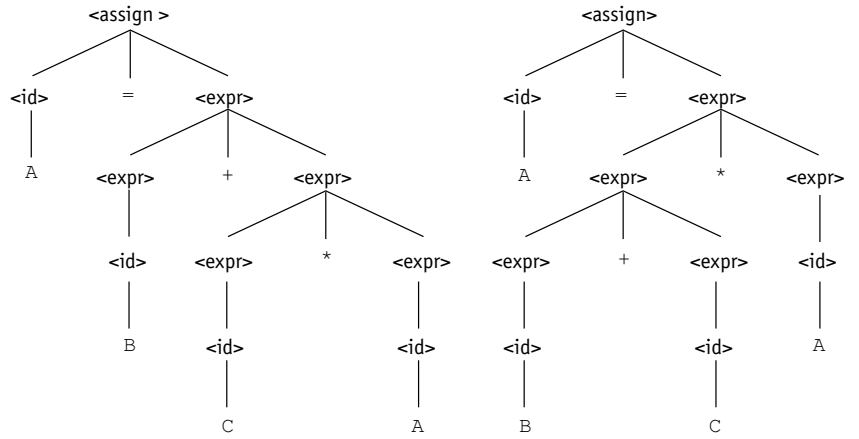


FIGURA 3.2
Duas árvores diferentes de análise sintática para a mesma sentença, `A = B + C * A`.

A ambiguidade sintática das estruturas de linguagem é um problema, porque os compiladores normalmente baseiam sua semântica nessas estruturas em sua forma sintática. Especificamente, o compilador escolhe o código a ser gerado para uma sentença examinando sua árvore de análise sintática. Se uma estrutura de linguagem tem mais de uma árvore de análise sintática, o significado da estrutura não pode ser determinado de modo único. Esse problema é discutido em dois exemplos específicos nas seções seguintes.

Existem várias outras características de uma gramática que às vezes são úteis para determinar se ela é ambígua.[1] Elas incluem as seguintes: (1) se a gramática gera uma sentença com mais de uma derivação mais à esquerda e (2) se a gramática gera uma sentença com mais de uma derivação mais à direita.

Alguns algoritmos de análise sintática podem ser baseados em gramáticas ambíguas. Quando o analisador sintático encontra uma construção ambígua, ele usa informação não gramatical fornecida pelo projetista para construir a árvore de análise sintática correta. Em muitos casos, uma gramática ambígua pode ser reescrita para ser não ambígua, mas ainda assim gerar a linguagem desejada.

3.3.1.8 Precedência de operadores

Quando uma expressão inclui dois operadores, por exemplo, `x + y * z`, uma questão semântica óbvia é a ordem de avaliação desses operadores (por exemplo, nessa expressão ocorre adição e depois multiplicação, ou vice-versa?). Essa questão pode ser respondida pela atribuição de diferentes níveis de precedência aos operadores. Por exemplo, se para o operador `*` for atribuída uma precedência mais alta do que para o operador `+`

[1]Observe que é matematicamente impossível determinar se uma gramática arbitrária é ambígua.

(pelo projetista da linguagem), a multiplicação será efetuada primeiro, independente da ordem de aparição dos dois na expressão.

Conforme mencionado, uma gramática pode descrever certa estrutura sintática de forma que parte do significado dessa estrutura possa ser determinada a partir de sua árvore de análise sintática. Em particular, o fato de um operador em uma expressão aritmética ser gerado abaixo na árvore de análise sintática (e, dessa forma, devendo ser avaliado primeiro) pode ser usado para indicar que ele tem precedência sobre um operador produzido mais acima. Na primeira árvore de análise sintática da Figura 3.2, por exemplo, o operador de multiplicação é gerado mais abaixo, o que poderia indicar que ele tem precedência sobre o operador de adição na expressão. A segunda árvore de análise sintática, entretanto, indica o oposto. Parece, então, que as duas árvores de análise sintática têm informações conflitantes.

Note que, apesar de a gramática do Exemplo 3.2 não ser ambígua, a ordem de precedência de seus operadores não é a usual. Nela, uma árvore de análise sintática de uma sentença com operadores múltiplos, independentemente dos operadores envolvidos, tem o operador mais à direita na expressão no ponto mais baixo da árvore de análise sintática, com os outros se movendo progressivamente mais para cima, à medida que nos movemos para a esquerda na expressão. Por exemplo, na expressão `A + B * C`, `*` é o mais baixo na árvore, o que indica que esse operador deve ser usado primeiro. Entretanto, na expressão `A * B + C`, + é o mais baixo.

Uma gramática para as expressões simples que estamos discutindo pode ser escrita de forma a ser tanto não ambígua quanto a especificar uma precedência coerente dos operadores + e `*`, independentemente da ordem em que eles aparecem em uma expressão. A ordem correta é especificada por meio de símbolos não terminais separados para representar os operandos dos operadores que têm precedência diferente. Isso requer não terminais adicionais e algumas regras novas. Em vez de usar <expr> para ambos os operandos de + e `*`, podemos usar três não terminais para representar os operandos, o que permite que a gramática force operadores diferentes para níveis diferentes na árvore de análise sintática. Se <expr> é símbolo raiz para expressões, + pode ser forçado ao topo da árvore de análise sintática ao fazermos com que <expr> gere diretamente apenas operadores +, usando o novo não terminal, <term>, como o operando à direita de +. Em seguida, podemos definir <term> para gerar operadores `*`, usando <term> como o operando da esquerda e um novo não terminal, <factor>, como o da direita. Agora, `*` vai sempre estar mais abaixo na árvore de análise sintática, simplesmente porque está mais longe do símbolo inicial que o + em cada derivação. Veja o Exemplo 3.4.

Exemplo 3.4 Uma gramática não ambígua para expressões.

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
       | <term>
<term> → <term> * <factor>
       | <factor>
<factor> → (<expr>)
         | <id>
```

A gramática do Exemplo 3.4 gera a mesma linguagem que as dos Exemplos 3.2 e 3.3, mas não é ambígua e especifica a ordem de precedência usual para os operadores de multiplicação e adição. A derivação da sentença `A = B + C * A`, a seguir, usa a gramática do Exemplo 3.4:

```
<assign> => <id> = <expr>
         => A = <expr>
         => A = <expr> + <term>
         => A = <term> + <term>
         => A = <factor> + <term>
         => A = <id> + <term>
         => A = B + <term>
         => A = B + <term> * <factor>
         => A = B + <factor> * <factor>
         => A = B + <id> * <factor>
         => A = B + C * <factor>
         => A = B + C * <id>
         => A = B + C * A
```

A árvore de análise sintática única para essa sentença, usando a gramática do Exemplo 3.4, é mostrada na Figura 3.3.

A conexão entre árvores de análise sintática e derivações é muito estreita: qualquer uma delas pode ser facilmente construída a partir da outra. Cada derivação com uma gramática não ambígua tem uma única árvore de análise sintática, apesar de ela poder ser representada por derivações diferentes. Por exemplo, a seguinte derivação da sentença `A = B + C * A` é diferente da derivação da mesma sentença dada anteriormente. Essa é uma derivação mais à direita, enquanto a anterior era mais à esquerda. Ambas as derivações são representadas pela mesma árvore de análise sintática.

```
<assign> => <id> = <expr>
         => <id> = <expr> + <term>
         => <id> = <expr> + <term> * <factor>
         => <id> = <expr> + <term> * <id>
         => <id> = <expr> + <term> * A
         => <id> = <expr> + <factor> * A
         => <id> = <expr> + <id> * A
         => <id> = <expr> + C * A
         => <id> = <term> + C * A
         => <id> = <factor> + C * A
         => <id> = <id> + C * A
         => <id> = B + C * A
         => A = B + C * A
```

3.3.1.9 Associatividade de operadores

Quando uma expressão inclui dois operadores com a mesma precedência (como `*` e `/` normalmente têm) – por exemplo, `A / B * C` –, uma regra semântica é necessária

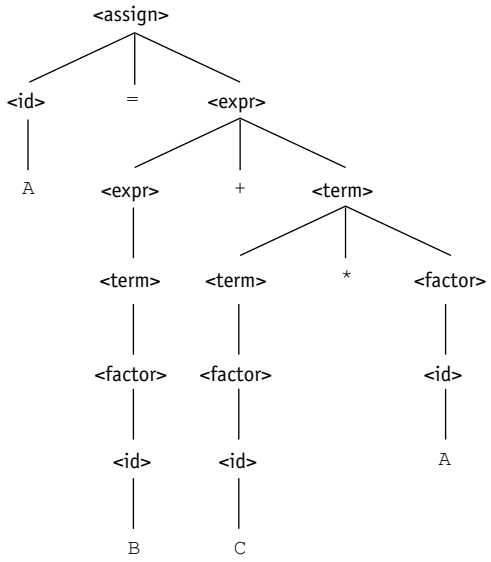


FIGURA 3.3
A árvore de análise sintática única para `A = B + C * A` usando uma gramática não ambígua.

para especificar qual dos operadores deve ter precedência.[2] Essa regra é chamada de *associatividade*.

Como era o caso com a precedência, uma gramática para expressões pode definir a associatividade de operadores corretamente. Considere o exemplo de sentença de atribuição a seguir:

```
A = B + C + A
```

A árvore de análise sintática para essa sentença, conforme definida com a gramática do Exemplo 3.4, é apresentada na Figura 3.4.

Ela mostra o operador de adição da esquerda mais baixo que o operador de adição da direita. Essa é a ordem correta, se a intenção era deixar a adição associativa. Em muitos casos, a associatividade da adição em um computador é irrelevante. Na matemática, a adição é associativa, ou seja, as ordens de avaliação associativa à esquerda e à direita significam a mesma coisa. Ou seja, `(A + B) + C = A + (B + C)`. A adição de ponto flutuante em um computador, entretanto, não é necessariamente associativa. Suponha que os valores de ponto flutuante armazenem sete dígitos de precisão. Considere o problema de somar 11 números, sendo que um deles é 10^7 e os outros 10 são 1. Se os números pequenos (os 1s) são somados ao grande, um de cada vez, não há efeitos nesse número, porque os pequenos ocorrem no oitavo dígito do grande. Entretanto, se os números pequenos são somados primeiro e o resultado é somado ao número grande, o resultado em uma precisão de sete dígitos é $1{,}000001 * 10^7$. A subtração e a divisão

[2]Uma expressão com duas ocorrências do mesmo operador tem o mesmo problema; por exemplo, `A / B / C`.

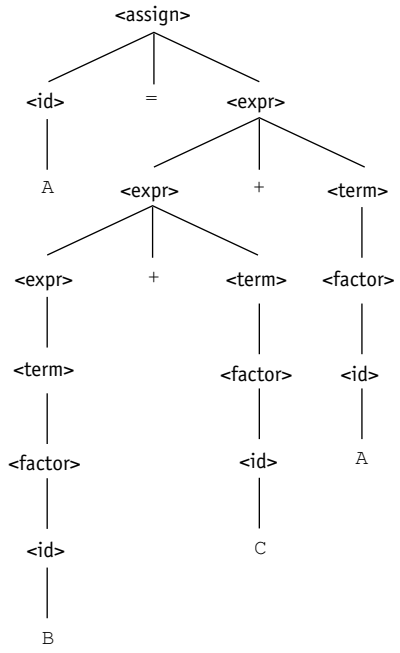


FIGURA 3.4
Uma árvore de análise sintática para `A = B + C + A` ilustrando a associatividade da adição.

não são associativas, seja na matemática ou em um computador. Logo, a associatividade correta pode ser essencial para uma expressão que contenha qualquer uma delas.

Quando uma regra gramatical tem seu lado esquerdo aparecendo também no início de seu lado direito, diz-se que ela é **recursiva à esquerda**. Essa recursão à esquerda especifica associatividade à esquerda. Por exemplo, a recursão à esquerda das regras da gramática do Exemplo 3.4 faz com que tanto a adição quanto a multiplicação sejam associativas à esquerda. Infelizmente, a recursão à esquerda não permite o uso de alguns algoritmos de análise sintática importantes. Quando um desses algoritmos é usado, a gramática deve ser modificada para remover a recursão à esquerda. Isso, por sua vez, impede a gramática de especificar precisamente que certos operadores são associativos à esquerda. Felizmente, a associatividade à esquerda pode ser forçada pelo compilador, mesmo que a gramática não exija isso.

Na maioria das linguagens que o fornecem, o operador de exponenciação é associativo à direita. Para indicar associatividade à direita, pode ser usada a recursão à direita. Uma regra gramatical é **recursiva à direita** se o lado esquerdo aparece bem no final do lado direito. Regras como

```
<factor> → <exp> ** <factor>
         | <exp>
<exp> → (<expr>)
      | id
```

podem ser usadas para descrever exponenciação como um operador associativo à direita.

3.3.1.10 Uma gramática não ambígua para `if-else`

As regras BNF para uma sentença **`if-else`** em Java são:

<if_stmt> → **`if`** (<logic_expr>) <stmt>
`if` (<logic_expr>) <stmt> **`else`** <stmt>

Se tivéssemos também <stmt> → <if_stmt>, essa gramática seria ambígua. A forma sentencial mais simples que ilustra essa ambiguidade é

`if` (<logic_expr>) **`if`** (<logic_expr>) <stmt> **`else`** <stmt>

As duas árvores de análise sintática na Figura 3.5 mostram a ambiguidade dessa forma sentencial. Considere o seguinte exemplo dessa construção:

```
if (done == true)
if (denom == 0)
  quotient = 0;
  else quotient = num / denom;
```

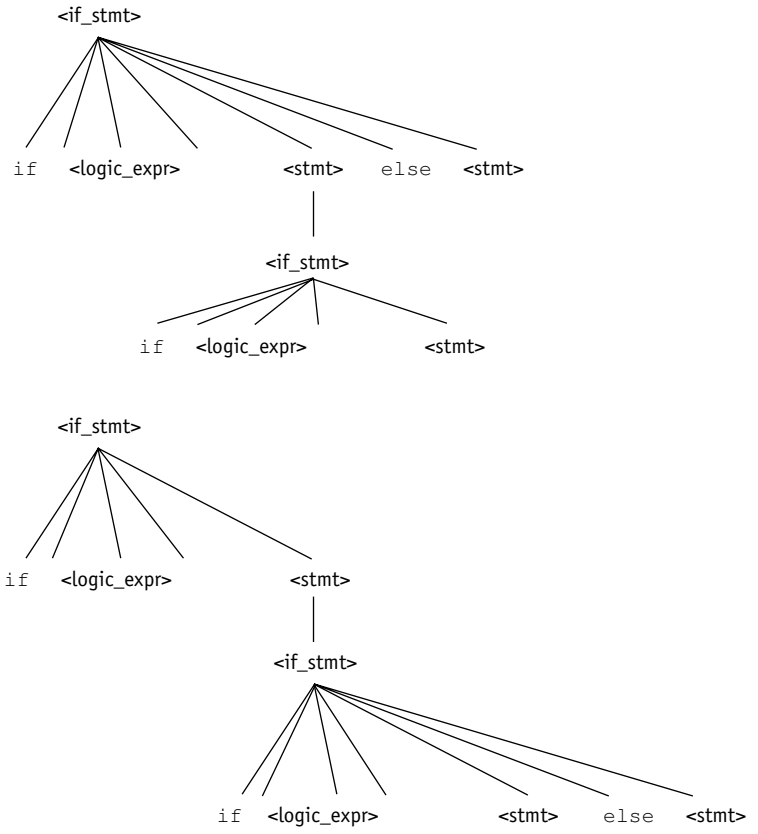


FIGURA 3.5
Duas árvores de análise sintática para a mesma forma sentencial.

O problema é que, se a árvore de análise sintática da Figura 3.5 for usada como base para a tradução, a cláusula senão (*else*) será executada quando `done` não for verdadeira, o que provavelmente não é o que o autor da construção pretendia. Examinaremos os problemas práticos relacionados a esse de associação com o senão no Capítulo 8.

Desenvolveremos agora uma gramática não ambígua que descreve essa sentença **`if`**. A regra para as construções **`if`** em muitas linguagens é que uma cláusula else, quando presente, casa com a cláusula then mais próxima que ainda não está casada. Dessa forma, não pode existir uma sentença **`if`** sem um **`else`** entre uma cláusula then e seu **`else`** correspondente. Então, para essa situação, as sentenças devem ser distinguidas entre aquelas que estão casadas e as que não estão, e as sentenças que não estão casadas são **`if`**s sem **`else`**. Todas as outras sentenças são casadas. O problema da primeira gramática é que ela trata todas as sentenças como se tivessem importância sintática igual – ou seja, como se todas casassem.

Para refletir sobre as diferentes categorias de sentenças, diferentes abstrações, ou não terminais, devem ser usadas. A gramática não ambígua baseada nessas ideias é:

<stmt> → <matched> | <unmatched>
<matched> → **`if`** (<logic_expr>) <matched> **`else`** <matched>
| qualquer sentença, fora if
<unmatched> → **`if`** (<logic_expr>) <stmt>
|**`if`** (<logic_expr>) <matched> **`else`** <unmatched>

Existe apenas uma árvore de análise sintática possível, usando essa gramática, para a seguinte forma sentencial:

`if` (<logic_expr>) **`if`** (<logic_expr>) <stmt> **`else`** <stmt>

3.3.2 BNF estendida

Por causa de algumas pequenas inconveniências na BNF, ela tem sido estendida de diversas maneiras. A maioria das versões estendidas é chamada de BNF Estendida, ou simplesmente EBNF, mesmo não sendo todas exatamente iguais. As extensões não aumentam o poder descritivo da BNF; apenas sua legibilidade e facilidade de escrita.

Três extensões são comumente incluídas nas versões de EBNF. A primeira delas denota uma parte opcional de um lado direito, delimitada por colchetes. Por exemplo, uma sentença **`if-else`** em C pode ser descrita como

<if_stmt> → **`if`** (<expression>) <statement> [**`else`** <statement>]

Sem o uso dos colchetes, a descrição sintática dessa sentença necessitaria das duas regras a seguir:

<if_stmt> → **`if`** (<expression>) <statement>
|**`if`** (<expression>) <statement> **`else`** <statement>

A segunda extensão é o uso de chaves em um lado direito para indicar que a parte envolta em chaves pode ser repetida indefinidamente ou deixada de fora. Essa extensão permite construir listas com uma regra, em vez de usar recursão e duas regras. Por exemplo, uma lista de identificadores separados por vírgulas pode ser descrita pela seguinte regra:

<ident_list> → <identifier> {, <identifier>}

Essa é uma substituição ao uso de recursão por meio de uma iteração implícita; a parte envolta em chaves pode ser iterada qualquer número de vezes.

A terceira extensão comum trata de opções de múltipla escolha. Quando um elemento deve ser escolhido de um grupo, as opções são colocadas em parênteses e separadas pelo operador OU, |. Por exemplo,

<term> → <term> (* | / | %) <factor>

Em BFN, uma descrição desse <term> precisaria das três regras a seguir:

<term> → <term> * <factor>
| <term> / <factor>
| <term> % <factor>

Os colchetes, chaves e parênteses nas extensões EBNF são **metassímbolos**, ou seja, são ferramentas notacionais e não símbolos terminais nas entidades sintáticas que ajudam a descrever. Nos casos em que esses metassímbolos também são símbolos terminais na linguagem descrita, as instâncias que são símbolos terminais devem ser destacadas (sublinhadas ou colocadas entre aspas). O Exemplo 3.5 ilustra o uso de chaves e múltipla escolha em uma gramática EBNF.

Exemplo 3.5 Versões BNF e EBNF de uma gramática de expressões.

BNF:

<expr> → <expr> + <term>
| <expr> – <term>
| <term>
<term> → <term> * <factor>
| <term> / <factor>
| <factor>
<factor> → <exp> ** <factor>
<exp>
<exp> → (<expr>)
| id

EBNF:

<expr> → <term> { (+ | –) <term>}
<term> → <factor> { (* | /) <factor>}
<factor> → <exp> { ** <exp>}
<exp> → (<expr>)
| id

A regra BNF

<expr> → <expr> + <term>

claramente especifica – de fato, força – que o operador + seja associativo à esquerda. Entretanto, a versão EBNF,

<expr> → <term> {+ <term>}

não força a direção da associatividade. Esse problema é resolvido em um analisador sintático baseado em uma gramática EBNF para expressões, projetando-se o processo do analisador sintático para garantir a associatividade correta. Isso é discutido mais detalhadamente no Capítulo 4.

Algumas versões de EBNF permitem anexar um valor numérico à chave direita para indicar um limite superior para o número de vezes em que as partes envoltas pelas chaves podem ser repetidas. Além disso, algumas versões usam um sinal de adição (+) sobrescrito para indicar uma ou mais repetições. Por exemplo,

<compound> → **begin** <stmt> {<stmt>} **end**

e

<compound> → **begin** {<stmt>}$^{+}$ **end**

são equivalentes.

Nos últimos anos, têm aparecido algumas variações na BNF e na EBNF. Entre elas, estão:

- Em vez da seta é usado um ponto e vírgula, e o RHS é colocado na próxima linha.
- Em vez de uma barra vertical para separar RHSs alternativos, eles são colocados em linhas separadas.
- Em vez de colchetes para indicar que algo é opcional, o subscrito opt é usado. Por exemplo,

  DeclaradorDeConstrutor → NomeSimples (ListaDeParâmetrosFormais$_{opt}$)
- Em vez do símbolo | em uma lista de elementos entre parênteses para indicar uma escolha, as palavras "one of" são usadas. Por exemplo,

  OperadorDeAtribuição → um entre = *= /= %= += -=
  <<= >>= &= ^= |=

Existe um padrão para EBNF, o ISO/IEC 14977:1996 (1996), mas é raramente usado. O padrão usa o sinal de igualdade (=), em vez de uma seta, em regras, termina cada RHS com um ponto e vírgula e exige aspas em todos os símbolos terminais. Ele também especifica inúmeras outras regras de notação.

3.3.3 Gramáticas e reconhecedores

Neste capítulo, sugerimos que existe uma forte relação entre dispositivos de geração e reconhecimento para uma linguagem. De fato, dada uma gramática livre de contexto, um reconhecedor para a linguagem gerada pela gramática pode ser construído algoritmicamente. Foram desenvolvidos alguns sistemas de software para realizar essa construção. Tais sistemas permitem a rápida criação da parte de análise sintática de um compilador para uma nova linguagem e, dessa forma, são bastante valiosos. Um dos primeiros desses geradores de analisador sintático se chama yacc (de *y*et *a*nother *c*ompiler *c*ompiler – ainda outro compilador) (Johnson, 1975). Existem agora muitos sistemas disponíveis.

» *nota histórica*

As gramáticas de atributos são usadas em uma ampla variedade de aplicações. Para fornecer descrições completas da sintaxe e semântica estática de linguagens de programação (Watt, 1979), como a definição formal de uma linguagem que pode ser informada para um sistema de geração de compiladores (Farrow, 1982) e como a base de diversos sistemas de edição dirigidos por sintaxe (Teitelbaum e Reps, 1981; Fischer et al., 1984). Além disso, gramáticas de atributos são usadas em sistemas de processamento de linguagem natural (Correa, 1992).

3.4 GRAMÁTICAS DE ATRIBUTOS

Uma **gramática de atributos** é um dispositivo usado para descrever mais sobre a estrutura de uma linguagem de programação do que poderia ser descrito com uma gramática livre de contexto. Uma gramática de atributos é a extensão de uma gramática livre de contexto. A extensão permite descrever convenientemente certas regras da linguagem, como a compatibilidade de tipos. Antes de podermos definir a forma das gramáticas de atributos, precisamos deixar claro o conceito de semântica estática.

3.4.1 Semântica estática

Algumas características das linguagens de programação são difíceis de descrever com BNF –algumas são impossíveis. Como exemplo de regra sintática difícil de especificar com uma BNF, considere as regras de compatibilidade de tipos. Em Java, por exemplo, um valor de ponto flutuante não pode ser atribuído a uma variável do tipo inteiro, apesar de o contrário ser permitido. Embora seja possível especificar essa restrição em BNF, ela requer símbolos não terminais e regras adicionais. Se todas as regras de tipos em Java fossem especificadas em BNF, a gramática se tornaria extensa demais para ser útil, porque o tamanho da gramática determina o tamanho do analisador sintático.

Como exemplo de uma regra sintática que não pode ser especificada em BNF, considere a regra comum de que todas as variáveis devem ser declaradas antes de serem referenciadas. Foi provado que essa regra não pode ser especificada em BNF.

Tais problemas exemplificam as categorias de regras de linguagem chamadas de regras semânticas. A **semântica estática** de uma linguagem é apenas indiretamente relacionada ao significado dos programas durante a execução; em vez disso, ela tem a ver com as formas permitidas dos programas (sintaxe, em vez da semântica). Muitas regras de semântica estática de uma linguagem definem suas restrições de tipo. A semântica estática é assim chamada porque a análise necessária para verificar essas especificações pode ser feita em tempo de compilação.

Devido aos problemas da descrição de semântica estática com BNF, uma variedade de mecanismos mais poderosos foi criada para essa tarefa. Um desses mecanismos, as gramáticas de atributos, foi projetado por Knuth (1968a) para descrever tanto a sintaxe quanto a semântica estática de programas.

As gramáticas de atributos são uma abordagem formal, usada tanto para descrever quanto para verificar a corretude das regras de semântica estática de um programa. Apesar de elas não serem sempre usadas de maneira formal no projeto de compiladores, os conceitos básicos das gramáticas de atributos são ao menos informalmente usados em todos os compiladores (consulte Aho et al., 1986).

A semântica dinâmica, que é o significado de expressões, sentenças e unidades de programas, é discutida na Seção 3.5.

3.4.2 Conceitos básicos

Gramáticas de atributos são gramáticas livres de contexto nas quais foram adicionados atributos, funções de computação de atributos e funções de predicado. **Atributos**, associados a símbolos de gramáticas (os símbolos terminais e não terminais), são similares às variáveis no sentido de poderem ter valores atribuídos a eles. **Funções de computação de atributos**, algumas vezes chamadas de funções semânticas, são associadas às regras gramaticais. Elas são usadas para especificar como os valores de atributos são computados. **Funções de predicado**, as quais descrevem as regras de semântica estática da linguagem, são associadas às regras gramaticais.

Esses conceitos se tornarão mais claros após definirmos formalmente as gramáticas de atributos e fornecermos um exemplo.

3.4.3 Definição de gramáticas de atributo

Uma gramática de atributo tem os seguintes recursos adicionais:

- Associado a cada símbolo X da gramática está um conjunto de atributos A(X). O conjunto A(X) é formado por dois conjuntos disjuntos S(X) e I(X), chamados de atributos sintetizados e atributos herdados, respectivamente. **Atributos sintetizados** são usados para passar informações semânticas para cima em uma árvore de análise sintática, enquanto **atributos herdados** passam informações semânticas para baixo e através de uma árvore.
- Associado a cada regra gramatical está um conjunto de funções semânticas e um conjunto, possivelmente vazio, de funções de predicado sobre os atributos dos símbolos na regra gramatical. Para uma regra $X_0 \rightarrow X_1 \ldots X_n$, os atributos sintetizados de X_0 são computados com funções semânticas da forma $S(X_0) = f(A(X_1), \ldots, A(X_n))$. Então, o valor de um atributo sintetizado em um nó de uma árvore de análise sintática depende apenas dos valores dos atributos dos nós filhos desse nó. Os atributos herdados de símbolos X_j, $1 \leq j \leq n$ (na regra acima) são computados com uma função semântica da forma $I(X_j) = f(A(X_0), \ldots, A(X_n))$. Então, o valor de um atributo herdado em um nó de uma árvore de análise sintática depende dos valores dos atributos do seu nó pai e de seus nós irmãos. Note que, para evitar circularidade, os atributos herdados geralmente são restritos a funções na forma $I(X_j) = f(A(X_0), \ldots, A(X_{(j-1)}))$. Essa forma evita que um atributo herdado dependa de si mesmo ou de tributos à direita na árvore de análise sintática.
- Uma função de predicado tem a forma de uma expressão booleana na união do conjunto de atributos $\{A(X_0), \ldots, A(X_n)\}$ e um conjunto de valores de atributo literais. As únicas derivações permitidas com uma gramática de atributos são aquelas nas quais cada predicado associado a cada não terminal é verdadeiro. Um valor falso para a função de predicado indica uma violação das regras de sintaxe ou de semântica estática da linguagem.

Uma árvore de análise sintática de uma gramática de atributos é a árvore de análise sintática baseada em sua gramática BNF associada, com um conjunto possivelmente vazio de valores de atributos anexado a cada nó. Se todos os valores de atributos em uma árvore de análise sintática forem computados, diz-se que a árvore é **completamente**

atribuída. Apesar de, na prática, isso não ser sempre feito dessa forma, é conveniente pensar nos valores de atributos sendo computados após a árvore de análise sintática sem atributos completa ser construída pelo compilador.

3.4.4 Atributos intrínsecos

Atributos intrínsecos são sintetizados de nós-folha cujos valores são determinados fora da árvore de análise sintática. Por exemplo, o tipo de uma instância de uma variável em um programa poderia vir da tabela de símbolos, a qual é usada para armazenar nomes de variáveis e seus tipos. O conteúdo da tabela de símbolos é definido com base nas sentenças de declaração anteriores. Inicialmente, supondo que uma árvore de análise sintática não atribuída foi construída e que os valores dos atributos são necessários, os únicos atributos com valores são os intrínsecos dos nós-folha. Dados os valores dos atributos intrínsecos em uma árvore de análise sintática, as funções semânticas podem ser usadas para computar os valores de atributos restantes.

3.4.5 Exemplos de gramáticas de atributos

Como um exemplo simples de como as gramáticas de atributos podem ser usadas para descrever semântica estática, considere o seguinte fragmento de gramática de atributo que descreve a regra que diz que o nome de um **`end`** de um procedimento em Ada deve corresponder ao nome do procedimento. (Essa regra não pode ser expressa em BNF.) O atributo string de <proc_name>, denotado por <proc_name>.string, é a cadeia de caracteres encontrada pelo compilador imediatamente após a palavra reservada **`procedure`**. Note que, quando existe mais de uma ocorrência de um não terminal em uma regra sintática em uma gramática de atributos, os não terminais são subscritos com colchetes para serem distinguidos. Nem os subscritos nem os colchetes fazem parte da linguagem descrita.

Regra sintática: <proc_def> → **`procedure`** <proc_name>[1]
<proc_body> **`end`** <proc_name>[2];
Predicado: <proc_name>[1]string == <proc_name>[2].string

Nesse exemplo, a regra de predicado diz que o nome no atributo string do não terminal <proc_name>, no cabeçalho do subprograma, deve corresponder ao nome no atributo string do não terminal <proc_name> após o final do subprograma.

A seguir, consideramos um exemplo maior de gramática de atributos. Ele ilustra como uma gramática de atributos pode ser usada para verificar as regras de tipo de uma sentença de atribuição simples. A sintaxe e a semântica estática dessa sentença de atribuição são as seguintes: os únicos nomes de variáveis são `A`, `B` e `C`. O lado direito das atribuições podem ser uma variável ou uma expressão na forma de uma variável adicionada a outra. As variáveis podem ser do tipo inteiro ou real. Quando existem duas variáveis no lado direito de uma atribuição, elas não precisam ser do mesmo tipo. Quando os tipos dos operandos não são os mesmos, o tipo da expressão é sempre real. Quando os tipos são os mesmos, o da expressão é o mesmo dos operandos. O tipo do lado esquerdo da atribuição deve s com o do lado direito. Então, os tipos dos operandos do lado direito podem ser mistos, mas a atribuição só é válida se o alvo e o valor resultante da avaliação

do lado direito são do mesmo tipo. A gramática de atributos especifica essas regras de semântica estática.

A porção sintática de nossa gramática de atributos de exemplo é

<assign> → <var> = <expr>
<expr> → <var> + <var>
| <var>
<var> → A | B | C

Os atributos para os não terminais na gramática de atributos de exemplo são descritos nos parágrafos a seguir:

- *actual_type* – Um atributo sintetizado associado aos não terminais <var> e <expr>. Ele é usado para armazenar o tipo atual de uma variável ou expressão (inteiro ou real). No caso de uma variável, o tipo atual é intrínseco. No caso de uma expressão, é determinado a partir dos tipos reais do(s) nó(s) filho(s) do não terminal <expr>.
- *expected_type* – Um atributo herdado associado ao não terminal <expr>. Ele é usado para armazenar o tipo, inteiro ou real, esperado para a expressão, conforme determinado pelo tipo da variável no lado esquerdo da sentença de atribuição.

A gramática de atributos completa é descrita no Exemplo 3.6.

Exemplo 3.6 Uma gramática de atributos para sentenças de atribuição simples.

1. Regra sintática: <assign> → <var> = <expr>
   Regra semântica: <expr>.expected_type ← <var>.actual_type
2. Regra sintática: <expr> → <var>[2] + <var>[3]
   Regra semântica: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicado: <expr>.actual_type == <expr>.expected_type
3. Regra sintática: <expr> → <var>
   Regra semântica: <expr>.actual_type ← <var>.actual_type
   Predicado: <expr>.actual_type == <expr>.expected_type
4. Regra sintática: <var> → A | B | C
   Regra semântica: <var>.actual_type ← look-up (<var>.string)

A função *lookup* busca um dado nome de variável na tabela de símbolos e retorna o tipo dessa variável.

Uma árvore de análise sintática da sentença `A = A + B` gerada pela gramática no Exemplo 3.6 é mostrada na Figura 3.6. Como na gramática, números entre colchetes são adicionados após os rótulos de nós repetidos, de forma que possam ser referenciados de maneira não ambígua.

3.4.6 Computação de valores de atributos

Agora, considere o processo de computar os valores de atributos de uma árvore de análise sintática, o que algumas vezes é chamado de **decorar** a árvore de análise sintática. Se todos os atributos forem herdados, esse processo poderia ser realizado em uma ordem completamente descendente, da raiz para as folhas. Ou, poderia ser realizado em uma ordem completamente ascendente, das folhas para a raiz, se todos os atributos forem sintetizados. Como nossa gramática tem tanto atributos sintetizados quanto herdados, o processo de avaliação não pode ser em uma única direção. A seguir está uma avaliação dos atributos, em uma ordem na qual é possível computá-los:

1. <var>.actual_type ← look-up(A) (Rule 4)
2. <expr>.expected_type ← <var>.actual_type (Rule 1)
3. <var>[2].actual_type ← look-up(A) (Rule 4)
   <var>[3].actual_type ← look-up(B) (Rule 4)
4. <expr>.actual_type ← either int or real (Rule 2)
5. <expr>.expected_type == <expr>.actual_type is either TRUE or FALSE (Rule 2)

A árvore na Figura 3.7 mostra o fluxo dos valores de atributo no exemplo da Figura 3.6. Linhas cheias mostram a árvore de análise sintática; linhas tracejadas mostram o fluxo de atributos na árvore.

A árvore da Figura 3.8 mostra os valores finais dos atributos nos nós. Nesse exemplo, A é definido como um real e B é definido como um inteiro.

Determinar a ordem de avaliação de atributos para o caso geral de uma gramática de atributos é um problema complexo, que necessita da construção de um grafo de dependência para mostrar todas as dependências entre os atributos.

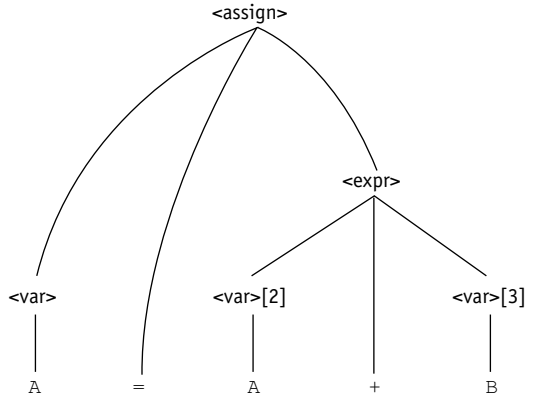


FIGURA 3.6
Uma árvore de análise sintática para A = A + B.

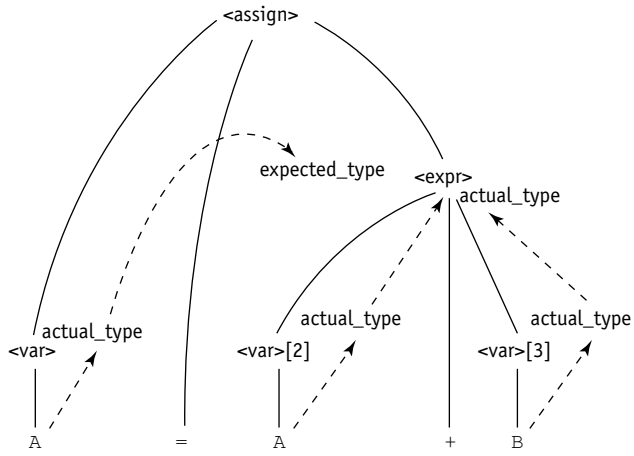


FIGURA 3.7
O fluxo de atributos na árvore.

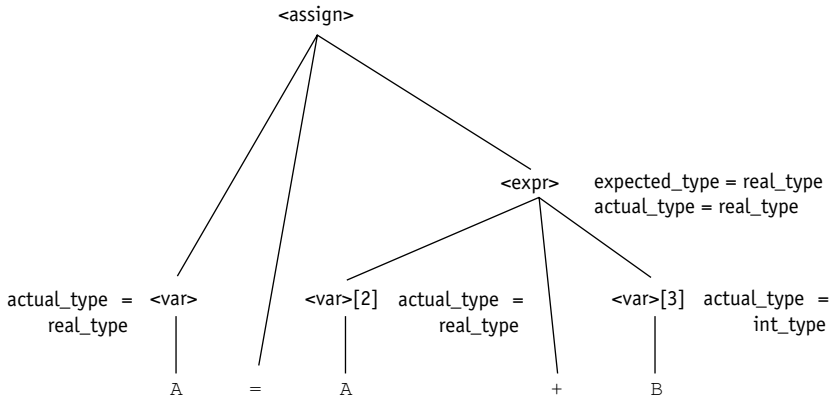


FIGURA 3.8
Uma árvore de análise sintática completamente atribuída.

3.4.7 Avaliação

Verificar as regras de semântica estática de uma linguagem é uma função essencial de todos os compiladores. Mesmo que um desenvolvedor de compiladores nunca tenha ouvido falar de uma gramática de atributos, ele precisará usar suas ideias fundamentais para projetar as verificações de regras de semântica estática para seu compilador.

Uma das principais dificuldades no uso de uma gramática de atributos para descrever toda a sintaxe e a semântica estática de uma linguagem de programação contemporânea real é o tamanho e a complexidade da gramática. O grande número de atributos e regras semânticas necessárias para uma linguagem de programação completa torna tais gramáticas difíceis de escrever e de ler. Além disso, os valores de atributos em uma grande árvore de análise sintática são dispendiosos para serem avaliados. Em contrapartida, gramáticas de atributos menos formais são ferramentas poderosas e comumente usadas

pelos desenvolvedores de compiladores, mais interessados no processo de produzir um compilador do que nos formalismos propriamente ditos.

3.5 DESCRIÇÃO DO SIGNIFICADO DE PROGRAMAS: SEMÂNTICA DINÂMICA

Agora, vamos à difícil tarefa de descrever a **semântica dinâmica**, ou o significado, das expressões, sentenças e unidades de programa de uma linguagem de programação. Devido ao poder e à naturalidade da notação disponível, descrever a sintaxe é algo relativamente simples. Por outro lado, nenhuma notação ou abordagem universalmente aceita foi inventada para semântica dinâmica. Nesta seção, descrevemos brevemente diversos métodos que vêm sendo desenvolvidos. No restante desta seção, quando usamos o termo *semântica*, nos referimos à semântica dinâmica.

Existem muitas razões que levam à necessidade de metodologia e notação para descrever semântica. Os programadores obviamente precisam saber exatamente o que as sentenças de uma linguagem fazem antes de usá-la em seus programas. Desenvolvedores de compiladores devem saber o que as construções da linguagem significam para projetar implementações para elas corretamente. Se existisse uma especificação precisa de semântica de uma linguagem de programação, os programas escritos na linguagem poderiam, potencialmente, ser provados corretos sem a necessidade de testes. Além disso, os compiladores poderiam produzir programas que exibissem o comportamento dado na definição da linguagem; ou seja, sua corretude poderia ser verificada. Uma especificação completa da sintaxe e da semântica da linguagem de programação poderia ser usada por uma ferramenta para gerar automaticamente um compilador para a linguagem. Por fim, os projetistas de linguagens que desenvolvessem as descrições semânticas de suas linguagens, poderiam descobrir ambiguidades e inconsistências em seus projetos durante esse processo.

Os desenvolvedores de software e os projetistas de compiladores normalmente determinam a semântica das linguagens de programação pela leitura de explicações em linguagem natural disponíveis nos manuais da linguagem. Como são explicações normalmente imprecisas e incompletas, essa abordagem é claramente insatisfatória. Em decorrência da falta de especificações semânticas completas de linguagens de programação, os programas raramente são provados corretos sem testes, e os compiladores comerciais nunca são gerados automaticamente a partir de descrições de linguagem.

Scheme, uma linguagem funcional descrita no Capítulo 15, é uma das poucas linguagens de programação cuja definição inclui uma descrição semântica formal. Entretanto, o método usado não é descrito neste capítulo, visto que se concentra em abordagens adequadas para linguagens imperativas.

3.5.1 Semântica operacional

A ideia da **semântica operacional** é descrever o significado de uma sentença ou programa pela especificação dos efeitos de executá-lo em uma máquina. Os efeitos na máquina são vistos como sequências de mudanças em seu estado, em que o estado da máquina é a coleção de valores em seu armazenamento. Uma descrição de semântica operacional óbvia é

dada pela execução de uma versão compilada do programa em um computador. A maioria dos programadores, em alguma ocasião, deve ter escrito um pequeno programa de teste para determinar o significado de alguma construção de linguagem de programação, especialmente enquanto estava aprendendo essa linguagem. Essencialmente, o que o programador faz, nesse caso, é usar semântica operacional para determinar o significado da construção.

Existem diversos problemas no uso dessa abordagem para descrições formais completas de semântica. Primeiro, os passos individuais na execução da linguagem de máquina e as mudanças resultantes no estado da máquina são muito pequenos e numerosos. Segundo, o armazenamento de um computador real é muito grande e complexo. Existem, normalmente, diversos níveis de dispositivos de memória, assim como conexões para incontáveis outros computadores e dispositivos de memória por meio de redes. Dessa forma, linguagens de máquina e computadores reais não são usados para semântica operacional formal. Em vez disso, linguagens de nível intermediário e interpretadores para computadores idealizados são projetados especificamente para o processo.

Existem diferentes níveis de usos para a semântica operacional. No mais alto, o interesse está no resultado final da execução de um programa completo. Isso é algumas vezes chamado de **semântica operacional natural**. No nível mais baixo, a semântica operacional pode ser usada para determinar o significado preciso de um programa, por meio do exame da sequência completa de mudanças de estados que ocorrem quando um programa é executado. Esse uso é algumas vezes chamado de **semântica operacional estrutural**.

3.5.1.1 O processo básico

O primeiro passo para criar uma descrição semântica operacional de uma linguagem é projetar uma linguagem intermediária apropriada, na qual a característica desejada primária é a clareza. Cada construção da linguagem intermediária deve ter um significado óbvio e não ambíguo. Essa linguagem está em um nível intermediário, porque as linguagens de máquina são de nível baixo demais para serem facilmente entendidas, e outra linguagem de alto nível obviamente não é adequada. Se a descrição semântica será usada para semântica operacional natural, uma máquina virtual (um interpretador) deve ser construída para a linguagem intermediária. A máquina virtual pode ser usada para executar cada sentença simples, segmentos de código ou programas completos. A descrição semântica pode ser usada sem uma máquina virtual se o significado de uma única sentença é o suficiente. Nesse caso, que é a semântica operacional estrutural, o código intermediário pode ser inspecionado visualmente.

O processo básico da semântica operacional é usual. O conceito é bastante usado em livros-texto de programação e em manuais de referência de linguagens de programação. Por exemplo, a semântica da construção **`for`** em C pode ser descrita em termos de sentenças mais simples, como em

Sentença C	*Significado*
`for` (expr1; expr2; expr3) {	expr1;
. . .	loop: **`if`** expr2 == 0 **`goto`** out
}	. . .
	expr3;
	`goto` loop
	out: . . .

O leitor humano de tal descrição é o computador virtual e presume-se que ele seja capaz de "executar" as instruções na definição e reconhecer os efeitos da "execução".

Com frequência, a linguagem intermediária e sua máquina virtual associada, usada para as descrições formais de semântica operacional, são altamente abstratas. A linguagem intermediária deve ser conveniente para a máquina virtual, em vez de para os leitores humanos. Para nossos propósitos, no entanto, uma linguagem mais orientada a humanos poderia ser usada. Como exemplo, considere a seguinte lista de sentenças, que seria adequada para descrever a semântica de sentenças de controle simples de uma linguagem de programação típica:

ident = var
ident = ident + 1
ident = ident – 1
goto label
if var relop var **goto** label

Nessas sentenças, relop é um dos operadores relacionais do conjunto {=, <>, >, <, >=, <=}, ident é um identificador e var é um identificador ou uma constante. Essas sentenças são todas simples e fáceis de entender e de implementar.

Uma ligeira generalização dessas três sentenças de atribuição permite descrever expressões aritméticas e sentenças de atribuição mais gerais. As novas sentenças são

ident = var bin_op var
ident = un_op var

onde bin_op é um operador aritmético binário e un_op é um operador unário. Múltiplos tipos de dados aritméticos e conversões de tipo automáticas, é claro, complicam essa generalização. A adição de apenas mais algumas instruções simples permitiria a descrição da semântica de vetores, registros, ponteiros e subprogramas.

No Capítulo 8, a semântica de várias sentenças de controle é descrita por meio dessa linguagem intermediária.

3.5.1.2 Avaliação

O primeiro e mais significativo uso de semântica operacional formal foi para descrever a semântica de PL/I (Wegner, 1972). Tal máquina abstrata em particular e as regras de tradução para PL/I foram chamadas de Vienna Definition Language (VDL), homenageando a cidade na qual a IBM a projetou.

A semântica operacional fornece um meio efetivo de descrever semântica para usuários e implementadores de linguagens, desde que as descrições se mantenham simples e informais. A descrição em VDL da PL/I, infelizmente, é tão complexa que não serve para propósito prático algum.

A semântica operacional depende de linguagens de programação de níveis mais baixos, não de matemática. As sentenças de uma linguagem de programação são descritas em termos de sentenças de uma linguagem de programação de nível mais baixo. Essa abordagem pode levar a circularidades, nas quais os conceitos são indiretamente

definidos em termos deles próprios. Os métodos descritos nas duas seções seguintes são muito mais formais, pois são baseados em matemática e lógica, não em linguagens de programação.

3.5.2 Semântica denotacional

A **semântica denotacional** é o método formal mais rigoroso e mais conhecido para a descrição do significado de programas. Ela é solidamente baseada na teoria de funções recursivas. Uma discussão completa do uso de semântica denotacional para descrever a semântica de linguagens de programação seria longa e complexa. É nosso objetivo fornecer ao leitor uma introdução aos conceitos centrais de semântica denotacional, com alguns exemplos simples que são relevantes para a especificação de linguagens de programação.

O processo de construção de uma especificação de semântica denotacional para uma linguagem de programação requer que alguém defina, para cada entidade da linguagem, tanto um objeto matemático quanto uma função que mapeie as instâncias dessa entidade de linguagem para instâncias do objeto matemático. Como os objetos são definidos rigorosamente, eles modelam o significado exato de suas entidades correspondentes. A ideia é baseada no fato de que existem maneiras rigorosas de manipular objetos matemáticos, mas não construções de linguagens de programação. A dificuldade ao usar esse método está na criação dos objetos e das funções de mapeamento. O método é chamado *denotacional* porque os objetos matemáticos denotam o significado de suas entidades sintáticas correspondentes.

As funções de mapeamento de uma especificação semântica denotacional de uma linguagem de programação, como as funções na matemática, têm um domínio e uma imagem. O domínio é a coleção de valores que são parâmetros legítimos para a função; a imagem é a coleção de objetos para os quais os parâmetros são mapeados. Na semântica denotacional, o domínio é chamado de **domínio sintático**, porque são mapeadas estruturas sintáticas. A imagem é chamada de **domínio semântico**.

A semântica denotacional está relacionada à semântica operacional. Na semântica operacional, as construções de linguagem de programação são traduzidas para construções mais simples, as quais se tornam a base para o significado da construção. Na semântica denotacional, as construções de linguagem de programação são mapeadas para objetos matemáticos – conjuntos ou funções. Entretanto, diferentemente da semântica operacional, a semântica denotacional não modela o processamento computacional passo a passo dos programas.

3.5.2.1 Dois exemplos simples

Para introduzir o método denotacional, usamos uma construção de linguagem muito simples: a representação em cadeias de caracteres de números binários. A sintaxe desses números binários pode ser descrita pelas seguintes regras gramaticais:

```
<bin_num> → '0'
          | '1'
          | <bin_num> '0'
          | <bin_num> '1'
```

Uma árvore de análise sintática para o número binário de exemplo `110` é mostrada na Figura 3.9. Note que colocamos apóstrofos em torno dos dígitos sintáticos para mostrar que eles não são dígitos matemáticos. Isso é similar ao relacionamento entre dígitos codificados em ASCII e dígitos matemáticos. Quando um programa lê um número como uma cadeia, ela deve ser convertida em um valor matemático antes de ser usada como um valor no programa.

O domínio sintático da função de mapeamento para números binários é o conjunto de todas as representações em cadeias de caracteres dos números binários. O domínio semântico é o conjunto dos números decimais não negativos, simbolizado por N.

Para descrever o significado dos números binários usando semântica denotacional, associamos o significado real (um número decimal) a cada regra que tem um único símbolo terminal como seu RHS.

Em nosso exemplo, os números decimais devem estar associados às duas primeiras regras gramaticais. As outras duas são, de certo modo, regras computacionais, porque combinam um símbolo terminal, ao qual um objeto pode ser associado, com um não terminal, o qual pode se esperar que represente alguma construção. Presumindo uma avaliação que progride para cima na árvore de análise sintática, o não terminal do lado direito já teria seu significado anexado. Então, uma regra sintática com um não terminal como RHS precisaria de uma função que computasse o significado do LHS, o qual representa o significado completo do RHS.

A função semântica, chamada M_{bin}, mapeia os objetos sintáticos, conforme descrito nas regras gramaticais anteriores, para os objetos em N, o conjunto de números decimais não negativos. A função M_{bin} é definida como:

M_{bin}('0') = 0
M_{bin}('1') = 1
M_{bin}(<bin_num> '0') = 2 * M_{bin}(<bin_num>)
M_{bin}(<bin_num> '1') = 2 * M_{bin}(<bin_num>) + 1

Os significados, ou objetos denotados (que, nesse caso, são números decimais), podem ser anexados aos nós da árvore de análise sintática mostrada na Figura 3.9, resultando na árvore da Figura 3.10. Essa é uma semântica dirigida por sintaxe. As entidades sintáticas são mapeadas para objetos matemáticos com significado concreto.

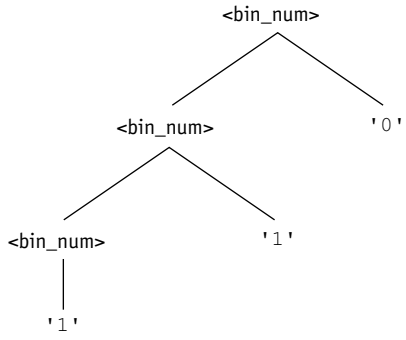


FIGURA 3.9
Uma árvore de análise sintática do número binário `110`.

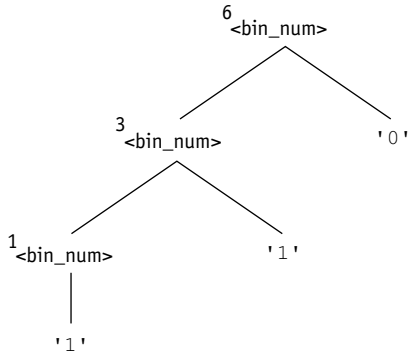


FIGURA 3.10
Uma árvore de análise sintática com objetos denotados para `110`.

Em parte porque precisaremos disso depois, mostraremos um exemplo similar para descrever o significado de literais decimais sintáticos. Neste caso, o domínio sintático é o conjunto de representações de números decimais por meio de cadeias de caracteres. O domínio semântico é, mais uma vez, o conjunto N.

<dec_num> → '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7''8'|'9'
|<dec_num> ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')

Os mapeamentos denotacionais para essas regras sintáticas são

M_{dec}('0') = 0, M_{dec}('1') = 1, M_{dec}('2') = 2, . . ., M_{dec}('9') = 9
M_{dec}(<dec_num> '0') = 10 * M_{dec}(<dec_num>)
M_{dec}(<dec_num> '1') = 10 * M_{dec}(<dec_num>) + 1
. . .
M_{dec}(<dec_num> '9') = 10 * M_{dec}(<dec_num>) + 9

Nas seções seguintes, apresentaremos as descrições em semântica denotacional de algumas construções simples. A premissa de simplificação mais importante feita aqui é que tanto a sintaxe quanto a semântica estática das construções estão corretas. Adiante, presumimos que apenas dois tipos escalares estão incluídos: inteiro e booleano.

3.5.2.2 O estado de um programa

A semântica denotacional de um programa pode ser definida em termos de mudanças de estado em um computador ideal. Semânticas operacionais são definidas assim, e semânticas denotacionais são definidas praticamente da mesma forma. Em uma simplificação adicional, entretanto, a semântica denotacional é definida apenas em termos dos valores de todas as variáveis dos programas. Então, a semântica denotacional usa o estado de um programa para descrever significado, enquanto a semântica operacional usa o estado de uma máquina. A diferença-chave entre as semânticas operacional e denotacional é que mudanças de estado em semântica operacional são definidas por algoritmos codificados, escritos em alguma linguagem de programação, enquanto em semântica denotacional as mudanças de estado são definidas por funções matemáticas.

Seja o estado s de um programa representado como um conjunto de pares ordenados conforme:

$s = \{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\}$

Cada i é o nome de uma variável, e os vs associados são os valores atuais dessas variáveis. Qualquer um desses vs podem ter o valor especial **undef**, indicando que sua variável associada está atualmente indefinida. Considere VARMAP como uma função com dois parâmetros: um nome variável e o estado do programa. O valor de VARMAP (i_j, s) é v_j (o valor correlacionado a i_j no estado s). A maioria das funções de mapeamento semântico para programas e construções de programas mapeia estados para estados. Essas mudanças de estado são usadas para definir o significado dos programas e de suas construções. Algumas construções de linguagem – por exemplo, expressões – são mapeadas para valores, não para estados.

3.5.2.3 Expressões

Expressões são fundamentais para a maioria das linguagens de programação. Supomos aqui que as expressões não têm efeitos colaterais. Além disso, lidamos apenas com expressões muito simples: os únicos operadores são + e *, e uma expressão pode ter, no máximo, um operador; os únicos operandos são variáveis inteiras escalares e literais inteiros; não existem parênteses; e o valor de uma expressão é um inteiro. A seguir, temos a descrição BNF dessas expressões:

<expr> → <dec_num> | <var> | <binary_expr>
<binary_expr> → <left_expr> <operator> <right_expr>
<left_expr> → <dec_num> | <var>
<right_expr> → <dec_num> | <var>
<operator> → + | *

O único erro que consideramos em expressões é uma variável contendo um valor não definido. Obviamente, outros erros podem ocorrer, mas a maioria é dependente de máquina. Considere Z como o conjunto de inteiros e **error** como o valor do erro. Então $Z^{\cup}$ {**error**} é o domínio semântico para a especificação denotacional de nossas expressões.

A função de mapeamento para uma dada expressão E e o estado s é a seguinte. Para distinguir entre definições de funções matemáticas e sentenças de linguagens de programação, usamos o símbolo $\Delta =$ para definir funções matemáticas. O símbolo de implicação, =>, usado nessa definição conecta a forma de um operando com sua construção case (ou switch) associada. Uma notação com pontos é usada para nos referenciarmos aos nós filhos de um nó. Por exemplo, <binary_expr>.<left_expr> se refere ao nó filho da esquerda de <binary_expr>.

M_e(<expr>, s) $\Delta =$ case <expr> of
    <dec_num>=> M_{dec}(<dec_num>, s)
    <var> =>if VARMAP(<var>, s) == **undef**
        then **error**
        else VARMAP(<var>, s)
    <binary_expr> =>

```
if(Me(<binary_expr>.<left_expr>,s) == undef OR
   Me(<binary_expr>.<right_expr>, s) == undef)
then error
else if (<binary_expr>.<operator> == '+')
        then Me(<binary_expr>.<left_expr>, s) +
              Me(<binary_expr>.<right_expr>, s)
        else Me(<binary_expr>.<left_expr>, s) *
              Me(<binary_expr>.<right_expr>, s)
```

3.5.2.4 Sentenças de atribuição

Uma sentença de atribuição é uma avaliação de expressão mais a definição da variável-alvo para o valor da expressão. Nesse caso, a função significado mapeia de um estado para outro. Essa função pode ser descrita como:

M_a (x = E, s) Δ= if M_e (E, s) == **error**
then **error**
else $s' = \{< i_1, v_1' >, < i_2, v_2' >, \ldots, < i_n, v_n' >\}$, where
for j = 1, 2, . . . , n
if ij == x
then $v_j' = M_e$ (E, s)
else $v_j' = VARMAP(i_j, s)$

Note que a comparação na antepenúltima linha acima, i^j == x, é em relação aos nomes, não aos valores.

3.5.2.5 Laços lógicos com pré-teste

A semântica denotacional de um laço lógico com pré-teste é enganosamente simples. Para agilizar a discussão, supomos que existem outras duas funções de mapeamento, M_{sl} e M_b, que mapeiam listas de sentenças e estados para estados, e expressões booleanas para valores booleanos (ou **error**), respectivamente. A função é

M_l (**`while`** B **`do`** L, s) Δ= if M_b (B, s) == **undef**
then **error**
else if M_b (B, s) == false
then s
else if M_{sl} (L, s) == **error**
then **error**
else M_l (**`while`** B **`do`** L, M_{sl} (L, s))

O significado do laço é simplesmente o valor das variáveis do programa após as sentenças no laço terem sido executadas o número de vezes necessário, supondo que não ocorreram erros. Essencialmente, o laço foi convertido de iteração em recursão, e o controle da recursão é definido matematicamente por outras funções recursivas de mapeamento de estado. É mais fácil descrever a recursão com rigor matemático do que a iteração.

Uma observação importante neste momento é que essa definição, como laços de programas reais, pode não computar coisa alguma por causa da não finalização.

3.5.2.6 Avaliação

Objetos e funções, como aqueles usados nas construções anteriores, podem ser definidos para as outras entidades sintáticas de linguagens de programação. Quando um sistema completo for definido para uma linguagem, ele pode ser usado para determinar o significado de programas completos nela. Isso fornece um contexto para se pensar sobre programação de uma maneira altamente rigorosa.

Conforme mencionado, a semântica denotacional pode auxiliar o projeto de linguagens. Por exemplo, sentenças para as quais a descrição em semântica denotacional é complexa e difícil podem indicar ao projetista que tais sentenças também serão difíceis para que os usuários as entendam e que talvez seja necessário um projeto alternativo.

Devido à complexidade das descrições denotacionais, elas são de pouca utilidade para os usuários das linguagens. No entanto, fornecem uma maneira excelente de descrever uma linguagem de maneira concisa.

Apesar do uso de semântica denotacional ser normalmente atribuído a Scott e Strachey (1971), a abordagem denotacional geral para descrição de linguagens pode remontar ao século XIX (Frege, 1892).

» nota histórica

Foi realizada uma quantidade significativa de trabalhos sobre a possibilidade de usar descrições de linguagens denotacionais para gerar compiladores automaticamente (Jones, 1980; Milos et al., 1984; Bodwin et al., 1982). Esses esforços mostraram que o método é viável, mas o trabalho nunca progrediu a ponto de poder ser usado para gerar compiladores úteis.

3.5.3 Semântica axiomática

A **semântica axiomática**, chamada assim porque é baseada em lógica matemática, é a abordagem mais abstrata para a especificação de semântica discutida neste capítulo. Em vez de especificar diretamente o significado de um programa, a semântica axiomática especifica o que pode ser provado sobre o programa. Lembre-se de que um dos usos possíveis da especificação semântica é a prova de corretude de programas.

Na semântica axiomática, não existe um modelo do estado de uma máquina ou programa, nem um modelo das mudanças de estado que ocorrem quando o programa é executado. O significado de um programa é baseado nas relações entre suas variáveis e constantes, que são as mesmas para cada execução do programa.

A semântica axiomática tem duas aplicações distintas: a verificação de programas e a especificação de semântica de programas. Esta seção se concentra na verificação de programas e na descrição de semântica axiomática.

A semântica axiomática foi definida em conjunto com o desenvolvimento de uma estratégia para provar a corretude de programas. Tais provas de corretude, quando podem ser construídas, mostram que um programa realiza a computação descrita em sua especificação. Em uma prova, cada sentença de um programa é precedida e seguida de uma expressão lógica que especifica restrições em variáveis dos programas. Tais restrições, em vez de o estado completo de uma máquina abstrata (como na semântica operacional), são usadas para especificar o significado da sentença. A notação usada

para descrever restrições – que são, na verdade, a linguagem da semântica axiomática – é o cálculo de predicados. Apesar de expressões booleanas simples serem normalmente adequadas para expressar restrições, em alguns casos elas não são.

Quando a semântica axiomática é usada para especificar formalmente o significado de uma sentença, ele é definido pelo efeito da sentença em asserções sobre os dados afetados pela sentença.

3.5.3.1 Asserções

As expressões lógicas usadas na semântica axiomática são chamadas de predicados ou **asserções**. Uma asserção que vem imediatamente antes de uma sentença de programa descreve as restrições nas variáveis do programa naquele ponto. Uma asserção imediatamente após uma sentença descreve as novas restrições sobre as variáveis (e possivelmente outras) após a execução da sentença. Essas asserções são denominadas, respectivamente, **pré-condição** e **pós-condição** da sentença. Para duas sentenças adjacentes, a pós-condição da primeira serve como uma pré-condição para a segunda. Desenvolver uma descrição axiomática ou prova de um programa requer que toda sentença do programa tenha tanto uma pré-condição quanto uma pós-condição.

Nas próximas seções, examinaremos as asserções do ponto de vista de que pré-condições para sentenças são computadas para pós-condições dadas, apesar de ser possível considerá-las no sentido oposto. Supomos que todas as variáveis são do tipo inteiro. Como um exemplo simples, considere a sentença de atribuição e a pós-condição a seguir:

```
sum = 2 * x + 1 {sum > 1}
```

As asserções de pré-condição e pós-condição são apresentadas em chaves para distingui-las das demais partes das sentenças dos programas. Uma possível pré-condição para essa sentença é `{x > 10}`.

Em semântica axiomática, o significado de uma sentença específica é definido por sua pré-condição e sua pós-condição. Na prática, as duas asserções especificam precisamente o efeito de executar uma sentença.

Nas subseções a seguir, enfocamos as provas de corretude de sentenças e de programas, que são um uso da semântica axiomática. O objetivo mais geral de semântica axiomática é expressar precisamente o significado de sentenças e programas em termos de expressões lógicas. A verificação de programas é uma aplicação das descrições axiomáticas de linguagens.

3.5.3.2 Pré-condições mais fracas

A **pré-condição mais fraca** é a menos restritiva que garantirá a validade da pós-condição associada. Por exemplo, na sentença e na pós-condição dadas na Seção 3.5.3.1, `{x > 10}`, `{x > 50}` e `{x >1000}` são todas pré-condições válidas. A mais fraca nesse caso é `{x > 0}`.

Se a pré-condição mais fraca pode ser computada a partir da pós-condição mais geral para cada um dos tipos de sentenças de uma linguagem, então os processos usados para computar essas pré-condições fornecem uma descrição concisa da semântica des-

sa linguagem. Além disso, provas de corretude podem ser construídas para programas nessa linguagem. Uma prova de programa começa com o uso das características dos resultados da execução dos programas como a pós-condição da última sentença. Essa pós-condição é usada para computar a pré-condição mais fraca para a última sentença. Essa pré-condição é, então, usada como pós-condição para a penúltima sentença. Esse processo continua até o início do programa ser alcançado. Nesse ponto, a pré-condição da primeira sentença descreve as condições nas quais o programa computará os resultados desejados. Se essas condições forem derivadas da especificação de entrada do programa, foi verificado que o programa é correto.

A **regra de inferência** é um método de inferir a verdade de uma asserção com base nos valores de outras asserções. A forma geral de uma regra de inferência é a seguinte:

$$\frac{S1,\ S2,\ \ldots,\ Sn}{S}$$

Essa regra diz que, se S1, S2, . . ., e S*n* são verdadeiros, então pode-se deduzir que S é verdadeiro. A parte de cima de uma regra de inferência é chamada de **antecedente**; a parte de baixo é chamada de **consequente**.

Um **axioma** é uma sentença lógica que se assume verdadeira. Logo, é uma regra de inferência sem um antecedente.

Para algumas sentenças de programa, a computação de uma pré-condição mais fraca para a sentença e uma pós-condição é simples e pode ser especificada por um axioma. Na maioria dos casos, entretanto, a pré-condição mais fraca pode ser especificada apenas por uma regra de inferência.

Para usar semântica axiomática com uma linguagem de programação, seja para prova de corretude ou para especificações semânticas formais, deve existir um axioma ou uma regra de inferência para cada tipo de sentença na linguagem.

Nas próximas subseções, apresentaremos um axioma para sentenças de atribuição e regras de inferência para sequências de sentenças, sentenças de seleção e sentenças de laços de repetição com pré-teste lógico. Note que supomos que nem as expressões aritméticas nem as booleanas têm efeitos colaterais.

3.5.3.3 Sentenças de atribuição

A pré-condição e a pós-condição de uma sentença de atribuição definem o seu significado. Para se definir o significado de uma sentença de atribuição, deve existir uma maneira de calcular sua pré-condição a partir de sua pós-condição.

Considere x = E uma sentença geral de atribuição e Q sua pós-condição. Então, sua pré-condição mais fraca, P, é definida pelo axioma

$$P = Q_{x \to E}$$

ou seja, P é computada como Q, com todas as instâncias de x substituídas por E. Por exemplo, se tivéssemos a sentença de atribuição e pós-condição

```
a = b / 2 - 1 {a < 10}
```

a pré-condição mais fraca seria computada pela substituição de `b / 2 - 1` por `a` na pós-condição {`a < 10`}, como:

```
b / 2 - 1 < 10
b < 22
```

Logo, a pré-condição mais fraca para a sentença de atribuição dada e sua pós-condição é {b < 22}. Lembre-se de que o axioma de atribuição é seguramente correto apenas na ausência de efeitos colaterais. Uma sentença de atribuição tem um efeito colateral se ela modifica alguma variável que não seja seu alvo.

A notação usual para especificar a semântica axiomática de uma forma sentencial é

{P} S {Q}

onde P é a pré-condição, Q é a pós-condição e S é a forma sentencial. No caso da sentença de atribuição, a notação é

$\{Q_{x \to E}\}\ x = E\{Q\}$

Como outro exemplo da computação de uma pré-condição para uma sentença de atribuição, considere:

```
x = 2 * y - 3 {x > 25}
```

A pré-condição é computada como:

```
2 * y - 3 > 25
y > 14
```

Logo, {y > 14} é a pré-condição mais fraca para essa sentença de atribuição e pós-condição.

Note que a aparição do lado esquerdo da sentença de atribuição em seu lado direito não afeta o processo de computar a pré-condição mais fraca. Por exemplo, para

```
x = x + y - 3 {x > 10}
```

a pré-condição mais fraca é

```
x + y - 3 > 10
y > 13 - x
```

Lembre-se de que a semântica axiomática foi desenvolvida para provar a corretude de programas. Assim, é natural neste ponto imaginarmos como o axioma para sentenças de atribuição pode ser usado para provar alguma coisa. Aqui está como: uma sentença de atribuição contendo tanto uma pré-condição quanto uma pós-condição pode ser considerada uma sentença lógica, ou teorema. Se o axioma de atribuição, quando aplicado à pós-condição e à sentença de atribuição, produz a pré-condição dada, o teorema está provado. Por exemplo, considere a seguinte sentença lógica:

```
{x > 3} x = x - 3 {x > 0}
```

Usar o axioma de atribuição na sentença e sua pós-condição produz {x > 3}, que é a pré-condição dada. Logo, provamos a sentença lógica de exemplo.

A seguir, considere a próxima sentença lógica

```
{x > 5} x = x - 3 {x > 0}
```

Nesse caso, a pré-condição dada, `{x > 5}`, não é a mesma que a asserção produzida pelo axioma. Entretanto, é óbvio que `{x > 5}` implica em `{x > 3}`. Para usar isso em uma prova, é necessária uma regra de inferência, chamada de **regra de consequência**. A forma da regra de consequência é

$$\frac{\{P\}\ S\ \{Q\},\ P' \Rightarrow P,\ Q \Rightarrow Q'}{\{P'\}S\{Q'\}}$$

O símbolo => significa "implica", e S pode ser qualquer sentença de um programa. A regra pode ser enunciada como segue: se a sentença lógica $\{P\}S\{Q\}$ é verdadeira, a asserção P′ implica na asserção P e a asserção Q implica na asserção Q′, então pode-se deduzir que $\{P'\}S\{Q'\}$. Em outras palavras, a regra de consequência diz que uma pós-condição pode sempre ser enfraquecida e que uma pré-condição pode sempre ser reforçada. Isso é bastante útil na prova de programas. Por exemplo, permite completar a prova da última sentença lógica de exemplo acima. Se considerarmos P como `{x > 3}`, Q e Q′ como `{x > 0}`, e P′ como `{x > 5}`, teríamos

```
{x > 3}x=x- 3{x > 0},(x > 5) => {x > 3},(x > 0) => (x > 0)
-----------------------------------------------------------
                  {x > 5}x = x - 3{x > 0}
```

O primeiro termo do antecedente (`{x > 3} x = x - 3 {x > 0}`) foi provado com o axioma de atribuição. O segundo e o terceiro termos são óbvios. Logo, pela regra da consequência, o consequente é verdadeiro.

3.5.3.4 Sequências

A pré-condição mais fraca para uma sequência de sentenças não pode ser descrita por um axioma, porque ela depende dos tipos de sentenças particulares na sequência. Nesse caso, a pré-condição só pode ser descrita com uma regra de inferência. Considere S1 e S2 sentenças adjacentes de um programa. Se S1 e S2 têm as seguintes pré-condição e pós-condição

{P1} S1 {P2}
{P2} S2 {P3}

a regra de inferência para tal sequência de duas sentenças é

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1,\ S2\ \{P3\}}$$

Então, para nosso exemplo, {P1} S1; S2 {P3} descreve a semântica axiomática da sequência S1; S2. A regra de inferência diz que, para se obter a pré-condição da sequência, a pré-condição da segunda sentença deve ser computada. Essa nova asserção é usada como a pós-condição da primeira sentença, a qual pode ser usada para computar a pré-condição da primeira – a qual também é a pré-condição de toda a sequência. Se S1 e S2 são as sentenças de atribuição

x1 = E1

e

x2 = E2

então temos

$\{P3_{x2 \rightarrow E2}\}\ x2 = E2\ \{P3\}$
$\{(P3_{x2 \rightarrow E2})_{x1 \rightarrow E1}\}\ x1 = E1\ \{P3_{x2 \rightarrow E2}\}$

Portanto, a pré-condição mais fraca da sequência x1 = E1; x2 = E2 com pós-condição P3 é $\{(P3_{x2 \rightarrow E2})_{x1 \rightarrow E1}\}$.

Por exemplo, considere a sequência e a pós-condição a seguir:

```
y = 3 * x + 1;
x = y + 3;
{x < 10}
```

A pré-condição para a segunda sentença de atribuição é

```
y < 7
```

que é usada como pós-condição para a primeira sentença. Agora, a pré-condição da primeira sentença pode ser computada:

```
3 * x + 1 < 7
x < 2
```

Logo, `{x < 2}` é a pré-condição tanto da primeira sentença quanto da sequência de duas sentenças.

3.5.3.5 Seleção

A seguir, consideramos a regra de inferência para sentenças de seleção, cuja forma geral é

if B **then** S1 **else** S2

Consideramos apenas seleções que incluem cláusulas **else**. A regra de inferência é

$$\frac{\{B \text{ and } P\}\ S1\ \{Q\},\ \{(\text{not } B) \text{ and } P\}\ S2\{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q\}}$$

Essa regra indica que as sentenças de seleção devem ser provadas tanto se a expressão booleana de controle for verdadeira quanto se for falsa. A primeira sentença lógica acima da linha representa a cláusula **then**; a segunda representa a cláusula **else**. De acordo com a regra de inferência, precisamos de uma pré-condição P que possa ser usada tanto na pré-condição da cláusula **then** quanto na da cláusula **else**.

Considere o seguinte exemplo da computação da pré-condição usando a regra de inferência para seleção. A sentença de seleção de exemplo é

```
if x > 0 then
 y = y - 1
else
 y = y + 1
```

Suponha que a pós-condição, Q, para essa sentença de seleção seja `{y > 0}`. Podemos usar o axioma para atribuição na cláusula **`then`**

```
y = y - 1 {y > 0}
```

Isso produz `{y - 1 > 0}` ou `{y > 1}`. E isso pode ser usado como a parte P da pré-condição para a cláusula **`then`**. Agora, aplicamos o mesmo axioma para a cláusula **`else`**

```
y = y + 1 {y > 0}
```

o que produz a pré-condição `{y + 1 > 0}` ou `{y > -1}`. Como `{y > 1} => {y > -1}`, a regra de consequência permite usarmos `{y > 1}` para a pré-condição de toda a sentença de seleção.

3.5.3.6 Laços lógicos com pré-teste

Outra construção essencial das linguagens de programação imperativa é o pré-teste lógico, ou laço **`while`**. Computar a pré-condição mais fraca para um laço **`while`** é inerentemente mais difícil do que para uma sequência, porque o número de iterações não pode sempre ser predeterminado. Em um caso em que o número de iterações é conhecido, o laço de repetição pode ser expandido e tratado como uma sequência.

O problema de computar a pré-condição mais fraca para laços de repetição é similar ao problema de provar um teorema acerca de todos os números inteiros positivos. No último caso, a indução normalmente é usada, e o mesmo método indutivo pode ser usado para alguns laços. O passo principal na indução é encontrar uma hipótese indutiva. O passo correspondente na semântica axiomática de um laço **`while`** é encontrar uma asserção chamada de **invariante de laço**, crucial para encontrar a pré-condição mais fraca.

A regra de inferência para computar a pré-condição para um laço **`while`** é a seguinte:

$$\frac{\{I \text{ and } B\}\ S\ \{I\}}{\{I\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{end}\ \{I \text{ and (not } B)\}}$$

Nessa regra, I é a invariante de laço de repetição. Isso parece simples, mas não é. A complexidade está em encontrar uma invariante de laço de repetição apropriada.

A descrição axiomática de um laço **`while`** é escrita como

{P} **`while`** B **`do`** S **`end`** {Q}

A invariante de laço de repetição deve satisfazer alguns requisitos para ser útil. Primeiro, a pré-condição mais fraca para o laço **`while`** deve garantir a verdade da invariante de laço. Por sua vez, a invariante de laço deve garantir a verdade da pós-condição no término do laço. Essas restrições nos levam da regra de inferência para a descrição axiomática. Durante a execução do laço, a verdade da invariante de laço não deve ser afetada pela avaliação da expressão booleana de controle do laço nem pelas sentenças do corpo do laço. Daí o nome *invariante*.

Outro fator complicador para os laços **while** é a questão do término do laço. Um laço que não termina não pode ser correto, e de fato não computa nada. Se Q é a pós-condição que deve ser satisfeita imediatamente após o término do laço, então uma pré-condição P para o laço é uma condição que garante Q no término do laço e também que o laço termine.

A descrição axiomática completa de uma construção **while** requer que todas as condições abaixo sejam verdadeiras, nas quais I é a invariante do laço:

P => I
{I and B} S {I}
(I and (not B)) => Q
o laço termina

Se um laço de repetição computa uma sequência de valores numéricos, é possível encontrar uma invariante de laço com uma abordagem usada para determinar a hipótese indutiva quando a indução matemática é utilizada para provar uma sentença a respeito de uma sequência matemática. O relacionamento entre o número de iterações e a pré-condição para o corpo do laço de repetição é computado para alguns casos, a fim de que um padrão apareça e possa ser aplicável para o caso geral. Ajuda tratar o processo de produzir uma pré-condição mais fraca como uma função, wp. Em geral

wp(sentença, pós-condição) = pré-condição

Uma função wp é geralmente chamada de **transformador de predicado**, porque recebe um predicado, ou asserção, como parâmetro e retorna outro predicado.

Para encontrar I, a pós-condição Q do laço de repetição é usada para computar as pré-condições para diversos números de iterações do corpo do laço, começando com zero. Se o corpo do laço contiver uma única sentença de atribuição, o axioma para sentenças de atribuição pode ser usado para computar esses casos. Considere o laço de exemplo:

```
while y <> x do y = y + 1 end {y = x}
```

Lembre-se de que o sinal de igualdade está sendo usado para dois propósitos aqui. Em asserções, ele representa igualdade matemática; fora das asserções, significa o operador de atribuição.

Para nenhuma iteração, a pré-condição mais fraca é, obviamente,

```
{y = x}
```

Para uma iteração, ela é

wp`(y = y + 1, {y = x}) = {y + 1 = x}`, or `{y = x - 1}`

Para duas iterações, é

wp`(y = y + 1, {y = x - 1})={y + 1 = x - 1}`, or `{y = x - 2}`

Para três iterações, é

wp(y = y + 1, {y = x - 2})={y + 1 = x - 2}, or {y = x - 3}

Agora está claro que {y < x} será suficiente para os casos de uma ou mais iterações. Combinando isso com {y = x} para o caso de nenhuma iteração, temos {y <= x}, que pode ser usada como a invariante do laço de repetição. Uma pré-condição para a sentença **while** pode ser determinada a partir da invariante do laço de repetição. Na verdade, I pode ser usado como a pré-condição P.

Devemos garantir que nossa escolha satisfaz os quatro critérios para I em nosso laço de repetição de exemplo. Primeiro, porque P = I, P => I. O segundo requisito é que o seguinte deve ser verdade:

{I and B} S {I}

Em nosso exemplo, temos

```
{y <= x and y <> x} y = y + 1 {y <= x}
```

Aplicando o axioma de atribuição a

```
y = y + 1 {y <= x}
```

obtemos {y + 1 <= x}, que é equivalente a {y < x}, o que é implicado por {y <= x and y <> x}. Logo, a sentença anterior está provada.

A seguir, devemos ter

{I and (not B)} => Q

Em nosso exemplo, temos

```
{(y <= x) and not (y <> x)} => {y = x}
{(y <= x) and (y = x)} => {y = x}
{y = x} => {y = x}
```

Logo, isso é obviamente verdade. A seguir, deve ser considerado o término do laço. Nesse exemplo, a questão é se o laço

```
{y <= x} while y <> x do y = y + 1 end {y = x}
```

termina. Lembrando que se presume que x e y são variáveis inteiras, é fácil de ver que esse laço termina. A pré-condição garante que, inicialmente, y não é maior que x. O corpo do laço incrementa y a cada iteração, até y ser igual a x. Não importa o quão menor y era em relação a x inicialmente, no final ele se tornará igual a x. Logo, o laço terminará. Como nosso I escolhido satisfaz os quatro critérios, é uma invariante e uma pré-condição do laço.

O processo anterior usado para computar a invariante para um laço nem sempre produz uma asserção que é a pré-condição mais fraca (apesar de ser no exemplo).

Como outro exemplo de encontrar uma invariante de laço com a abordagem usada na indução matemática, considere a seguinte sentença de laço:

```
while s > 1 do s = s / 2 end {s = 1}
```

Como antes, usamos o axioma de atribuição para tentar encontrar uma invariante e uma pré-condição para o laço. Para nenhuma iteração, a pré-condição mais fraca é `{s = 1}`.

Para uma iteração, ela é

wp`(s = s / 2, {s = 1}) = {s / 2 = 1}`, or `{s = 2}`

Para duas iterações, é

wp`(s = s / 2, {s = 2}) = {s / 2 = 2}`, or `{s = 4}`

Para três iterações, é

wp`(s = s / 2, {s = 4}) = {s / 2 = 4}`, or `{s = 8}`

A partir desses casos, podemos ver claramente que a invariante é

{`s` é uma potência não negativa de `2`}

Mais uma vez, o I computado pode servir como P, e I passa nos quatro requisitos. Diferentemente de nosso exemplo anterior da busca por uma pré-condição do laço de repetição, essa não é uma pré-condição mais fraca. Considere usar a pré-condição `{s > 1}`. A sentença lógica

```
{s > 1} while s > 1 do s = s / 2 end {s = 1}
```

pode ser facilmente provada, e essa pré-condição é significativamente mais ampla do que a computada anteriormente. O laço e a pré-condição são satisfeitos para qualquer valor positivo de s, não apenas para potências de 2, como o processo indica. Devido à regra de consequência, usar uma pré-condição mais forte que a pré-condição mais fraca não invalida uma prova.

Encontrar invariantes de laços nem sempre é fácil. É benéfico entender a natureza dessas invariantes. Primeiro, uma invariante de laço é uma versão mais fraca da pós-condição do laço e também uma pré-condição para o laço. Então, I deve ser fraca o suficiente para ser satisfeita antes do início da execução do laço, mas, quando combinada com a condição de saída do laço, deve ser forte o suficiente para forçar a verdade da pós-condição.

Devido à dificuldade de provar o término de laços, esse requisito é geralmente ignorado. Se o término de um laço puder ser mostrado, a descrição axiomática do laço é chamada de **corretude total**. Se as outras condições puderem ser satisfeitas, mas o término não é garantido, ela é chamada de **corretude parcial**.

Em laços mais complexos, encontrar uma invariante de laço adequada, mesmo para corretude parcial, requer uma boa dose de criatividade. Como computar a pré-condição

para um laço **while** depende da descoberta de uma invariante de laço, pode ser difícil provar a corretude de programas com laços **while** usando semântica axiomática.

3.5.3.7 Provas de programas

Esta seção fornece validações para dois programas simples. O primeiro exemplo de prova de corretude é para um programa muito pequeno, formado por uma sequência de três sentenças de atribuição que trocam os valores de duas variáveis.

```
{x = A AND y = B}
t = x;
x = y;
y = t;
{x = B AND y = A}
```

Como o programa é formado inteiramente por sentenças de atribuição em uma sequência, o axioma de atribuição e a regra de inferência para sequências podem ser usados para provar sua corretude. O primeiro passo é usar o axioma de atribuição na última sentença e a pós-condição para o programa completo. Isso leva à pré-condição

```
{x = B AND t = A}
```

A seguir, usamos essa nova pré-condição como a pós-condição na sentença do meio e computamos sua pré-condição,

```
{y = B AND t = A}
```

A seguir, usamos essa nova asserção como a pós-condição na primeira sentença e aplicamos o axioma de atribuição, o que leva a

```
{y = B AND x = A}
```

que é a mesma pré-condição no programa, exceto pela ordem dos operandos no operador AND. Como AND é um operador simétrico, nossa prova está completa.

O seguinte exemplo é uma prova de corretude para um programa em pseudocódigo que computa a função fatorial.

```
{n >= 0}
count = n;
fact = 1;
while count <> 0 do
    fact = fact * count;
    count = count - 1;
end
{fact = n!}
```

O método descrito anteriormente para encontrar a invariante do laço não funciona para o laço desse exemplo. Alguma outra alternativa é necessária aqui, e ela pode ser encontrada por meio de um breve estudo do código. O laço computa primeiro a função

fatorial na ordem da última multiplicação; ou seja, `(n - 1) * n` é efetuada primeiro, supondo que `n` é maior que `1`. Então, parte da invariante pode ser a seguinte:

```
fact = (count + 1) * (count + 2) * . . . * (n - 1) * n
```

Mas também precisamos garantir que `count` seja sempre não negativo, o que podemos fazer adicionando essa informação à asserção acima, para obter o seguinte:

```
I = (fact = (count + 1) * . . . * n) AND (count >= 0)
```

A seguir, precisamos confirmar que esse I satisfaz os requisitos para invariantes. Mais uma vez, deixamos I ser usada para P, de forma que P implica I. A próxima questão é

{I and B} S {I}

I e B são os seguintes:

```
((fact = (count + 1) * . . . * n) AND (count >= 0)) AND
   (count <> 0)
```

Isso se reduz a

```
(fact = (count + 1) * . . . * n) AND (count > 0)
```

No nosso caso, precisamos computar a pré-condição do corpo do laço usando a invariante para a pós-condição. Para

```
{P} count = count - 1 {I}
```

computamos P como

```
{(fact = count * (count + 1) * . . . * n) AND
   (count >= 1)}
```

Usando isso como a pós-condição para a primeira atribuição no corpo do laço,

```
{P} fact = fact * count {(fact = count * (count + 1)
                  * . . . * n) AND (count >= 1)}
```

Nesse caso, P é

```
{(fact = (count + 1) * . . . * n) AND (count >= 1)}
```

Está claro que I e B implicam esse P. Então, pela regra de consequência,

```
{I AND B} S {I}
```

é verdadeira. Por fim, o último teste de I é

```
I AND (NOT B) => Q
```

Para nosso exemplo, isso é

```
((fact = (count + 1) * . . . * n) AND (count >= 0)) AND
(count = 0)) => fact = n!
```

Isso é claramente verdade, visto que, quando `count = 0`, a primeira parte é a definição de fatorial. Então, nossa escolha de I atende aos requisitos para uma invariante de laço. Agora, podemos usar nosso P (o mesmo que I) a partir do **`while`** como a pós-condição na segunda atribuição do programa

```
{P} fact = 1 {(fact = (count + 1) * . . . * n) AND
  (count >= 0)}
```

que leva ao P

```
(1 = (count + 1) * . . . * n) AND (count >= 0))
```

Usando isso como a pós-condição para a primeira atribuição no código

```
{P} count = n {(1 = (count + 1) * . . . * n) AND
    (count >= 0)) }
```

o que produz para P

```
{(n + 1) * . . . * n = 1) AND (n >= 0)}
```

O operando esquerdo do operador `AND` é verdadeiro (porque `1 = 1`) e o operando direito é exatamente a pré-condição do segmento de código completo, `{n >= 0}`. Logo, provamos que programa está correto.

3.5.3.8 Avaliação

Como já mencionado, para definir a semântica de uma linguagem de programação completa por meio do método axiomático, deve existir um axioma ou uma regra de inferência para cada tipo de sentença na linguagem. Definir axiomas ou regras de inferência para algumas das sentenças de linguagens de programação se mostrou uma tarefa difícil. Uma solução óbvia para esse problema é projetar a linguagem com o método axiomático em mente, de forma a incluir apenas sentenças para as quais axiomas ou regras de inferência podem ser escritos. Infelizmente, uma linguagem assim necessariamente omitiria algumas partes úteis e expressivas.

A semântica axiomática é uma ferramenta poderosa para a pesquisa na área de prova de corretude de programas, e ela fornece um esquema excelente no qual se pode pensar acerca de programas, tanto durante sua construção quanto posteriormente. Contudo, sua utilidade em descrever o significado de linguagens de programação para os usuários de linguagens e para os desenvolvedores de compiladores é altamente limitada.

RESUMO

A Forma de Backus-Naur e as gramáticas livres de contexto são metalinguagens equivalentes, bastante adequadas para a tarefa de descrever a sintaxe de linguagens de programação. Não são apenas ferramentas descritivas concisas; as árvores de análise sintática que podem ser associadas às suas ações de geração fornecem uma evidência gráfica das estruturas sintáticas subjacentes. Além disso, elas são naturalmente relacionadas aos dispositivos de reconhecimento para as linguagens que geram, o que leva à construção relativamente fácil de analisadores sintáticos para compiladores para essas linguagens.

Uma gramática de atributos é um formalismo descritivo que pode descrever tanto a sintaxe quanto a semântica estática de uma linguagem. Gramáticas de atributos são extensões de gramáticas livres de contexto. Uma gramática de atributos consiste em uma gramática, um conjunto de atributos, um conjunto de funções de computação de atributos e um conjunto de predicados que descrevem as regras de semântica estática.

Este capítulo forneceu uma breve introdução aos três métodos de descrição semântica: operacional, denotacional e axiomático. A semântica operacional é um método para a descrição do significado das construções de uma linguagem em termos de seus efeitos em uma máquina ideal. Na semântica denotacional, objetos matemáticos são usados para representar o significado das construções da linguagem. Entidades de linguagem são convertidas nesses objetos matemáticos por meio de funções recursivas. A semântica axiomática, baseada em lógica formal, foi criada como uma ferramenta para provar a corretude de programas.

NOTAS BIBLIOGRÁFICAS

A descrição de sintaxe feita por meio de gramáticas livres de contexto e BNF é muito discutida em Cleaveland e Uzgalis (1976).

A pesquisa em semântica axiomática começou com Floy (1967) e foi complementada por Hoare (1969). A semântica de grande parte de Pascal foi descrita por Hoare e Wirth (1973) por meio desse método. As partes que eles não concluíram envolviam efeitos colaterais de funções e sentenças goto. Esses foram os aspectos que se mostraram mais difíceis de descrever.

A técnica de usar pré-condições e pós-condições durante o desenvolvimento de programas é descrita (e sugerida) por Dijkstra (1976) e discutida em detalhes em Gries (1981).

Boas introduções à semântica denotacional podem ser encontradas em Gordon (1979) e Stoy (1977). Introduções a todos os métodos de descrição semântica discutidos neste capítulo podem ser encontradas em Marcotty et al. (1976). Outra boa referência para grande parte do material do capítulo é Pagan (1981). A forma das funções de semântica denotacional deste capítulo é similar à encontrada em Meyer (1990).

QUESTÕES DE REVISÃO

1. Defina *sintaxe* e *semântica*.
2. Para quem as descrições de linguagens são criadas?
3. Descreva a operação de um gerador de linguagens típico.
4. Descreva a operação de um reconhecedor de linguagens típico.
5. Qual é a diferença entre uma sentença e uma forma sentencial?
6. Defina uma regra gramatical recursiva à esquerda.
7. Quais três extensões são comuns para a maioria das EBNFs?
8. Diferencie a semântica estática da semântica dinâmica.
9. Para que servem os predicados em uma gramática de atributos?
10. Qual é a diferença entre um atributo sintetizado e um herdado?
11. Como a ordem de avaliação de atributos é determinada para as árvores de uma gramática de atributos?
12. Qual é o principal uso das gramáticas de atributos?
13. Explique os principais usos de metodologia e notação para descrever a semântica de linguagens de programação.
14. Por que as linguagens de máquina não podem ser usadas para definir sentenças em semântica operacional?
15. Descreva os dois níveis de uso da semântica operacional.
16. Na semântica denotacional, o que são os domínios sintáticos e semânticos?
17. O que é armazenado no estado de um programa para semântica denotacional?
18. Que abordagem semântica é mais conhecida?
19. Que duas coisas devem ser definidas para cada entidade de linguagem, de forma a construir uma descrição denotacional da linguagem?
20. Qual parte de uma regra de inferência é o antecedente?
21. O que é uma função de transformação de predicado?
22. O que a corretude parcial significa para um laço?
23. Em que ramo da matemática a semântica axiomática é baseada?
24. Em que ramo da matemática a semântica denotacional é baseada?
25. Qual é o problema de usar um interpretador puro de software para semântica operacional?
26. Explique o que as pré-condições e as pós-condições de uma sentença significam na semântica axiomática.

27. Descreva a estratégia de usar semântica axiomática para provar a corretude de um programa.
28. Descreva o conceito básico de semântica denotacional.
29. De que forma fundamental a semântica operacional e a semântica denotacional diferem?

PROBLEMAS

1. Os dois modelos matemáticos de descrição de linguagens são a geração e o reconhecimento. Descreva como cada um pode definir a sintaxe de uma linguagem de programação.
2. Escreva descrições EBNF para:
   a. Uma sentença de cabeçalho de definição de classe em Java
   b. Uma sentença de chamada a método em Java
   c. Uma sentença **`switch`** em C
   d. Uma definição de **`union`** em C
   e. Literais **`float`** em C
3. Reescreva a BNF do Exemplo 3.4 para dar ao operador **1** precedência sobre * e para forçar + a ser associativo à direita.
4. Reescreva a BNF do Exemplo 3.4 para adicionar os operadores unários ++ e -- de Java.
5. Escreva uma descrição BNF das expressões booleanas de Java, incluindo os três operadores `&&`, `||` e `!` e as expressões relacionais.
6. Usando a gramática do Exemplo 3.2, mostre uma árvore de análise sintática e uma derivação mais à esquerda para cada uma das seguintes sentenças:
   a. `A = A * (B + (C * A))`
   b. `B = C * (A * C + B)`
   c. `A = A * (B + (C))`
7. Usando a gramática do Exemplo 3.4, mostre uma árvore de análise sintática e uma derivação mais à esquerda para cada uma das seguintes sentenças:
   a. `A = (A + B) * C`
   b. `A = B + C + A`
   c. `A = A * (B + C)`
   d. `A = B * (C * (A + B))`
8. Prove que a seguinte gramática é ambígua:

   <S> → <A>
   <A> → <A> + <A> | <id>
   <id> → a | b | c

9. Modifique a gramática do Exemplo 3.4 para adicionar um operador unário de subtração que tenha precedência mais alta que + ou *.

10. Descreva, em português, a linguagem definida pela seguinte gramática:

    <S> → <A> <B> <C>
    <A> → a <A> | a
    <B> → b <B> | b
    <C> → c <C> | c

11. Considere a seguinte gramática:

    <S> → <A> a <B> b
    <A> → <A> b | b
    <B> → a <B> | a

    Quais das sentenças abaixo estão na linguagem gerada por essa gramática?

    a. baab
    b. bbbab
    c. bbaaaaaS
    d. bbaab

12. Considere a seguinte gramática:

    <S> → a <S> c <B> | <A> | b
    <A> → c <A> | c
    <B> → d | <A>

    Quais das sentenças abaixo estão na linguagem gerada por essa gramática?

    a. abcd
    b. acccbd
    c. acccbcc
    d. acd
    e. accc

13. Escreva uma gramática para a linguagem com cadeias que têm *n* cópias da letra a seguidas pelo mesmo número de cópias da letra b, onde $n > 0$. Por exemplo, as cadeias ab, aaaabbbb e aaaaaaaabbbbbbbb estão na linguagem, mas a, abb, ba e aaabb não estão.

14. Escreva árvores de análise sintática para as sentenças aabb e aaaabbbb, derivadas da gramática do Problema 13.

15. Converta a BNF do Exemplo 3.1 em EBNF.

16. Converta a BNF do Exemplo 3.3 em EBNF.

17. Converta a seguinte EBNF em BNF:

    S→A{bA}
    A → a[b]A

18. Qual é a diferença entre um atributo intrínseco e um atributo sintetizado não intrínseco?

19. Escreva uma gramática de atributos cuja base BNF seja a do Exemplo 3.6 da Seção 3.4.5, mas cujas regras de linguagem sejam as seguintes: os tipos de dados não podem ser misturados nas expressões, mas as sentenças de atribuição não precisam ter os mesmos tipos nos dois lados do operador de atribuição.

20. Escreva uma gramática de atributo cuja base da BNF seja a do Exemplo 3.2 e cujas regras de tipo sejam as mesmas do exemplo de sentença de atribuição da Seção 3.4.5.

21. Usando as instruções de máquina virtual dadas na Seção 3.5.1.1, defina a semântica operacional das seguintes construções:

    a. **`do-while`** de Java
    b. **`for`** de Ada
    c. **`if-then-else`** de C++
    d. **`for`** de C
    e. **`switch`** de C

22. Escreva uma função de mapeamento de semântica denotacional para as seguintes sentenças:

    a. **`for`** de Ada
    b. **`do-while`** de Java
    c. expressões booleanas de Java
    d. **`for`** de Java
    e. **`switch`** de C

23. Compute a pré-condição mais fraca para cada uma das seguintes sentenças de atribuição e pós-condições:

    a. `a = 2 * (b - 1) - 1 {a > 0}`
    b. `b = (c + 10) / 3 {b > 6}`
    c. `a = a + 2 * b - 1 {a > 1}`
    d. `x = 2 * y + x - 1 {x > 11}`

24. Compute a pré-condição mais fraca para cada uma das seguintes sequências de sentenças de atribuição e suas pós-condições:

    a.
    ```
    a = 2 * b + 1;
    b = a - 3
    {b < 0}
    ```
    b.
    ```
    a = 3 * (2 * b + a);
    b = 2 * a - 1
    {b > 5}
    ```

25. Compute a pré-condição mais fraca para cada uma das seguintes construções de seleção e suas pós-condições:

a.
```
if (a == b)
   b = 2 * a + 1
else
   b = 2 * a;
{b > 1}
```

b.
```
if (x < y)
   x = x + 1
else
   x = 3 * x
{x < 0}
```

c.
```
if (x > y)
   y = 2 * x + 1
else
   y = 3 * x - 1;
{y > 3}
```

26. Explique os quatro critérios utilizados para provar a corretude de um laço de repetição com pré-teste lógico da forma **while** B **do** S **end**.

27. Prove que (n + 1) * . . . * n = 1.

28. Prove que o seguinte programa está correto:

```
{n > 0}
count = n;
sum = 0;
while count <> 0 do
  sum = sum + count;
  count = count - 1;
end
{sum = 1 + 2 + . . . + n}
```

4

Análise léxica e sintática

4.1 Introdução

4.2 Análise léxica

4.3 O problema da análise sintática

4.4 Análise sintática descendente recursiva

4.5 Análise sintática ascendente

Este capítulo começa com uma introdução à análise léxica, com um exemplo simples. Em seguida, é discutido o problema geral de análise sintática, incluindo as duas principais abordagens a ela e a complexidade do processo. Então, apresentamos a técnica de implementação recursiva descendente para analisadores sintáticos descendentes, incluindo exemplos de partes de um analisador sintático recursivo descendente e a saída de uma análise sintática usando tal analisador. A última seção discute a análise sintática ascendente e o algoritmo de análise sintática LR. A seção inclui um exemplo de uma pequena tabela de análise sintática LR e a análise sintática de uma cadeia realizada por meio do processo de análise sintática LR.

4.1 INTRODUÇÃO

Uma investigação consistente do projeto de compiladores requer ao menos um semestre de estudo intensivo e inclui o projeto e a implementação de um compilador para uma linguagem de programação simples, mas realista. A primeira parte de tal curso é dedicada às análises léxica e sintática. O analisador sintático é o coração de um compilador, porque diversos outros componentes importantes, incluindo o analisador semântico e o gerador de código intermediário, são dirigidos por suas ações.

Alguns leitores podem se perguntar por que um capítulo sobre qualquer parte de um compilador seria incluído em um livro sobre linguagens de programação. Existem ao menos duas razões para uma discussão sobre análises léxica e sintática neste livro: primeiro, os analisadores sintáticos são baseados diretamente nas gramáticas discutidas no Capítulo 3; portanto, é natural discutir esses assuntos como uma aplicação de gramáticas. Segundo, os analisadores léxicos e sintáticos são necessários em numerosas situações fora do contexto do projeto de compiladores. Muitas aplicações, entre elas programas para formatar listagens, para computar a complexidade de programas e para analisar e reagir ao conteúdo de um arquivo de configuração, precisam fazer análises léxica e sintática. Logo, elas são tópicos importantes para os desenvolvedores de software, mesmo que nunca precisem escrever um compilador. Além disso, alguns cursos de ciência da computação não exigem mais que os alunos façam uma disciplina de projeto de compiladores, o que os deixa sem instrução sobre análise léxica e sintática. Nesses casos, este capítulo pode ser usado no curso de linguagens de programação. Em cursos de graduação que tenham uma disciplina obrigatória sobre o projeto de compiladores, ele pode ser omitido.

O Capítulo 1 introduz três estratégias para a implementação de linguagens de programação: compilação, interpretação pura e implementação híbrida. A estratégia de compilação usa um programa chamado compilador, que traduz programas escritos em uma linguagem de programação de alto nível em código de máquina. A compilação é usada para implementar linguagens de programação para grandes aplicações, normalmente escritas em linguagens como C++ e COBOL. Sistemas de interpretação pura não realizam traduções; em vez disso, os programas são interpretados em sua forma original por um software interpretador. A interpretação pura é normalmente usada para sistemas menores, nos quais a eficiência de execução não é crítica, como *scripts* incorporados em documentos HTML, escritos em linguagens como JavaScript. Sistemas de implementação híbridos traduzem programas escritos em linguagens de alto nível em formatos intermediários, os quais são interpretados. Esses sistemas são agora mais usados do que

nunca, em grande parte graças à popularidade das linguagens de *scripting*. Tradicionalmente, sistemas híbridos têm resultado na execução muito mais lenta de programas do que sistemas baseados em compiladores. Entretanto, nos últimos anos, o uso de compiladores Just-in-Time (JIT) têm se ampliado, particularmente para programas Java e programas escritos para o sistema .NET da Microsoft. Um compilador JIT, que traduz código intermediário em código de máquina, é usado em métodos, na primeira vez em que eles são chamados. Na prática, um compilador JIT transforma um sistema híbrido em um sistema de compilação adiada.

Todas as três estratégias de implementação que acabamos de discutir usam analisadores léxicos e sintáticos.

Analisadores sintáticos, ou *parsers*, são quase sempre baseados em uma descrição formal da sintaxe dos programas. O formalismo mais usado para descrição de sintaxe é a gramática livre de contexto, ou BNF, apresentada no Capítulo 3. O uso de BNF, de modo oposto ao uso de algumas descrições informais de sintaxe, tem ao menos três vantagens significativas. Primeiro, as descrições em BNF da sintaxe dos programas são claras e concisas, tanto para humanos quanto para os sistemas de software que as utilizam. Segundo, a descrição em BNF pode ser usada como base direta para o analisador sintático. Terceiro, implementações baseadas em BNF são relativamente fáceis de manter, em função de sua modularidade.

Praticamente todos os compiladores separam a tarefa de analisar a sintaxe em duas partes distintas, as análises léxica e sintática, apesar de essa terminologia ser confusa. O analisador léxico trata de construções de linguagem de pequena escala, como nomes e literais numéricos. O analisador sintático trata de construções de larga escala, como expressões, sentenças e unidades de programas. A Seção 4.2 introduz os analisadores léxicos. As Seções 4.3, 4.4 e 4.5 discutem os analisadores sintáticos.

Existem três razões pelas quais a análise léxica é separada da sintática:

1. Simplicidade – As técnicas de análise léxica são menos complexas que as necessárias para análise sintática; portanto, o processo de análise léxica pode ser mais simples se realizado separadamente. Além disso, remover os detalhes de baixo nível da análise léxica do analisador sintático torna-o menor e menos complexo.
2. Eficiência – Apesar de valer a pena otimizar o analisador léxico, como a análise léxica requer uma porção significativa do tempo total de compilação, não é proveitoso otimizar o analisador sintático. A separação facilita essa otimização seletiva.
3. Portabilidade – Como o analisador léxico lê arquivos de programa de entrada e normalmente inclui a utilização de *buffers* de entrada, ele pode ser dependente de plataforma. Entretanto, o analisador sintático pode ser independente. É sempre bom isolar partes dependentes de máquina de qualquer sistema de software.

4.2 ANÁLISE LÉXICA

Um analisador léxico é basicamente um casador de padrões. Um casador de padrões tenta encontrar uma subcadeia de uma cadeia de caracteres que case com um padrão de caracteres. O casamento de padrões é uma parte tradicional da computação. Um dos primeiros usos do casamento de padrões ocorreu por meio dos editores de texto, como o

editor de linhas `ed`, introduzido em uma das primeiras versões do UNIX. Desde então, o casamento de padrões é utilizado em diversas linguagens de programação – por exemplo, Perl e JavaScript. Ele também está disponível por meio das bibliotecas de classe padrão de Java, C++ e C#.

Um analisador léxico serve como o passo inicial de um analisador sintático. Tecnicamente, a análise léxica é uma parte da sintática. Um analisador léxico realiza análise sintática no nível mais baixo da estrutura dos programas. Um programa de entrada aparece para um compilador como uma única cadeia de caracteres. O analisador léxico coleta caracteres e os agrupa logicamente, atribuindo códigos internos aos agrupamentos, de acordo com sua estrutura. No Capítulo 3, esses agrupamentos lógicos são chamados de **lexemas**, e os códigos internos para as categorias desses agrupamentos são chamados de ***tokens***. Lexemas são reconhecidos por meio do casamento da cadeia de caracteres de entrada com padrões de cadeias de caracteres. Apesar de os *tokens* serem normalmente representados como valores inteiros, por questões de legibilidade dos analisadores léxicos e sintáticos, eles são referenciados por constantes nomeadas.

Considere o exemplo de sentença de atribuição a seguir:

```
result = oldsum - value / 100;
```

A seguir estão os *tokens* e os lexemas dessa sentença:

Token	*Lexema*
IDENT	result
ASSIGN_OP	=
IDENT	oldsum
SUB_OP	-
IDENT	value
DIV_OP	/
INT_LIT	100
SEMICOLON	;

Os analisadores léxicos extraem lexemas de uma cadeia de entrada e produzem os *tokens* correspondentes. Nos primórdios dos compiladores, os analisadores léxicos processavam todo um arquivo de programa de entrada e produziam um arquivo de *tokens* e lexemas. Agora, entretanto, a maioria dos analisadores léxicos é composta de subprogramas que localizam o próximo lexema na entrada, determinam seu código de *token* associado e o retornam para o chamador, o analisador sintático. Assim, cada chamada ao analisador léxico retorna um único lexema e seu *token*. O único aspecto do programa de entrada visto pelo analisador sintático é a saída do analisador léxico, um *token* por vez.

O processo de análise léxica inclui deixar comentários e espaços em branco fora dos lexemas, visto que não são relevantes para o significado do programa. Além disso, o analisador léxico insere lexemas para nomes definidos pelo usuário na tabela de símbolos, usada por fases posteriores do compilador. Por fim, os analisadores léxicos detectam erros sintáticos em *tokens*, como literais de ponto flutuante malformados, e informam tais erros ao usuário.

Existem três estratégias para construir um analisador léxico:

1. Escrever uma descrição formal dos padrões de *token* da linguagem, usando uma linguagem descritiva relacionada a expressões regulares.[1] Essas descrições são usadas como entrada para uma ferramenta de software que gera um analisador léxico automaticamente. Existem muitas ferramentas disponíveis para isso. A mais antiga delas, chamada lex, é incluída como parte dos sistemas UNIX.
2. Projetar um diagrama de transições de estado que descreva os padrões de *tokens* da linguagem e escrever um programa que implemente o diagrama.
3. Descrever um diagrama de transições de estado que descreva os padrões de *tokens* da linguagem e construir manualmente uma implementação dirigida por tabela do diagrama de estados.

Um diagrama de transição de estados, ou apenas **diagrama de estados**, é um grafo dirigido. Os nós de um diagrama de estados são rotulados com nomes de estados. Os arcos são rotulados com os caracteres de entrada que causam as transições entre os estados. Um arco pode incluir também ações que o analisador léxico deve realizar quando a transição ocorrer.

Diagramas de estados como os usados pelos analisadores léxicos são representações de uma classe de máquinas matemáticas chamadas de **autômatos finitos**. Estes podem ser projetados para reconhecer membros de uma classe de linguagens chamada **linguagens regulares**. Gramáticas regulares são dispositivos geradores de linguagens regulares. Os *tokens* de uma linguagem de programação formam uma linguagem regular, e um analisador léxico é um autômato finito.

Agora, ilustraremos a construção de um analisador léxico com um diagrama de estados e o código que o implementa. O diagrama de estados pode incluir simplesmente estados e transições para cada um dos padrões de *tokens*. Entretanto, os resultados dessa estratégia são diagramas muito grandes e complexos, já que cada nó no diagrama de estados precisaria de uma transição para cada caractere no conjunto de caracteres da linguagem analisada. Verificaremos maneiras de simplificar isso.

Suponha que precisássemos de um analisador léxico que reconhecesse como operandos apenas expressões aritméticas, incluindo nomes de variáveis e literais inteiros. Suponha que os nomes de variáveis consistam em cadeias de letras maiúsculas, letras minúsculas e dígitos, mas devam começar com uma letra. Os nomes não têm limitação de tamanho. A primeira observação a ser feita é que existem 52 caracteres diferentes (qualquer letra maiúscula ou minúscula) que podem iniciar um nome, o que exigiria 52 transições a partir do estado inicial do diagrama de transições. Contudo, um analisador léxico só está interessado em determinar se se trata de um nome e não se preocupa com o nome específico. Portanto, definimos uma classe de caracteres chamada LETTER para todas as 52 letras e usamos uma única transição na primeira letra de qualquer nome.

Outra possibilidade de simplificar o diagrama de transição é por meio dos *tokens* que representam literais inteiros. Existem 10 caracteres que podem iniciar um lexema de literal inteiro. Isso exigiria 10 transições a partir do estado inicial do diagrama de

[1] Essas expressões regulares são a base dos recursos de casamento de padrões que agora fazem parte de muitas linguagens de programação, ou diretamente ou por meio de uma biblioteca de classes.

estados. Como dígitos específicos não são uma preocupação do analisador léxico, podemos construir um diagrama de estados muito mais compacto se definirmos uma classe de caracteres chamada DIGIT para dígitos e usarmos uma única transição em qualquer caractere nessa classe para um estado que coleta literais inteiros.

Como nossos nomes contêm dígitos, a transição a partir do nó seguinte ao primeiro caractere de um nome pode usar uma única transição em LETTER ou DIGIT para continuar coletando seus caracteres.

A seguir, definimos alguns subprogramas utilitários para as tarefas comuns dentro do analisador léxico. Primeiro, precisamos de um subprograma, o qual chamamos de `getChar`, que tem diversos deveres. Quando chamado, `getChar` obtém o próximo caractere de entrada do programa de entrada e o coloca na variável global `nextChar`. O subprograma `getChar` também deve determinar a classe do caractere de entrada e colocá-la na variável global `charClass`. O lexema a ser construído pelo analisador léxico, que pode ser implementado como uma cadeia de caracteres ou como um vetor, será chamado de `lexeme`.

Implementamos o processo de colocar o caractere em `nextChar` no vetor `lexeme` em um subprograma chamado `addChar`. Esse subprograma deve ser explicitamente chamado, porque os programas incluem alguns caracteres que não precisam ser colocados em `lexeme`; por exemplo, os caracteres de espaço em branco entre lexemas. Em um analisador léxico mais realista, os comentários também não seriam colocados em `lexeme`.

Quando o analisador léxico é chamado, é conveniente que o próximo caractere da entrada seja o primeiro do próximo lexema. Por isso, uma função chamada `getNonBlank` é usada para pular espaços em branco sempre que o analisador for chamado.

Por fim, um subprograma chamado `lookup` é necessário para computar o código de *token* para os *tokens* de caractere único. Em nosso exemplo, esses são os parênteses e os operadores aritméticos. Os códigos de *token* são números arbitrariamente atribuídos a eles pelo desenvolvedor do compilador.

O diagrama de estados da Figura 4.1 descreve os padrões para nossos *tokens*. Ele inclui as ações necessárias em cada transição do diagrama de estados.

A seguir, temos uma implementação em C de um analisador léxico especificado no diagrama de estados da Figura 4.1, incluindo uma função principal para propósitos de teste:

```
/* front.c – um sistema analisador léxico para
             expressões aritméticas simples */

#include <stdio.h>
#include <ctype.h>

/* Declarações globais */
/* Variáveis */
int  charClass;
char  lexeme [100];
char  nextChar;
int  lexLen;
int  token;
int  nextToken;
```

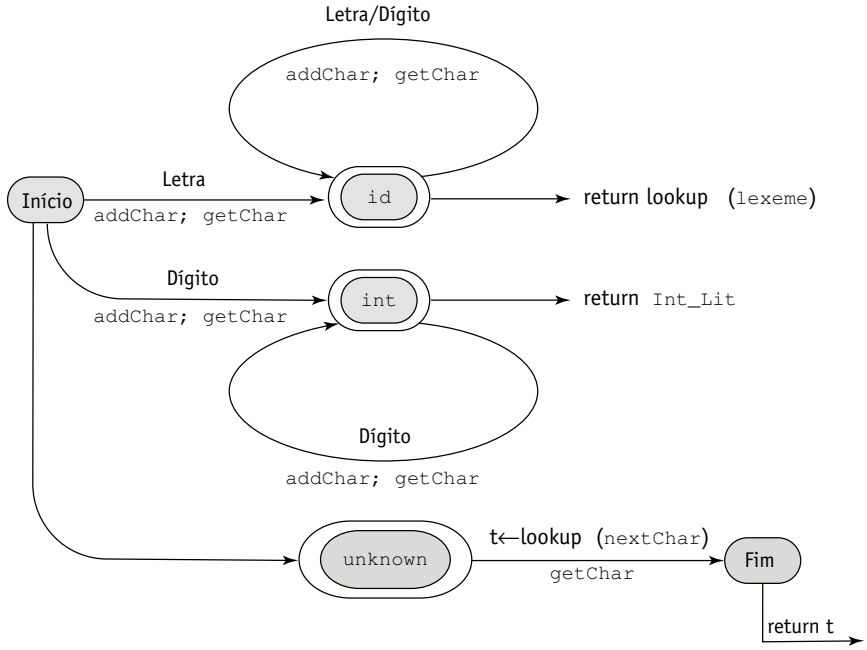


FIGURA 4.1
Um diagrama de estados para reconhecer nomes, parênteses e operadores aritméticos.

```
FILE *in_fp, *fopen();
/* Declarações de função */
void  addChar();
void  getChar();
void  getNonBlank();
int   lex();

/* Classes de caracteres */
#define LETTER 0
#define DIGIT 1
#define UNKNOWN 99

/* Códigos de tokens */
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26
/*****************************************************/
/* função principal */
main() {
```

```
/* Abre o arquivo de dados de entrada e processa seu conteúdo */
   if  ((in_fp = fopen("front.in", "r")) == NULL)
     printf("ERROR - cannot open front.in \n");
   else  {
     getChar();
     do {
       lex();
    }  while  (nextToken != EOF);
  }
}

/*****************************************************/
/* lookup - uma função para processar operadores e parênteses
            e retornar o token  */
int  lookup(char  ch) {
  switch  (ch) {
    case  '(':
      addChar();
      nextToken = LEFT_PAREN;
      break;

    case  ')':
      addChar();
      nextToken = RIGHT_PAREN;
      break;

    case  '+':
      addChar();
      nextToken = ADD_OP;
      break;

    case  '-':
      addChar();
      nextToken = SUB_OP;
      break;
    case  '*':
      addChar();
      nextToken = MULT_OP;
      break;

    case  '/':
      addChar();
      nextToken = DIV_OP;
      break;

    default:
      addChar();
      nextToken = EOF;
      break;
  }
```

```
  return  nextToken;
}

/*****************************************************/
/* addChar - uma função para adicionar nextChar a lexeme */
void addChar() {
  if (lexLen <= 98) {
    lexeme[lexLen++] = nextChar;
    lexeme[lexLen] = 0;
  }
  else
    printf("Error - lexeme is too long \n");
}

/*****************************************************/
/* getChar - uma função para obter o próximo caractere de
             entrada e determinar sua classe */
void getChar() {
  if ((nextChar = getc(in_fp)) != EOF) {
    if (isalpha(nextChar))
      charClass = LETTER;
    else if (isdigit(nextChar))
           charClass = DIGIT;
         else  charClass = UNKNOWN;
   }
   else
     charClass = EOF;
}

/*****************************************************/
/* getNonBlank - uma função para chamar getChar até que ele
                 retorne um caractere que não seja um espaço em
                 branco */
void getNonBlank() {
  while  (isspace(nextChar))
    getChar();
}
/

*****************************************************/
/* lex - um analisador léxico simples para expressões
         aritméticas */
int lex() {
  lexLen = 0;
  getNonBlank();
  switch  (charClass) {
/* Analisa identificadores sintaticamente */
    case  LETTER:
      addChar();
      getChar();
      while  (charClass == LETTER || charClass == DIGIT) {
```

```
        addChar();
        getChar();
      }
    nextToken = IDENT;
    break;

/* Analisa literais sintaticamente */
    case  DIGIT:
      addChar();
      getChar();
      while  (charClass == DIGIT) {
        addChar();
        getChar();
      }
      nextToken = INT_LIT;
      break;

/* Parênteses e operadores */
    case  UNKNOWN:
      lookup(nextChar);
      getChar();
      break;

/* Fim do arquivo */
    case  EOF:
      nextToken = EOF;
      lexeme[0] = 'E';
      lexeme[1] = 'O';
      lexeme[2] = 'F';
      lexeme[3] = 0;
      break;
  } /* Fim do switch */
  printf("Next token is: %d, Next lexeme is %s\n",
         nextToken, lexeme);
  return  nextToken;
}  /* Fim da função lex */
```

Esse código ilustra a relativa simplicidade dos analisadores léxicos. É claro, deixamos de fora o uso de *buffers* de entrada, assim como alguns outros detalhes importantes. Além disso, lidamos com uma linguagem de entrada muito pequena e simples.

Considere a expressão:

```
(sum + 47) / total
```

A seguir, temos a saída do analisador léxico de `front.c` quando usado com essa expressão:

```
Next token is: 25 Next lexeme is (
Next token is: 11 Next lexeme is sum
Next token is: 21 Next lexeme is +
Next token is: 10 Next lexeme is 47
```

```
Next token is: 26 Next lexeme is)
Next token is: 24 Next lexeme is /
Next token is: 11 Next lexeme is total
Next token is: -1 Next lexeme is EOF
```

Nomes e palavras reservadas em programas têm padrões similares. Apesar de ser possível construir um diagrama de estados para reconhecer cada palavra reservada específica de uma linguagem de programação, isso resultaria em um diagrama de estados inviavelmente grande. É muito mais simples e rápido fazer com que o analisador léxico reconheça nomes e palavras reservadas com o mesmo padrão e use uma consulta em uma tabela de palavras reservadas para determinar quais nomes são palavras reservadas. O uso dessa estratégia considera as palavras reservadas como exceções na categoria de nomes de *tokens*.

Um analisador léxico geralmente é responsável pela construção inicial da tabela de símbolos, a qual atua como uma base de dados de nomes para o compilador. As entradas na tabela de símbolos armazenam informações acerca dos nomes definidos pelo usuário, assim como sobre os atributos desses nomes. Por exemplo, se um nome for o de uma variável, o tipo da variável é um de seus atributos que será armazenado na tabela de símbolos. Os nomes normalmente são colocados na tabela de símbolos pelo analisador léxico. Os atributos de um nome são colocados na tabela de símbolos por alguma parte do compilador posterior às ações do analisador léxico.

4.3 O PROBLEMA DA ANÁLISE SINTÁTICA

A parte do processo de analisar a sintaxe denominada *análise sintática* também é chamada de ***parsing***. Usaremos os dois termos como sinônimos.

Esta seção discute o problema geral de análise sintática e apresenta as duas principais categorias de algoritmos de análise sintática, descendente (*top-down*) e ascendente (*bottom-up*), além de discutir a complexidade do processo de análise sintática.

4.3.1 Introdução à análise sintática

Analisadores sintáticos para linguagens de programação constroem árvores de análise sintática para os programas dados. Em alguns casos, a árvore de análise sintática é construída apenas implicitamente, o que significa que talvez seja gerado somente um percurso da árvore. Mas, em todos os casos, as informações necessárias para criar a árvore de análise sintática são geradas durante a análise. Tanto árvores de análise sintática quanto derivações incluem todas as informações sintáticas necessárias para um processador de linguagem.

Existem dois objetivos de análise sintática distintos: primeiro, verificar o programa de entrada para determinar se ele está sintaticamente correto. Quando um erro for encontrado, o analisador deverá produzir uma mensagem de diagnóstico e se recuperar. Nesse caso, recuperação significa voltar a um estado normal e continuar a análise do programa de entrada. Esse passo é necessário para que o compilador encontre o máximo de erros possível durante uma análise do programa de entrada. Se não for feita corretamente, a recuperação de erros pode gerar mais erros, ou pelo menos mais mensagens de erro. O

segundo objetivo da análise sintática é produzir uma árvore de análise sintática completa, ou ao menos percorrer a estrutura da árvore para uma entrada sintaticamente correta. A árvore de análise sintática (ou seu percurso) é usada como base para a tradução.

Os analisadores sintáticos são classificados de acordo com a direção na qual constroem as árvores de análise sintática. As duas classes amplas de analisadores sintáticos são os analisadores **descendentes**, nos quais a árvore é construída da raiz para as folhas, e os **ascendentes**, das folhas para a raiz.

Neste capítulo, usamos um pequeno conjunto de convenções de notação para símbolos gramaticais e cadeias, a fim de tornar a discussão menos desordenada. Para linguagens formais, elas são as seguintes:

1. Símbolos terminais – letras minúsculas do início do alfabeto (a, b, . . .)
2. Símbolos não terminais – letras maiúsculas do início do alfabeto (A, B, . . .)
3. Terminais ou não terminais – letras maiúsculas do final do alfabeto (W, X, Y, Z).
4. Cadeias de terminais – letras minúsculas do final do alfabeto (w, x, y, z).
5. Cadeias mistas (terminais e/ou não terminais) – letras gregas minúsculas (α, β, δ, γ).

Para linguagens de programação, os símbolos terminais são as construções sintáticas de pequena escala, às quais nos referimos como lexemas. Os símbolos não terminais das linguagens de programação são geralmente nomes conotativos ou abreviações, envoltos por sinais de menor que e maior que (< e >) – por exemplo, <while_statement>, <expr> e <function_def>. As sentenças de uma linguagem (programas, no caso de uma linguagem de programação) são cadeias de terminais. Cadeias mistas descrevem lados direitos (RHSs) de regras gramaticais e são usadas nos algoritmos de análise sintática.

4.3.2 Analisadores sintáticos descendentes

Um analisador sintático descendente percorre ou constrói uma árvore de análise sintática em pré-ordem. Um percurso em pré-ordem de uma árvore de análise sintática começa na raiz. Cada nó é visitado antes de seus ramos serem seguidos. Os ramos de determinado nó são seguidos da esquerda para direita. Isso corresponde a uma derivação mais à esquerda.

Em termos de derivação, um analisador sintático descendente pode ser descrito como segue: dada uma forma sentencial que faz parte de uma derivação mais à esquerda, a tarefa do analisador sintático é encontrar a próxima forma sentencial nessa derivação mais à esquerda. A forma geral de uma forma sentencial à esquerda é xAα, em que, de acordo com nossa convenção de notação, x é uma cadeia de símbolos terminais, A é um não terminal e α é uma cadeia mista. Como x contém apenas terminais, A é o não terminal mais à esquerda na forma sentencial, logo é aquele que deve ser expandido para obter a próxima forma sentencial em uma derivação mais à esquerda. Determinar a próxima forma sentencial é uma questão de escolher a regra gramatical correta que tem A como seu LHS. Por exemplo, se a forma sentencial é xAα e as regras A são A $\rightarrow$ bB, A $\rightarrow$ cBb e A $\rightarrow$ a, um analisador sintático descendente deve escolher entre essas três regras para obter a próxima forma sentencial, que poderia ser xbBα, xcBbα ou xaα. Esse é o problema de decisão de análise sintática para analisadores sintáticos descendentes.

Diferentes algoritmos de análise sintática descendente usam informações distintas para tomar decisões de análise sintática. Os analisadores sintáticos descendentes mais comuns escolhem o RHS correto para o não terminal mais à esquerda na forma sentencial corrente, ao comparar o próximo *token* da entrada com os primeiros símbolos que podem ser gerados pelos RHSs dessas regras. O RHS que tiver esse *token* na extremidade esquerda da cadeia que ela gera é o correto. Então, na forma sentencial xAα, o analisador sintático usaria qualquer *token* que fosse o primeiro gerado por A para determinar qual regra de A deveria ser usada para obter a próxima forma sentencial. No exemplo acima, todos os três RHSs das regras A começam com símbolos terminais diferentes. O analisador sintático pode facilmente escolher o RHS correto com base no próximo *token* de entrada, que pode ser a, b, ou c nesse exemplo. De modo geral, a escolha do RHS correto não é tão direta assim, porque alguns dos RHSs do não terminal mais à esquerda na forma sentencial atual podem começar com um não terminal.

Os algoritmos mais comuns de análise sintática descendente são fortemente relacionados. Um **analisador sintático descendente recursivo** é uma versão codificada de um analisador sintático baseado diretamente na descrição BNF da sintaxe da linguagem. A alternativa mais comum para os analisadores descendentes recursivos é usar uma tabela de análise sintática, em vez de código, para implementar as regras BNF. Ambas as alternativas, chamadas de **algoritmos LL**, são igualmente eficientes. Ambas trabalham no mesmo subconjunto de todas as gramáticas livres de contexto. O primeiro L em LL especifica uma varredura da esquerda para a direita da entrada; o segundo especifica que uma derivação mais à esquerda é gerada. A Seção 4.4 apresenta a abordagem descendente recursiva para a implementação de um analisador sintático LL.

4.3.3 Analisadores sintáticos ascendentes

Um analisador sintático ascendente constrói uma árvore de análise sintática começando pelas folhas e progredindo em direção à raiz. Essa ordem de análise sintática corresponde ao inverso de uma derivação mais à direita. Isto é, as formas sentenciais da derivação são produzidas da última para a primeira. Em termos da derivação, um analisador sintático ascendente pode ser descrito como segue: dada uma forma sentencial à direita α, o analisador sintático deve determinar qual subcadeia de α é o RHS da regra na gramática que deve ser reduzido para seu LHS, a fim de produzir a forma sentencial anterior na derivação mais à direita. Por exemplo, o primeiro passo de um analisador sintático ascendente é determinar qual subcadeia da sentença inicial dada é o RHS a ser reduzido ao LHS correspondente para obter a penúltima forma sentencial na derivação. O processo de encontrar o RHS correto a reduzir é complicado pelo fato de uma forma sentencial à direita poder incluir mais de um RHS da gramática da linguagem que está sendo analisada sintaticamente. O RHS correto é chamado de **manipulador** (*handle*). A forma sentencial à direita é uma forma sentencial que aparece em uma derivação mais à direita.

Considere a seguinte gramática e derivação:

S → aAc
A → aA | b

S => aAc => aaAc => aabc

Um analisador sintático ascendente dessa sentença, aabc, começa com a sentença e deve encontrar o manipulador nela. Nesse exemplo, é uma tarefa fácil, visto que a cadeia contém apenas um RHS, b. Quando o analisador sintático substitui b por seu LHS, A, ele obtém a penúltima forma sentencial na derivação, aaAc. No caso geral, conforme mencionado anteriormente, encontrar o manipulador é muito mais difícil, porque uma forma sentencial pode incluir diversos RHSs diferentes.

Um analisador sintático ascendente encontra o manipulador de uma forma sentencial examinando os símbolos em um ou em ambos os lados de um manipulador possível. Símbolos à direita do manipulador são normalmente *tokens* da entrada que ainda não foram analisados.

Os algoritmos mais comuns de análise sintática ascendente estão na família LR, na qual o L especifica uma varredura da esquerda para a direita da entrada e o R especifica que uma derivação mais à direita é gerada.

4.3.4 A complexidade da análise sintática

Os algoritmos de análise sintática que funcionam para qualquer gramática não ambígua são complicados e ineficientes. Na prática, a complexidade de tais algoritmos é $O(n^3)$; ou seja, a quantidade de tempo que eles levam é na ordem do cubo do tamanho da cadeia a ser analisada. Essa quantidade de tempo relativamente grande é necessária porque esses algoritmos geralmente precisam voltar e reanalisar parte da sentença analisada. A reanálise sintática será necessária quando o analisador sintático tiver cometido um erro no processo de análise. Voltar o analisador sintático também exige que a parte com problemas da árvore de análise sintática que está sendo construída (ou seu percurso) seja desmantelada e reconstruída. Os algoritmos $O(n^3)$ normalmente não são úteis para processos práticos, como a análise sintática para um compilador, porque são lentos demais. Em situações como essa, os cientistas da computação buscam algoritmos mais rápidos, apesar de menos gerais. A generalidade é trocada pela eficiência. Em termos de análise sintática, foram encontrados algoritmos mais rápidos funcionando para apenas um subconjunto do conjunto de todas as gramáticas possíveis. Esses algoritmos são aceitáveis, desde que o subconjunto inclua gramáticas que descrevam linguagens de programação. (Na verdade, conforme discutido no Capítulo 3, a classe inteira de gramáticas livres de contexto é inadequada para descrever toda a sintaxe da maioria das linguagens de programação.)

Todos os algoritmos usados pelos analisadores sintáticos dos compiladores comerciais têm complexidade O(n), ou seja, o tempo que eles levam é linearmente relacionado ao tamanho da cadeia que está sendo analisada sintaticamente. Tais algoritmos são amplamente mais eficientes que os algoritmos $O(n^3)$.

4.4 ANÁLISE SINTÁTICA DESCENDENTE RECURSIVA

Esta seção apresenta o processo de implementação de análise sintática descendente recursiva.

4.4.1 O processo de análise sintática descendente recursiva

Um analisador sintático descendente recursivo é chamado assim porque consiste em uma coleção de subprogramas, muitos recursivos, e produz uma árvore de análise sintática em ordem descendente. Essa recursão é um reflexo da natureza das linguagens de programação, que inclui diversos tipos de estruturas aninhadas. Por exemplo, sentenças são geralmente aninhadas em outras sentenças. Os parênteses em expressões também devem ser aninhados apropriadamente. A sintaxe dessas estruturas é naturalmente descrita com regras gramaticais recursivas.

EBNF é idealmente adequada para analisadores sintáticos descendentes recursivos. Lembre-se, do Capítulo 3, de que as principais extensões da EBNF eram as chaves, as quais especificam que o que está envolto entre elas pode aparecer zero ou mais vezes, e os colchetes, os quais especificam que o que está envolto entre eles pode aparecer uma vez ou nenhuma. Note que, em ambos os casos, os símbolos envoltos são opcionais. Considere os exemplos a seguir:

<if_statement> → **`if`** <logic_expr> <statement> [**`else`** <statement>]
<ident_list> → ident {, ident}

Na primeira regra, a cláusula **`else`** de uma sentença **`if`** é opcional. Na segunda, um <ident_list> é um identificador, seguido por zero ou mais repetições de uma vírgula e um identificador.

Um analisador sintático descendente recursivo tem um subprograma para cada não terminal em sua gramática associada. A responsabilidade do subprograma associado a determinado não terminal é a seguinte: quando informada uma cadeia de entrada, ele percorre a árvore de análise sintática que pode ser enraizada naquele não terminal e cujas folhas casam com a cadeia de entrada. Na prática, um subprograma de análise sintática descendente recursiva é um analisador sintático para a linguagem (conjunto de cadeias) gerada por seu não terminal associado.

Considere a seguinte descrição EBNF de expressões aritméticas simples:

<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | int_constant | (<expr>)

Lembre-se de que uma gramática EBNF para expressões aritméticas não impõe nenhuma regra de associatividade. Logo, quando tal gramática for usada como base de um compilador, deve-se tomar cuidado para garantir que o processo de geração de código, normalmente dirigido pela análise sintática, produza código que satisfaça as regras de associatividade da linguagem. Isso pode ser feito facilmente quando a análise sintática descendente recursiva é usada.

Na função descendente recursiva a seguir, `expr`, o analisador léxico é a função implementada na Seção 4.2. Ele obtém o próximo lexema e coloca seu código de *token* na variável global `nextToken`. Os códigos de *tokens* são definidos como constantes nomeadas, como na Seção 4.2.

Um subprograma descendente recursivo para uma regra com um único RHS é relativamente simples. Para cada símbolo terminal no RHS, esse símbolo é comparado com `nextToken`. Se eles não casam, isso é um erro sintático. Se eles casam, o analisador léxico é chamado para obter o próximo *token* de entrada. Para cada não terminal, o subprograma de análise sintática para o não terminal é chamado.

O subprograma descendente recursivo para a primeira regra da gramática de exemplo anterior, escrito em C, é

```
/* expr
   Analisa sintaticamente cadeias na linguagem gerada pela regra:
   <expr> -> <term> {(+ | -) <term>}
   */
void  expr() {
  printf("Enter <expr>\n");

/* Analisa sintaticamente o primeiro termo */
  term();

/* Desde que o próximo token seja + ou -, obtém o próximo
   token e analisa sintaticamente o próximo termo */
  while  (nextToken == ADD_OP || nextToken == SUB_OP) {
    lex();
    term();
  }
  printf("Exit <expr>\n");
}  /* Fim da função expr */
```

Note que a função `expr` inclui sentenças de saída de percurso para produzir a saída de exemplo mostrada mais adiante nesta seção.

Os subprogramas de análise sintática descendente recursiva são escritos com a convenção de que cada um deixa o próximo *token* de entrada em `nextToken`. Então, sempre que uma função de análise sintática começa, ela presume que `nextToken` tem o código para o *token* mais à esquerda da entrada que ainda não foi usado no processo de análise sintática.

A parte da linguagem que a função `expr` analisa sintaticamente consiste em um ou mais termos, separados pelo operador de adição ou pelo de subtração. Essa é a linguagem gerada pelo não terminal <expr>. Logo, primeiro ela chama a função que analisa sintaticamente os termos (`term`). Então, continua para chamar essa função desde que encontre os *tokens* `ADD_OP` ou `SUB_OP` (que são passados por meio de uma chamada a `lex`). Essa função descendente recursiva é mais simples que a maioria, porque sua regra associada tem apenas um RHS. Além disso, ela não inclui qualquer código para detecção ou recuperação de erros de sintaxe, porque não existem erros detectáveis associados à regra gramatical.

Um subprograma de análise sintática descendente recursiva para um não terminal cuja regra tenha mais de um RHS começa com código para determinar qual RHS deve ser analisado sintaticamente. Cada RHS é examinado (em tempo de construção do compilador) para determinar o conjunto de símbolos terminais que podem aparecer no início de sentenças que ele pode gerar. Ao associar esses conjuntos com o próximo *token* de entrada, o analisador sintático pode escolher o RHS correto.

O subprograma de análise sintática para <term> é similar àquele para <expr>:

```
/* term
   Analisa sintaticamente cadeias na linguagem gerada pela regra:
   <term> -> <factor> {(* | /) <factor>)
   */
void  term() {
  printf("Enter <term>\n");

/* Analisa sintaticamente o primeiro fator */
  factor();

/* Desde que o próximo token seja * ou /, obtém o próximo
   token e analisa sintaticamente o próximo fator */
  while  (nextToken == MULT_OP || nextToken == DIV_OP) {
    lex();
    factor();
  }
  printf("Exit <term>\n");
}  /* Fim da função term */
```

A função para o não terminal <factor> de nossa gramática de expressões aritméticas deve escolher entre seus dois RHSs. Ela também inclui detecção de erros. Na função para <factor>, a reação ao detectar um erro de sintaxe é simplesmente chamar a função de erro. Em um analisador sintático real, quando um erro é detectado, deve ser produzida uma mensagem de diagnóstico. Além disso, os analisadores sintáticos devem se recuperar do erro de forma que o processo de análise sintática possa continuar.

```
/* factor
   Analisa sintaticamente cadeias na linguagem gerada pela regra:
   <factor> -> id | int_constant | (<expr)
   */
void factor() {
  printf("Enter <factor>\n");

/* Determina qual RHS */
  if (nextToken == IDENT || nextToken == INT_LIT)
/* Obtém o próximo token */
    lex();

/* Se o RHS é ( <expr>), chama lex para ignorar o
   parêntese à esquerda, chama expr e busca o
   parêntese à direita */
  else  {
    if  (nextToken == LEFT_PAREN) {
      lex();
      expr();
      if (nextToken == RIGHT_PAREN)
        lex();
      else
        error();
```

```
    }  /* Fim do if (nextToken == ... */

/* Não era um identificador, um literal inteiro ou um
   parêntese à esquerda */
    else
      error();
  }  /* Fim do else */

  printf("Exit <factor>\n");;
}  /* Fim da função factor */
```

A seguir, temos a saída da análise sintática da expressão de exemplo `(sum + 47) / total`, usando as funções de análise sintática `expr`, `term` e `factor`, e a função `lex` da Seção 4.2. Note que a análise sintática começa pela chamada a `lex` e pela rotina do símbolo inicial, nesse caso, `expr`.

```
Next token is: 25 Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11 Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21 Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10 Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 26 Next lexeme is)
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24 Next lexeme is /
Exit <factor>
Next token is: 11 Next lexeme is total
Enter <factor>
Next token is: -1 Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```

A árvore de análise sintática percorrida pelo analisador sintático para a expressão anterior é mostrada na Figura 4.2.

Mais um exemplo de regra gramatical e função de análise sintática deve ajudar a solidificar o entendimento do leitor sobre a análise sintática descendente recursiva. A seguir, temos uma descrição gramatical da sentença **`if`** do Java:

<ifstmt> → **`if`** (<boolexpr>) <statement> [**`else`** <statement>]

O subprograma descendente recursivo para essa regra é:

```
/* Função ifstmt
    Analisa sintaticamente cadeias na linguagem gerada pela regra:
    <ifstmt> -> if (<boolexpr>) <statement>
                    [else <statement>]
    */
void ifstmt() {
/* Certifica-se de que o primeiro token é 'if' */
  if  (nextToken != IF_CODE)
    error();
  else  {
/* Chama lex para obter o próximo token */
    lex();
/* Verifica se existe o parêntese esquerdo */
     if  (nextToken != LEFT_PAREN)
      error();
     else  {
/* Analisa sintaticamente a expressão booleana */
      boolexpr();
/* Verifica se existe o parêntese direito */
     if (nextToken != RIGHT_PAREN)
       error();
     else  {
/* Analisa sintaticamente a cláusula then */
        statement();
/* Se um else for o próximo, analisa sintaticamente a cláusula else
```

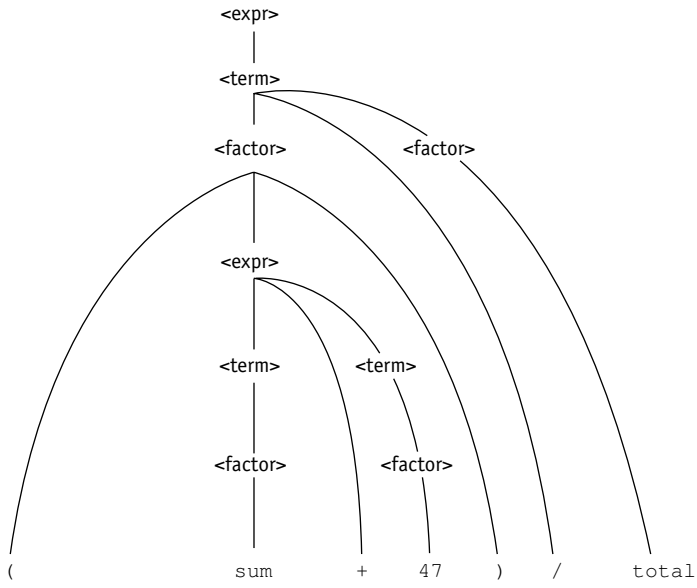


FIGURA 4.2
Árvore de análise sintática para (sum + 47) / total.

```
*/
          if  (nextToken == ELSE_CODE) {
/* Chama lex para passar pelo else */
           lex();
           statement();
         } /* fim do if (nextToken == ELSE_CODE ... */
       } /* fim do else de if (nextToken != RIGHT ... */
     } /* fim do else de if (nextToken != LEFT ... */
   } /* fim do else de if (nextToken != IF_CODE ... */
} /* fim de ifstmt */
```

Note que essa função usa funções de análise sintática para sentenças e expressões booleanas que não são dadas nesta seção.

O objetivo desses exemplos é convencer você de que um analisador sintático descendente recursivo pode ser facilmente escrito, se uma gramática apropriada estiver disponível para a linguagem. As características de uma gramática que permite um analisador sintático descendente recursivo ser construído são discutidas na subseção seguinte.

4.4.2 A classe de gramáticas LL

Antes de optar pela estratégia de análise sintática descendente recursiva para um compilador ou para outra ferramenta de análise de programas, devem ser consideradas as limitações da estratégia em termos de restrições gramaticais. Esta seção discute tais restrições e suas possíveis soluções.

Uma característica simples das gramáticas que causa um problema catastrófico para analisadores sintáticos LL é a recursão à esquerda. Por exemplo, considere a seguinte regra:

$A \rightarrow A + B$

Um subprograma de um analisador sintático descendente recursivo para A imediatamente chama a si mesmo para analisar sintaticamente o primeiro símbolo em seu RHS. Então, a ativação do subprograma A do analisador chama imediatamente a si mesmo mais uma vez, e novamente, e assim por diante. É fácil ver que isso não leva a nada (exceto a um estouro de pilha).

A recursão à esquerda na regra $A \rightarrow A + B$ é chamada de **recursão à esquerda direta** porque ocorre em uma regra. A recursão à direita direta pode ser eliminada de uma gramática com o seguinte processo:

Para cada não terminal, A,

- Agrupe as regras A como $A \rightarrow A\alpha 1, \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ onde nenhum dos βs começa com A
- Substitua as regras A originais por

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_m A' \mid \varepsilon$$

Note que ε e especifica a cadeia vazia. Uma regra que tem ε como seu lado direito (RHS) é chamada de *regra de apagamento*, porque seu uso em uma derivação efetivamente apaga seu LHS da forma sentencial.

Considere a seguinte gramática de exemplo e a aplicação do processo acima:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

Para as regras E, temos $\alpha 1 = + T$ e $\beta = T$, então substituímos as regras E por

$E \rightarrow T E'$
$E' \rightarrow + T E' \mid \varepsilon$

Para as regras T, temos $\alpha_1 = *F$ e β, então substituímos as regras T por

$T \rightarrow F T'$
$T' \rightarrow * F T' \mid \varepsilon$

Como não existe recursão à esquerda nas regras F, elas permanecem as mesmas, então a gramática substituta completa é

$E \rightarrow T E'$
$E' \rightarrow + T E' \mid \varepsilon$
$T \rightarrow F T'$
$T' \rightarrow * F T' \mid \varepsilon$
$F \rightarrow (E) \mid id$

Essa gramática gera a mesma linguagem que a original, mas não é recursiva à esquerda.

Como era o caso da gramática de expressões escrita com EBNF na Seção 4.4.1, essa gramática não especifica associatividade à esquerda dos operadores. Entretanto, é relativamente fácil projetar a geração de código baseada nessa gramática de forma que os operadores de adição e multiplicação tenham associatividade à esquerda.

A recursão à esquerda indireta apresenta o mesmo problema da recursão à esquerda direta. Por exemplo, suponha que tivéssemos

$A \rightarrow B\ a\ A$
$B \rightarrow A\ b$

Um analisador sintático descendente recursivo para essas regras faria um subprograma de A chamar imediatamente o subprograma de B, o qual chamaria imediatamente o subprograma de A. Assim, o problema é igual ao da recursão à esquerda direta. O problema da recursão à esquerda não está restrito à abordagem descendente recursiva para a construção de analisadores sintáticos descendentes. É um problema que ocorre em todos os algoritmos de análise sintática descendente. Felizmente, a recursão à esquerda não é um problema para os algoritmos de análise sintática ascendente.

Existe um algoritmo para modificar uma gramática a fim de remover a recursão à esquerda indireta (Aho et al., 2006), mas ele não é abordado aqui. Durante a escrita de uma gramática para uma linguagem de programação, pode-se evitar a inclusão da recursão à esquerda, tanto direta quanto indireta.

A recursão à esquerda não é a única característica gramatical que impede a análise sintática descendente. Outra ocorre se o analisador sintático puder sempre escolher o RHS correto com base no próximo símbolo de entrada, usando apenas o primeiro símbolo gerado pelo não terminal mais à esquerda na forma sentencial atual. Existe um teste relativamente simples para uma gramática não recursiva à esquerda que indica se isso pode ser feito, o **teste de disjunção par a par**. Esse teste requer a habilidade de computar um conjunto com base nos lados direitos de um símbolo não terminal dado em uma gramática. Tais conjuntos, chamados FIRST, são definidos como

FIRST (α) = {a | α =>* aβ} (If α =>* ε, ε is in FIRST(α))

No qual =>* significa "em 0 ou mais passos de derivação".

Um algoritmo para computar FIRST para qualquer cadeia mista α pode ser encontrado em Aho et al. (2006). Para os nossos propósitos, FIRST pode ser computado pela inspeção da gramática.

O teste de disjunção par a par é como segue:

> Para cada não terminal, A, na gramática que tem mais de um lado direito, para cada par de regras, $A \rightarrow \alpha_i$ e $A \rightarrow \alpha_j$, deve ser verdadeiro que
>
> $$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \phi$$
>
> (A intersecção dos dois conjuntos, FIRST(α_i) e FIRST(α_j), deve ser vazia.)

Em outras palavras, se um não terminal A tiver mais de um RHS, o primeiro símbolo terminal que pode ser gerado em uma derivação para cada um deles deve ser único àquele RHS. Considere as seguintes regras:

A → aB | bAb | Bb
B → cB | d

Os conjuntos FIRST para os RHSs das regras A são {a}, {b} e {c}, {d}, que são claramente disjuntos. Logo, essas regras passam no teste de disjunção par a par. Em termos de um analisador sintático descendente recursivo, isso significa que o código do subprograma para analisar sintaticamente o não terminal A pode escolher qual RHS ele está tratando ao ver o primeiro símbolo terminal da entrada (*token*) gerado pelo não terminal. Agora considere as regras

A → aB | BAb
B → aB | b

Os conjuntos FIRST para os RHSs nas regras A são {a} e {a}, {b}, claramente não disjuntos. Logo, essas regras falham no teste de disjunção par a par. Em termos do analisador sintático, o subprograma para A não pode determinar qual RHS foi analisado sintaticamente olhando para o próximo símbolo da entrada, porque, se ele for um a, pode ser qualquer um dos RHSs. Essa questão é mais complexa se um ou mais RHSs começam com não terminais.

Em muitos casos, uma gramática que falha no teste de disjunção par a par pode ser modificada para passar no teste. Por exemplo, considere a regra

<variable> → identifier | identifier [<expression>]

Essa regra diz que uma variável (<variable>) é um identificador ou um identificador seguido por uma expressão entre colchetes (um índice). Essas regras não passam no teste de disjunção par a par, porque ambos os RHSs começam com o mesmo terminal, chamado identifier. Esse problema pode ser atenuado com o processo de **fatoração à esquerda**.

Agora, adotaremos uma visão informal para a fatoração à esquerda. Considere nossas regras para <variable>. Ambos os RHSs começam com identifier. As partes que seguem identifier nos dois RHSs são α (a cadeia vazia) e [<expression>]. As duas regras podem ser substituídas pelas duas regras a seguir:

<variable> → identifier <new>

<new> → ε | [<expression>]

Não é difícil notar que, juntas, essas duas regras geram a mesma linguagem das duas regras com as quais iniciamos. Entretanto, as duas passam no teste de disjunção par a par.

Se a gramática é usada como base para um analisador sintático descendente recursivo, está disponível uma alternativa à fatoração à esquerda. Com uma extensão EBNF, o problema desaparece de uma forma bastante similar à solução da fatoração à esquerda. Considere as regras originais acima para <variable>. O índice pode se tornar opcional se colocado entre colchetes, como em

<variable> → identifier [[<expression]]

Nessa regra, os colchetes externos são metassímbolos indicando que o que está dentro deles é opcional. Os colchetes internos são símbolos terminais da linguagem de programação que é descrita. A questão é que substituímos duas regras por uma que gera a mesma linguagem, mas que passa no teste de disjunção par a par.

Um algoritmo formal para fatoração à esquerda pode ser encontrado em Aho et al. (2006). A fatoração à esquerda não pode resolver todos os problemas de disjunção par a par. Em alguns casos, as regras devem ser reescritas de outras maneiras para eliminar o problema.

4.5 ANÁLISE SINTÁTICA ASCENDENTE

Esta seção apresenta o processo geral da análise sintática ascendente e uma descrição do algoritmo de análise sintática LR.

4.5.1 O problema da análise sintática para analisadores sintáticos ascendentes

Considere a seguinte gramática para expressões aritméticas:

E → E + T | T
T → T * F | F
F → (E) | id

Note que essa gramática gera as mesmas expressões aritméticas do exemplo da Seção 4.4. A diferença é que essa é recursiva à esquerda, o que é aceitável para analisadores sintáticos ascendentes. Note também que as gramáticas para analisadores sintáticos ascendentes normalmente não incluem metassímbolos, como aqueles usados para especificar extensões à BNF. A seguinte derivação mais à direita ilustra essa gramática:

E => E + T
  => E + T * F
  => E + T * id
  => E + F * id
  => E + id * id
  => T + id * id
  => F + id * id
  => id + id * id

A parte sublinhada de cada forma sentencial nessa derivação é o RHS reescrito como seu LHS correspondente para obter a forma sentencial anterior. O processo de análise sintática ascendente produz o inverso de uma derivação mais à direita. Então, na derivação de exemplo, um analisador ascendente começa com a última forma sentencial (a sentença de entrada) e produz a sequência de formas sentenciais a partir dela, até que só reste o símbolo inicial, que nessa gramática é E. Em cada passo, a tarefa do analisador sintático ascendente é encontrar o RHS específico, o manipulador, na forma sentencial que deve ser reescrita para obter a próxima (anterior) forma sentencial. Conforme mencionado anteriormente, uma forma sentencial à direita pode incluir mais de um RHS. Por exemplo, a forma sentencial à direita

E + T * id

inclui três RHSs, E + T, T e id. Apenas um desses é o manipulador. Por exemplo, se o RHS E + T fosse escolhido para ser reescrito nessa forma sentencial, a forma sentencial resultante seria E * id, mas E * id não é uma forma sentencial à direita para a gramática dada.

O manipulador de uma forma sentencial à direita é único. A tarefa de um analisador sintático ascendente é encontrar o manipulador de qualquer forma sentencial à direita que possa ser gerado por sua gramática associada. Formalmente, o manipulador é definido como:

> Definição: β é o **manipulador** da forma sentencial direita $\gamma = \alpha\beta w$ se e somente se $S =>^*_{rm} \alpha A w =>_{rm} \alpha\beta w$.

Nessa definição, $=>_{rm}$ especifica um passo de derivação mais à direita, e $=>^*_{rm}$ especifica zero ou mais passos de derivação mais à direita. Apesar de a definição de um manipulador ser matematicamente concisa, ela fornece pouca ajuda em encontrar o manipulador de uma forma sentencial à direita. A seguir, fornecemos as definições de diversas subcadeias de formas sentenciais relacionadas aos manipuladores. O objetivo delas é fornecer alguma percepção acerca dos manipuladores.

Definição: β é uma **frase** da forma sentencial à direita γ se
e somente se S => $*\gamma = \alpha_1 A \alpha_2 =>+ \alpha_1 \beta \alpha_2$

Nessa definição, => + significa um ou mais passos de derivação.

Definição: β é uma **frase simples** da forma sentencial à direita γ se
e somente se S => $*\gamma = \alpha_1 A \alpha_2 => + \alpha_1 \beta \alpha_2$

Se essas duas definições forem comparadas cuidadosamente, está claro que diferem apenas na última especificação de derivação. A definição de frase usa um ou mais passos, enquanto a definição de frase simples usa exatamente um passo.

As definições de frase e de frase simples talvez pareçam ter a mesma falta de valor prático que os manipuladores, mas isso não é verdade. Considere o que uma frase é em relação a uma árvore de análise sintática. Ela é a cadeia de todas as folhas da árvore de análise sintática parcial que tem como raiz determinado nó interno da árvore de análise sintática completa. Uma frase simples é aquela que realiza um único passo de derivação a partir de seu nó não terminal raiz. Em termos de uma árvore de análise sintática, uma frase pode ser derivada de um único não terminal em um ou mais níveis da árvore, mas uma frase simples pode ser derivada em apenas um nível da árvore. Considere a árvore de análise sintática mostrada na Figura 4.3.

As folhas dessa árvore correspondem à forma sentencial E + T * id. Como existem três nós internos, existem três frases. Cada nó interno é a raiz de uma subárvore, cujas folhas são uma frase. O nó raiz da árvore de análise sintática completa, E, gera toda a forma sentencial resultante, E + T * id, que é uma frase. O nó interno, T, gera as folhas T * id, outra frase. Finalmente, o nó interno, F, gera id, também uma frase. Então, as frases da forma sentencial E + T * id são E + T * id, T * id e id. Note que as frases não são necessariamente RHSs na gramática correspondente.

As frases simples formam um subconjunto das frases. No exemplo anterior, a única frase simples é id. Uma frase simples é sempre um RHS da gramática.

O motivo para discutirmos frases e frases simples é este: o manipulador de qualquer forma sentencial mais à direita é sua frase simples mais à esquerda. Então, agora temos

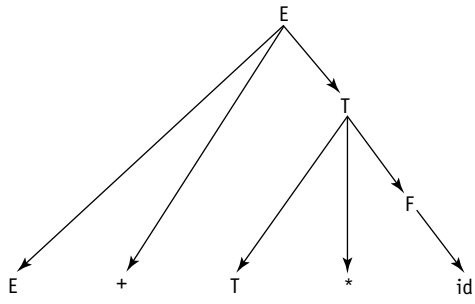


FIGURA 4.3
Uma árvore de análise sintática para E + T * id.

uma forma intuitiva de encontrar o manipulador de quaisquer formas sentenciais à direita, supondo que temos a gramática e podemos desenhar uma árvore de análise sintática. É claro que essa estratégia de encontrar manipuladores não é prática para um analisador sintático. (Se você já tem uma árvore de análise sintática, por que precisaria de um analisador sintático?) Seu único propósito é fornecer ao leitor alguma ideia intuitiva sobre o que é um manipulador, em relação a uma árvore de análise sintática, que é mais fácil do que tentar pensar sobre manipuladores em termos de formas sentenciais.

Agora, podemos considerar a análise sintática ascendente em termos de árvores de análise sintática, apesar de o propósito de um analisador sintático ser a produção de uma árvore de análise sintática. Dada a árvore para uma sentença inteira, você pode encontrar o manipulador, que é o primeiro item a ser reescrito na sentença para se obter a forma sentencial anterior. Então, o manipulador pode ser removido da árvore de análise sintática e o processo pode ser repetido. Continuando até a raiz da árvore de análise sintática, toda a derivação mais à direita pode ser construída.

4.5.2 Algoritmos de deslocamento e redução (*shift-reduce*)

Analisadores sintáticos ascendentes são chamados de **algoritmos de deslocamento e redução** (*shift-reduce*), porque deslocar e reduzir são as duas ações mais comuns que especificam. Uma parte integrante de todo analisador sintático ascendente é a pilha. Assim como nos outros analisadores sintáticos, em um analisador ascendente a entrada é o fluxo de *tokens* de um programa, e a saída é uma sequência de regras gramaticais. A ação de deslocar (*shift*) move o próximo símbolo de entrada para a pilha do analisador sintático. Uma ação de redução (*reduce*) substitui um RHS (o manipulador) no topo da pilha do analisador sintático por seu LHS correspondente. Todo analisador sintático para uma linguagem de programação é um **autômato de pilha** (**PDA**), pois um PDA é um reconhecedor para uma linguagem livre de contexto. Você não precisa entender profundamente os PDAs para entender como um analisador sintático ascendente funciona, apesar de isso ajudar. Um PDA é uma máquina matemática simples que varre cadeias de símbolos da esquerda para a direita. Um PDA é assim chamado porque usa uma pilha como memória. Os PDAs podem ser usados como reconhecedores para linguagens livres de contexto. Dada uma cadeia de símbolos de um alfabeto de uma linguagem livre de contexto, um PDA que seja projetado para o propósito pode determinar se a cadeia é ou não é uma sentença da linguagem. No processo, o PDA pode produzir a informação necessária para construir a árvore de análise sintática para a sentença.

Com um PDA, a cadeia de entrada é examinada, um símbolo de cada vez, da esquerda para a direita. A entrada é tratada de forma bastante similar à que seria se estivesse em outra pilha, porque o PDA nunca vê além do símbolo mais à esquerda da entrada.

Note que um analisador sintático descendente recursivo também é um PDA. Nesse caso, a pilha é a do sistema em tempo de execução, a qual grava chamadas a subprogramas (entre outras coisas), que correspondem aos não terminais da gramática.

4.5.3 Analisadores sintáticos LR

Muitos algoritmos diferentes de análise sintática ascendente foram inventados. Em sua maioria, são variações de um processo chamado LR. Os analisadores sintáticos LR usam

um programa relativamente pequeno e uma tabela de análise sintática construída para uma linguagem de programação específica. O algoritmo LR original foi projetado por Donald Knuth (Knuth, 1965). Chamado de **LR canônico**, não foi usado nos anos imediatamente subsequentes à sua publicação porque produzir a tabela de análise sintática necessária exigia muito tempo e memória computacional. Subsequentemente, diversas variações do processo de construção de tabela do LR canônico foram desenvolvidas (DeRemer, 1971; DeRemer e Pennello, 1982). Elas são caracterizadas por duas propriedades: (1) exigem bem menos recursos que o algoritmo LR canônico para produzir a tabela de análise sintática necessária e (2) trabalham em classes de gramáticas menores que o algoritmo LR canônico.

Existem três vantagens no uso de analisadores sintáticos LR:

1. Eles podem ser construídos para todas as linguagens de programação.
2. Eles podem detectar erros de sintaxe o mais cedo possível em uma varredura da esquerda para a direita.
3. A classe de gramáticas LR é um superconjunto da classe analisável sintaticamente por analisadores LL (por exemplo, muitas gramáticas recursivas à esquerda são LR, mas nenhuma é LL).

A única desvantagem da análise LR é a dificuldade de produzir manualmente a tabela de análise sintática de gramática para uma linguagem de programação completa. Essa não é uma desvantagem séria, entretanto, já que existem diversos programas disponíveis que recebem uma gramática como entrada e produzem a tabela de análise sintática, conforme discutido posteriormente nesta seção.

Antes da aparição do algoritmo de análise sintática LR, existiam alguns algoritmos que encontravam manipuladores de formas sentenciais à direita por meio da inspeção, tanto para a esquerda quanto para a direita da subcadeia, da forma sentencial que se suspeitava ser o manipulador. A descoberta de Knuth foi que efetivamente era possível procurar à esquerda do manipulador sob suspeita, até o final da pilha de análise sintática, para determinar se ele seria o manipulador. Mas todas as informações na pilha de análise sintática relevantes para o processo de análise poderiam ser representadas por um único estado, o qual poderia ser armazenado no topo da pilha. Em outras palavras, Knuth descobriu que, independentemente do tamanho da cadeia de entrada, do tamanho da forma sentencial ou da profundidade da pilha de análise sintática, existia apenas um número relativamente pequeno de situações diferentes nas quais o processo de análise estava interessado. Cada situação poderia ser representada por um estado e armazenada na pilha de análise sintática, um símbolo de estado para cada símbolo da gramática na pilha. No topo da pilha sempre haveria um símbolo de estado, o qual representava a informação relevante de toda a história da análise sintática até o momento. Usaremos um S em letra maiúscula, com subscritos, para representar os estados do analisador sintático.

A Figura 4.4 mostra a estrutura de um analisador sintático LR. O conteúdo da pilha de análise sintática para um analisador LR tem a seguinte forma:

$S_0X_1S_1X_2 \ldots X_mS_m$ (top)

onde os Ss são os símbolos de estado e os Xs são os da gramática. Uma configuração de um analisador LR é um par de cadeias (pilha, entrada), com a forma detalhada

$(S_0X_1S_1X_2S_2 \ldots X_mS_m, a_ia_{i+1} \ldots a_n\$)$

Note que a cadeia de entrada tem um cifrão na extremidade direita. O sinal é colocado lá durante a inicialização do analisador sintático. Ele é usado para o término normal do analisador sintático. Com essa configuração do analisador sintático, podemos definir formalmente o processo de análise sintática LR, o qual é baseado na tabela de análise sintática.

Uma tabela de análise sintática LR tem duas partes, chamadas ACTION e GOTO. A parte ACTION da tabela especifica a maior parte do que o analisador sintático faz. Ela tem símbolos de estado como rótulos de linhas e os símbolos terminais da gramática como rótulos de colunas. Dado um estado atual do analisador sintático, representado pelo símbolo de estado no topo da pilha de análise sintática, e o próximo símbolo (*token*) de entrada, a tabela de análise sintática especifica o que o analisador sintático deve fazer. As duas ações principais do analisador sintático são o deslocamento e a redução. Ou o analisador sintático desloca o próximo símbolo de entrada para a árvore de análise sintática, ou ele já tem o manipulador no topo da pilha, o qual reduz para o LHS da regra cujo RHS é o mesmo do manipulador. Duas outras ações são possíveis: aceitar (*accept*), ou seja, o analisador sintático concluiu com êxito a análise sintática da entrada, e erro (*error*), ou seja, o analisador sintático detectou um erro de sintaxe.

As linhas da parte GOTO da tabela de análise sintática LR têm símbolos de estado como rótulos. Essa parte da tabela tem não terminais como rótulos de colunas. Os valores na parte GOTO da tabela indicam qual símbolo de estado deve ser inserido na pilha de análise sintática após uma redução ter sido concluída; ou seja, o manipulador foi removido da árvore de análise sintática e o novo não terminal foi inserido na pilha de análise sintática. O símbolo específico é encontrado na linha cujo rótulo é o símbolo de estado no

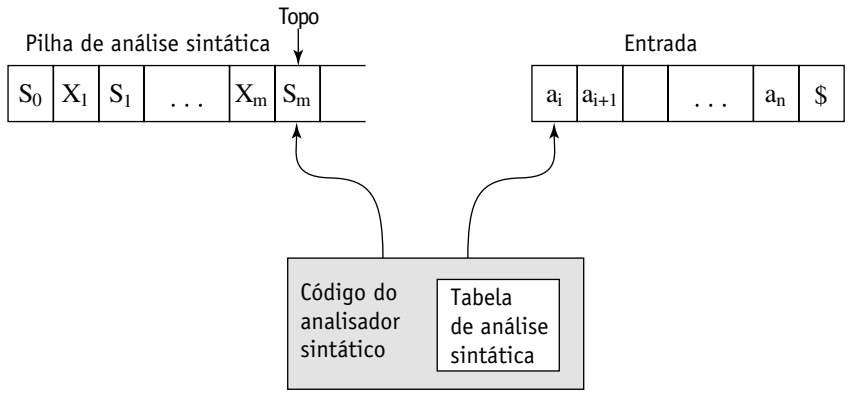


FIGURA 4.4
A estrutura de um analisador sintático LR.

topo da pilha de análise sintática, após o manipulador e seus símbolos de estado associados terem sido removidos. A coluna da tabela GOTO usada é aquela com o rótulo que é o LHS da regra usada na redução.

Considere a gramática tradicional para expressões aritméticas:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

As regras dessa gramática são numeradas para fornecer uma maneira simples de referenciá-las em uma tabela de análise sintática.

A Figura 4.5 mostra a tabela para essa gramática. Abreviações são usadas para as ações: R para redução e S para deslocamento. R4 significa reduzir usando a regra 4; S6

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

FIGURA 4.5
A tabela de análise sintática LR para uma gramática de expressões aritméticas.

significa deslocar o próximo símbolo da entrada na pilha e inserir o estado S6 na pilha. Posições vazias na tabela ACTION indicam erros de sintaxe. Em um analisador sintático, essas posições poderiam ter chamadas a rotinas de tratamento de erros.

As tabelas de análise sintática LR podem ser facilmente construídas com uma ferramenta de software, como a yacc (Johnson, 1975), que recebe a gramática como entrada. Apesar de as tabelas de análise sintática LR poderem ser produzidas manualmente, para uma gramática de uma linguagem de programação real, a tarefa pode ser extensa, tediosa e passível de erros. Para compiladores reais, as tabelas de análise sintática LR são geradas com ferramentas de software.

A configuração inicial de um analisador sintático LR é

$(S_0, a_1 \ldots a_n\$)$

As ações do analisador sintático são informalmente definidas como segue:

1. O processo de *shift* é simples: o próximo símbolo da entrada é inserido na pilha, com o símbolo de estado que é parte da especificação de deslocamento na tabela ACTION.
2. Para uma ação *reduce*, o manipulador deve ser removido da pilha. Como para cada símbolo da gramática na pilha existe um símbolo de estado, o número de símbolos removidos da pilha é duas vezes o de símbolos no manipulador. Após a remoção do manipulador e de seus símbolos de estado associados, o lado esquerdo da regra é inserido na pilha. Por fim, a tabela GOTO é usada, com o rótulo da linha sendo o símbolo exposto quando o manipulador e seus símbolos de estado foram removidos, e o rótulo da coluna sendo o não terminal que é o lado esquerdo da regra usada na redução.
3. Quando a ação é Accept, a análise sintática está completa e nenhum erro foi encontrado.
4. Quando a ação é Error, o analisador sintático chama uma rotina de tratamento de erros.

Apesar de existirem muitos algoritmos de análise sintática baseados no conceito LR, eles diferem apenas na construção da tabela de análise sintática. Todos os analisadores sintáticos usam o mesmo algoritmo de análise.

Talvez a melhor forma de conhecer o processo de análise sintática LR seja com um exemplo. Inicialmente, a pilha de análise sintática tem o único símbolo 0, que representa o estado 0 do analisador sintático. A entrada contém a cadeia de entrada com um marcador de fim, nesse caso um cifrão, anexado ao final da cadeia. A cada passo, as ações do analisador sintático são ditadas pelo símbolo mais ao topo da pilha de análise sintática (mais à direita, na Figura 4.4) e pelo próximo símbolo de entrada (mais à esquerda, na Figura 4.4). A ação correta é escolhida a partir da célula correspondente da parte ACTION da tabela de análise sintática. A parte GOTO da tabela de análise sintática é usada após uma ação de redução. Lembre-se de que GOTO é usada para determinar qual símbolo de estado é colocado na pilha de análise sintática após a redução.

A seguir, é mostrada uma listagem da saída de um analisador sintático usando a cadeia id + id * id, o algoritmo de análise sintática LR e a tabela de análise sintática mostrada na Figura 4.5.

Pilha	*Entrada*	*Ação*
0	id + id * id $	Deslocar 5
0id5	+ id * id $	Reduzir 6 (use GOTO[0, F])
0F3	+ id * id $	Reduzir 4 (use GOTO[0, T])
0T2	+ id * id $	Reduzir 2 (use GOTO[0, E])
0E1	+ id * id $	Deslocar 6
0E1+6	id * id $	Deslocar 5
0E1+6id5	* id $	Reduzir 6 (use GOTO[6, F])
0E1+6F3	* id $	Reduzir 4 (use GOTO[6, T])
0E1+6T9	* id $	Deslocar 7
0E1+6T9*7	id $	Deslocar 5
0E1+6T9*7id5	$	Reduzir 6 (use GOTO[7, F])
0E1+6T9*7F10	$	Reduzir 3 (use GOTO[6, T])
0E1+6T9	$	Reduzir 1 (use GOTO[0, E])
0E1	$	Aceitar

Os algoritmos para gerar tabelas de análise sintática LR para gramáticas dadas, descritos em Aho et al. (2006), não são muito complexos, mas estão além dos objetivos de um livro sobre linguagens de programação. Conforme mencionado anteriormente, existem vários sistemas de software diferentes para gerar tabelas de análise sintática LR.

RESUMO

A análise sintática é uma parte comum da implementação de linguagens, independentemente da estratégia de implementação usada. Ela normalmente é baseada em uma descrição de sintaxe formal da linguagem que está sendo implementada. Uma gramática livre de contexto, também chamada de BNF, é a estratégia mais comum para descrever sintaxe. A tarefa de análise sintática é dividida em duas partes: análise léxica e análise sintática. Existem diversas razões para separar a análise léxica – entre elas, a simplicidade, a eficiência e a portabilidade.

Um analisador léxico é um casador de padrões que isola as partes de pequena escala de um programa, chamadas de lexemas. Os lexemas ocorrem em categoriais, como os literais inteiros e os nomes. Essas categorias são chamadas de *tokens*. Para cada *token*, é atribuído um código numérico, que, com o lexema, é o que o analisador léxico produz. Existem três estratégias distintas para a construção de um analisador léxico: usar uma ferramenta de software para gerar uma tabela para um analisador dirigido por tabela, construir tal tabela manualmente e escrever código para implementar uma descrição de um diagrama de estados dos *tokens* da linguagem que está sendo implementada. O diagrama de estados para *tokens* pode ser razoavelmente pequeno se as classes de caracteres forem usadas para transições, em vez de se ter transições para cada possível caractere para cada nó de estado. Além disso, o diagrama de estados pode ser simplificado por meio de uma tabela de busca para reconhecer palavras reservadas.

Os analisadores sintáticos têm dois objetivos: detectar erros sintáticos em um programa e construir uma árvore de análise sintática, ou apenas a informação necessária

para construir tal árvore. Ou os analisadores sintáticos são descendentes, o que significa que constroem derivações mais à esquerda e uma árvore de análise sintática de maneira descendente, ou ascendentes, quando constroem o inverso de uma derivação mais à direita e uma árvore de análise sintática de maneira ascendente. Analisadores sintáticos que funcionam para todas as gramáticas não ambíguas têm complexidade $O(n^3)$. Entretanto, analisadores sintáticos usados para implementar analisadores para linguagens de programação funcionam em subclasses de gramáticas não ambíguas e têm complexidade O(n).

Um analisador sintático descendente recursivo é um analisador sintático LL implementado escrevendo-se código diretamente da gramática da linguagem fonte. EBNF é ideal como base para os analisadores sintáticos descendentes recursivos. Um analisador sintático descendente recursivo tem um subprograma para cada não terminal da gramática. O código para uma regra gramatical é simples se a regra tem somente um lado direito. O lado direito é examinado da esquerda para a direita. Para cada não terminal, o código chama o subprograma associado que analisa sintaticamente aquilo que o não terminal gerar. Para cada terminal, o código compara o terminal com o próximo *token* da entrada. Se eles casarem, o código chama o analisador léxico para obter o próximo *token*. Se não casarem, o subprograma relata um erro de sintaxe. Se uma regra tem mais de um lado direito, o subprograma deve primeiro determinar qual lado direito ela deve analisar sintaticamente. Deve ser possível determinar isso com base no próximo *token* de entrada.

Duas características distintas das gramáticas impedem a construção de um analisador sintático descendente recursivo baseado na gramática. Uma delas é a recursão à esquerda. O processo de eliminar a recursão à esquerda direta de uma gramática é relativamente simples. Apesar de não abordarmos isso, existe um algoritmo para remover tanto a recursão à esquerda direta quanto a indireta. O outro problema é detectado no teste de disjunção par a par, que testa se um subprograma de análise sintática pode determinar qual lado direito está sendo analisado sintaticamente com base no próximo *token* de entrada. Algumas gramáticas que falham no teste de disjunção par a par podem ser modificadas para passar no teste, usando fatoração à esquerda.

O problema da análise sintática para analisadores sintáticos ascendentes é encontrar a subcadeia da forma sentencial atual que deve ser reduzida para seu lado esquerdo associado, a fim de obter a próxima (anterior) forma sentencial na derivação mais à direita. Essa subcadeia é chamada de manipulador da forma sentencial. Uma árvore de análise sintática pode fornecer uma maneira intuitiva de reconhecer um manipulador. Um analisador sintático ascendente é um algoritmo de deslocamento-redução, porque, na maioria dos casos, ele desloca o próximo lexema da entrada para a pilha de análise sintática ou reduz o manipulador que está no topo da pilha.

A família LR de analisadores sintáticos de deslocamento e redução é a abordagem de análise sintática ascendente mais usada para linguagens de programação, visto que analisadores sintáticos dessa família têm diversas vantagens em relação às alternativas. Um analisador sintático LR usa uma pilha de análise sintática, a qual contém símbolos gramaticais e símbolos de estado para manter o estado do analisador sintático. O símbolo no topo da pilha de análise sintática é sempre um símbolo de estado que representa toda a informação na pilha que é relevante para o processo. Analisadores sintáticos usam duas tabelas de análise sintática: ACTION e GOTO. A parte ACTION especifica o que o analisador sintático deve fazer, dados o símbolo de estado no topo da pilha de análise

sintática e o próximo *token* de entrada. A tabela GOTO é usada para determinar qual símbolo de estado deve ser colocado na pilha após a realização de uma redução.

QUESTÕES DE REVISÃO

1. Quais são as três razões pelas quais os analisadores sintáticos são baseados em gramáticas?
2. Explique as três razões pelas quais a análise léxica é separada da análise sintática.
3. Defina *lexema* e *token*.
4. Quais são as tarefas primárias de um analisador léxico?
5. Descreva brevemente as três abordagens para a construção de um analisador léxico.
6. O que é um diagrama de transição de estados?
7. Por que são usadas classes de caracteres, em vez de caracteres individuais, para as transições de letras e dígitos de um diagrama de estados de um analisador léxico?
8. Quais são os dois objetivos distintos da análise sintática?
9. Descreva as diferenças entre analisadores sintáticos descendentes e ascendentes.
10. Descreva o problema de análise sintática para um analisador sintático descendente.
11. Descreva o problema de análise sintática para um analisador sintático ascendente.
12. Explique por que os compiladores usam algoritmos de análise sintática que funcionam em apenas um subconjunto de todas as gramáticas.
13. Por que são usadas constantes nomeadas, em vez de números, para códigos de *tokens*?
14. Descreva como um subprograma de análise sintática descendente recursiva é escrito para uma regra com um único lado direito.
15. Explique as duas características das gramáticas que as impedem de serem usadas como a base para um analisador sintático descendente.
16. O que é o conjunto FIRST para uma gramática e forma sentencial?
17. Descreva o teste de disjunção par a par.
18. O que é fatoração à esquerda?
19. O que é uma frase de uma forma sentencial?
20. O que é uma frase simples de uma forma sentencial?
21. O que é o manipulador de uma forma sentencial?
22. Qual é a máquina matemática em que os analisadores sintáticos descendentes e ascendentes são baseados?
23. Descreva três vantagens dos analisadores sintáticos LR.

24. Qual foi a descoberta de Knuth ao desenvolver a técnica de análise sintática LR?
25. Descreva o propósito da tabela ACTION de um analisador sintático LR.
26. Descreva o propósito da tabela GOTO de um analisador sintático LR.
27. A recursão à esquerda é um problema para analisadores sintáticos LR?

PROBLEMAS

1. Faça o teste de disjunção par a par para as seguintes regras gramaticais.

   a. A → aB | b | cBB
   b. B → aB | bA | aBb
   c. C → aaA | b | caB

2. Faça o teste de disjunção par a par para as seguintes regras gramaticais.

   a. S → aSb | bAA
   b. A → b{aB} | a
   c. B → aB | a

3. Mostre a saída do analisador sintático descendente recursivo dado na Seção 4.4.1 para a cadeia `a + b * c`.
4. Mostre a saída do analisador sintático descendente recursivo dado na Seção 4.4.1 para a cadeia `a * (b + c)`.
5. Dada a seguinte gramática e a forma sentencial à direita, desenhe uma árvore de análise sintática e mostre as frases e as frases simples, assim como o manipulador.

   S → aAb | bBA A → ab | aAB B → aB | b

   a. aaAbb
   b. bBab
   c. aaAbBb

6. Dada a seguinte gramática e a forma sentencial à direita, desenhe uma árvore de análise sintática e mostre as frases e as frases simples, assim como o manipulador.

   S → AbB | bAc A → Ab | aBB B → Ac | cBb | c

   a. aAcccbbc
   b. AbcaBccb
   c. baBcBbbc

7. Mostre uma análise sintática completa, incluindo o conteúdo da pilha de análise sintática, a cadeia de entrada e as ações para a cadeia id * (id + id), usando a gramática e a tabela de análise sintática da Seção 4.5.3.

8. Mostre uma análise sintática completa, incluindo o conteúdo da pilha de análise sintática, a cadeia de entrada e as ações para a cadeia (id + id) * id, usando a gramática e a tabela de análise sintática da Seção 4.5.3.
9. Obtenha o algoritmo para remover a recursão à esquerda indireta de uma gramática de Aho et al. (2006). Use esse algoritmo para remover todas as recursões à esquerda da seguinte gramática: S → Aa | Bb A → Aa | Abc | c | Sb B → bb

EXERCÍCIOS DE PROGRAMAÇÃO

1. Projete um diagrama de estados para reconhecer uma das formas de comentários das linguagens de programação baseadas em C, aquela que inicia com `/*` e termina com `*/`.
2. Projete um diagrama de estados para reconhecer os literais de ponto flutuante de sua linguagem de programação favorita.
3. Escreva e teste o código para implementar o diagrama de estados do Problema 1.
4. Escreva e teste o código para implementar o diagrama de estados do Problema 2.
5. Modifique o analisador léxico dado na Seção 4.2 para que reconheça a lista de palavras reservas a seguir e retorne seus respectivos códigos de *token*: **for** (FOR_CODE, 30), **if** (IF_CODE, 31), **else** (ELSE_CODE, 32), **while** (WHILE_CODE, 33), **do** (DO_CODE, 34), **int** (INT_CODE, 35), **float** (FLOAT_CODE, 36), **switch** (SWITCH_CODE, 37).
6. Converta o analisador léxico (escrito em C) dado na Seção 4.2 em Java.
7. Converta as rotinas de análise sintática descendente recursiva para <expr>, <term> e <factor> dadas na Seção 4.4.1 em Java.
8. Para as regras que passaram no teste no Problema 1, escreva um subprograma de análise sintática descendente recursiva que analise sintaticamente a linguagem gerada pelas regras. Suponha que você tem um analisador léxico chamado `lex` e um subprograma de tratamento de erros denominado `error`, chamado sempre que um erro de sintaxe for detectado.
9. Para as regras que passaram no teste no Problema 2, escreva um subprograma de análise sintática descendente recursiva que analise sintaticamente a linguagem gerada pelas regras. Suponha que você tem um analisador léxico chamado `lex` e um subprograma de tratamento de erros denominado `error`, chamado sempre que um erro de sintaxe for detectado.
10. Implemente e teste o algoritmo de análise sintática LR dado na Seção 4.5.3.
11. Escreva uma regra EBNF que descreva a sentença **while** de Java ou C++. Escreva o subprograma descendente recursivo em Java ou C++ para essa regra.
12. Escreva uma regra EBNF que descreva a sentença **for** de Java ou C++. Escreva o subprograma descendente recursivo em Java ou C++ para essa regra.

Esta página foi deixada em branco intencionalmente.

5

Nomes, vinculações e escopos

5.1 Introdução

5.2 Nomes

5.3 Variáveis

5.4 O conceito de vinculação

5.5 Escopo

5.6 Escopo e tempo de vida

5.7 Ambientes de referenciamento

5.8 Constantes nomeadas

Este capítulo apresenta os problemas semânticos fundamentais das variáveis. Ele começa descrevendo a natureza dos nomes e palavras especiais nas linguagens de programação. Os atributos das variáveis, incluindo o tipo, o endereço e o valor, são discutidos, assim como a questão do uso de apelidos (*aliases*). Os importantes conceitos de vinculação e tempos de vinculação são apresentados em seguida, incluindo os diferentes tempos de vinculação possíveis para atributos de variáveis e como eles definem quatro diferentes categorias de variáveis. Depois disso, são descritas duas regras para o escopo de nomes, que pode ser estático ou dinâmico, com o conceito de um ambiente de referenciamento para uma sentença. Por fim, são discutidas as constantes nomeadas e a inicialização de variáveis.

5.1 INTRODUÇÃO

As linguagens de programação imperativas são, em graus diferentes, abstrações da arquitetura de computadores subjacente de von Neumann. Os dois principais componentes da arquitetura são sua memória, que armazena tanto instruções quanto dados, e seu processador, que fornece operações para modificar o conteúdo da memória. As abstrações para as células de memória da máquina em uma linguagem são as variáveis. Em alguns casos, as características das abstrações e das células são muito parecidas. Um exemplo disso é uma variável inteira, normalmente representada diretamente em um ou mais bytes de memória. Em outros casos, as abstrações são muito distantes da organização de memória em hardware, como nas matrizes tridimensionais, que exigem uma função de mapeamento em software para oferecer suporte à abstração.

Uma variável pode ser caracterizada por uma coleção de propriedades, ou atributos, das quais a mais importante é o tipo, um conceito fundamental em linguagens de programação. O projeto dos tipos de dados de uma linguagem exige considerar diversas questões. (Os tipos de dados são discutidos no Capítulo 6.) Entre as mais importantes estão o escopo e o tempo de vida das variáveis.

As linguagens de programação funcionais permitem nomear expressões. Essas expressões nomeadas aparecem como atribuições a nomes de variável em linguagens imperativas, mas são fundamentalmente diferentes, pois não podem ser alteradas. Portanto, elas são como as constantes nomeadas das linguagens imperativas. As linguagens funcionais puras não têm variáveis como as das linguagens imperativas. Contudo, muitas linguagens funcionais incluem tais variáveis.

No restante deste livro, famílias de linguagens serão referenciadas como se fossem linguagens únicas. Por exemplo, Fortran significará todas as versões de Fortran. Esse também será o caso para Ada. Da mesma forma, uma referência para C significará a versão original de C, assim como C89 e C99. Quando uma versão específica de uma linguagem é usada, é porque ela é diferente dos outros membros da família no tópico em discussão. Se adicionarmos um sinal de adição (+) ao nome da versão de uma linguagem, queremos nos referir a todas as versões da linguagem a partir daquela mencionada. Por exemplo, Fortran 95+ significa todas as versões de Fortran a partir de Fortran 95. A frase **linguagens baseadas em C** será usada para fazer referência a C, Objective-C, C++, Java e C#.[1]

[1]Ficamos tentados a incluir como baseadas em C as linguagens de *scripting* JavaScript e PHP, mas decidimos que elas são diferentes demais de suas ancestrais.

5.2 NOMES

Antes de iniciarmos nossa discussão sobre variáveis, devemos abordar o projeto de um dos atributos fundamentais das variáveis: os nomes. Nomes também são associados a subprogramas, parâmetros formais e outras construções de programa. O termo *identificador* é muito usado como sinônimo de *nome*.

5.2.1 Questões de projeto

As principais questões de projeto para nomes são:

- Os nomes são sensíveis à diferenciação de maiúsculas/minúsculas?
- As palavras especiais da linguagem são palavras reservadas ou palavras-chave?

Essas questões são discutidas nas duas subseções a seguir, com exemplos de diversas escolhas de projeto.

5.2.2 Formato de nomes

Um **nome** é uma cadeia de caracteres usada para identificar alguma entidade em um programa.

C99 não tem limitação em seus nomes internos, mas apenas os 63 primeiros são significativos. Em C99, os nomes externos (definidos fora das funções, que devem ser manipulados pelo ligador) são restritos a 31 caracteres. Nomes em Java e C# não têm limites de tamanho, e todos os caracteres são significativos. C++ não especifica um limite de tamanho para nomes, embora os implementadores limitem, às vezes.

» *nota histórica*

As primeiras linguagens de programação usavam nomes com um caractere. Essa notação era natural porque, no início, a programação era principalmente matemática, e os matemáticos têm usado nomes com um único caractere há muito tempo para variáveis em suas notações formais.

Fortran I quebrou com a tradição de nomes de um único caractere, permitindo até seis em seus nomes.

Na maioria das linguagens de programação, os nomes têm o mesmo formato: uma letra seguida por uma cadeia de letras, dígitos e sublinhados (_). Apesar de o uso de sublinhados ter sido disseminado nos anos 1970 e 1980, a prática é menos popular atualmente. Nas linguagens baseadas em C, eles foram, em grande medida, substituídos pela assim chamada *notação camelo* (CamelCase), na qual, em um nome de várias palavras, todas elas, exceto a primeira, começam com maiúsculas, como em `myStack`.[2] Note que o uso de sublinhados e de caixa mista em nomes é uma questão de estilo de programação, não de projeto de linguagem.

Todos os nomes de variáveis em PHP devem começar com um cifrão. Em Perl, o caractere especial no início do nome de uma variável, `$`, `@` ou `%`, especifica o seu tipo (embora em um sentido diferente do de outras linguagens). Em Ruby, caracteres espe-

[2]A notação é chamada "camelo" porque as palavras nela escritas muitas vezes têm letras maiúsculas incorporadas, as quais se parecem com as corcovas de um camelo.

ciais no início do nome, `@` ou `@@`, indicam que a variável é uma instância ou uma variável de classe, respectivamente.

Em muitas linguagens, mais notavelmente nas baseadas em C, as letras maiúsculas e minúsculas nos nomes são distintas; ou seja, as linguagens são **sensíveis à diferenciação de maiúsculas/minúsculas**. Por exemplo, os três nomes a seguir são distintos em C++: `rosa`, `ROSA` e `Rosa`. Para algumas pessoas, esse é um sério detrimento à legibilidade, porque nomes que se parecem denotam entidades diferentes. Nesse sentido, essa sensibilidade viola o princípio de projeto que diz que as construções de linguagem parecidas devem ter significados parecidos. Mas em linguagens cujos nomes de variáveis são sensíveis à diferenciação de maiúsculas/minúsculas, apesar de `Rosa` e `rosa` parecerem similares, não existe uma conexão entre elas.

Obviamente, nem todo mundo concorda que a sensibilidade à diferenciação de maiúsculas/minúsculas é ruim para nomes. Em C, os problemas da sensibilidade à diferenciação de maiúsculas/minúsculas são evitados pela convenção de que os nomes das variáveis não devem incluir letras maiúsculas. Em Java e C#, entretanto, o problema não pode ser evitado porque muitos dos nomes predefinidos incluem tanto maiúsculas quando minúsculas. Por exemplo, o método em Java para converter uma cadeia em um valor inteiro é `parseInt`. Nesse caso, as variações `ParseInt` e `parseint` não são reconhecidas. Esse é um problema de facilidade de escrita e não de legibilidade, porque a necessidade de lembrar o uso de diferenciação de maiúsculas/minúsculas específica torna mais difícil a escrita correta de programas. É um tipo de intolerância da parte do projetista da linguagem que é verificada e garantida pelo compilador.

5.2.3 Palavras especiais

tPalavras especiais em linguagens de programação são usadas para tornar os programas mais legíveis ao nomearem as ações a serem realizadas. Elas também são usadas para separar as partes sintáticas das sentenças e programas. Na maioria das linguagens, as palavras especiais são classificadas como palavras reservadas, o que significa que não podem ser redefinidas pelos programadores. Mas, em algumas, como em Fortran, elas são apenas palavras-chave, ou seja, podem ser redefinidas.

Uma **palavra reservada** é uma palavra especial de uma linguagem de programação que não pode ser usada como um nome. Há um problema em potencial com as palavras reservadas: se a linguagem inclui um grande número delas, o usuário tem dificuldades para inventar nomes que não sejam reservados. O melhor exemplo disso é COBOL, que tem 300 palavras reservadas. Infelizmente, alguns dos nomes mais escolhidos pelos programadores estão na lista das palavras reservadas – como, por exemplo, `LENGTH`, `BOTTOM`, `DESTINATION` e `COUNT`.

Nos códigos de programa de exemplo deste livro, as palavras reservadas são apresentadas em negrito.

Na maioria das linguagens, nomes que são definidos em outras unidades de programa, como os pacotes em Java e as bibliotecas em C e C++, podem se tornar visíveis para um programa. Esses nomes são predefinidos, mas visíveis apenas se explicitamente importados. Uma vez importados, eles não podem ser redefinidos.

5.3 VARIÁVEIS

Uma variável de programa é uma abstração de uma célula de memória de um computador ou de uma coleção de células. Os programadores geralmente pensam em variáveis como nomes para locais de memória, mas elas são muito mais do que apenas um nome.

A mudança das linguagens de máquina para as de montagem foi motivada pela necessidade de substituir por nomes os endereços numéricos e absolutos de memória para dados, tornando os programas muito mais legíveis e fáceis de serem escritos e mantidos. As linguagens de montagem também forneceram uma saída para o problema do endereçamento absoluto feito de forma manual, porque o tradutor que convertia os nomes em endereços reais também escolhia esses endereços.

Uma variável pode ser caracterizada por seis atributos: nome, endereço, valor, tipo, tempo de vida e escopo. Apesar de isso parecer muito complicado para um conceito aparentemente tão simples, essa caracterização é a maneira mais clara de explicar os vários aspectos das variáveis.

Nossa discussão sobre os atributos das variáveis levará ao exame dos importantes conceitos relacionados, como o uso de apelidos, a vinculação, os tempos de vinculação, as declarações, as regras de escopo e os ambientes de referenciamento.

Os atributos nome, endereço, tipo e valor das variáveis são discutidos nas seções seguintes. Os atributos tempo de vida e escopo são tratados nas Seções 5.4.3 e 5.5, respectivamente.

5.3.1 Nome

Os nomes de variáveis são os mais comuns nos programas. Eles são tratados extensivamente na Seção 5.2, no contexto geral dos nomes de entidades em programas. A maioria das variáveis é nomeada. Aquelas que não são nomeadas são tratadas na Seção 5.4.3.3.

5.3.2 Endereço

O **endereço** de uma variável é o endereço de memória de máquina ao qual ela está associada. Essa associação não é tão simples como pode parecer. Em muitas linguagens, é possível que a mesma variável seja associada a diferentes endereços em vários momentos durante a execução do programa. Por exemplo, se um subprograma tem uma variável local alocada a partir da pilha de tempo de execução quando o subprograma é chamado, diferentes chamadas podem resultar em a variável ter diferentes endereços. Essas são, em certo sentido, instanciações diferentes da mesma variável.

O processo de associar variáveis a endereços é discutido em mais detalhes na Seção 5.4.3. Um modelo de implementação para subprogramas e suas ativações é discutido no Capítulo 10.

O endereço de uma variável é algumas vezes chamado de seu **valor esquerdo** (*l-value*), porque o endereço é o que é necessário quando o nome de uma variável aparece no lado esquerdo de uma atribuição.

É possível haver diversas variáveis com o mesmo endereço. Quando mais de um nome de variável pode ser usado para acessar a mesma posição de memória, as variáveis são chamadas de **apelidos** (*aliases*). O uso de apelidos é um problema para a legibilidade, porque permite que uma variável tenha seu valor modificado por uma atribuição a

uma variável diferente. Por exemplo, se variáveis chamadas `total` e `sum` são apelidos, quaisquer mudanças no valor de `total` também modificam o valor de `sum` e vice-versa. Um leitor do programa deve sempre se lembrar de que `total` e `sum` são nomes diferentes para a mesma célula de memória. Como pode existir qualquer número de apelidos em um programa, isso pode ser muito difícil na prática. O uso de apelidos também dificulta a verificação de programas.

Apelidos podem ser criados de diversas formas nos programas. Uma maneira comum em C e C++ é por meio de seus tipos de união. Uniões são discutidas no Capítulo 6.

Duas variáveis de ponteiro são apelidos quando apontam para a mesma posição de memória. O mesmo ocorre com as variáveis de referência. Esse tipo de uso de apelidos é um efeito colateral da natureza dos ponteiros e das referências. Quando um ponteiro em C++ é configurado para apontar para uma variável nomeada, o ponteiro (quando desreferenciado) e o nome da variável são apelidos.

Apelidos podem ser criados em muitas linguagens por meio de parâmetros de subprogramas. Esses tipos de apelidos são discutidos no Capítulo 9.

O momento no qual uma variável se associa a um endereço é muito importante para o entendimento das linguagens de programação. Esse assunto é discutido na Seção 5.4.3.

5.3.3 Tipo

O **tipo** de uma variável determina a faixa de valores que ela pode armazenar e o conjunto de operações definidas para valores do tipo. Por exemplo, o tipo **`int`** em Java especifica uma faixa de valores de –2147483648 a 2147483647 e operações aritméticas para adição, subtração, multiplicação, divisão e módulo.

5.3.4 Valor

O **valor** de uma variável é o conteúdo da(s) célula(s) de memória associada(s) a ela. É conveniente pensar na memória de um computador em termos de células *abstratas*, em vez de em termos de células físicas. As células físicas, ou unidades endereçáveis individualmente, da maioria das memórias de computador contemporâneas tem o tamanho de um byte, que, por sua vez, tem oito bits. Esse tamanho é pequeno demais para a maioria das variáveis de programa. Uma célula abstrata de memória tem o tamanho exigido pela variável à qual está associada. Por exemplo, apesar de os valores de ponto flutuante poderem ocupar quatro bytes físicos em uma implementação de determinada linguagem, um valor de ponto flutuante é visto como se ocupasse uma única célula abstrata de memória. O valor de cada tipo não estruturado simples é considerado como ocupante de uma única célula abstrata. Daqui em diante, o termo *célula de memória* significará uma célula abstrata de memória.

O valor de uma variável é, algumas vezes, chamado de **lado direito** (*r-value*), porque é exigido quando o nome da variável aparece no lado direito de uma sentença de atribuição. Para se acessar o valor do lado *direito*, primeiro o valor do lado *esquerdo* deve ser determinado. Tais determinações nem sempre são simples. Por exemplo, regras de escopo podem complicar enormemente as coisas, conforme discutido na Seção 5.5.

5.4 O CONCEITO DE VINCULAÇÃO

Vinculação é a associação entre um atributo e uma entidade, como entre uma variável e seu tipo ou valor, ou entre uma operação e um símbolo. O momento no qual uma vinculação ocorre é chamado de **tempo de vinculação**. A vinculação e os tempos de vinculação são conceitos proeminentes na semântica das linguagens de programação. As vinculações podem ocorrer em tempo de projeto da linguagem, em tempo de implementação da linguagem, em tempo de compilação, em tempo de carga, em tempo de ligação ou em tempo de execução. Por exemplo, o símbolo de asterisco (*) é geralmente ligado à operação de multiplicação em tempo de projeto de uma linguagem. Um tipo de dados, como **`int`** em C, é vinculado a uma faixa de valores possíveis em tempo de implementação da linguagem. Em tempo de compilação, uma variável em um programa Java é vinculada a um tipo de dados em particular. Uma variável pode ser vinculada a uma célula de armazenamento quando o programa é carregado na memória. A mesma vinculação não acontece até o tempo de execução, em alguns casos, como as variáveis declaradas em métodos Java. Uma chamada a um subprograma de uma biblioteca é vinculada ao código do subprograma em tempo de ligação.

Considere a seguinte sentença de atribuição em C++:

```
count = count + 5;
```

Algumas das vinculações e seus tempos de vinculação para as partes dessa sentença são:

- O tipo de `count` é vinculado em tempo de compilação.
- O conjunto dos valores possíveis de `count` é vinculado em tempo de projeto do compilador.
- O significado do símbolo de operador `+` é vinculado em tempo de compilação, quando os tipos dos operandos tiverem sido determinados.
- A representação interna do literal `5` é vinculada ao tempo de projeto do compilador.
- O valor de `count` é vinculado em tempo de execução a essa sentença.

O entendimento completo dos tempos de vinculação para os atributos de entidades de programa é um pré-requisito para entender a semântica de uma linguagem de programação. Por exemplo, para entender o que um subprograma faz, deve-se entender como os valores reais em uma chamada são vinculados aos parâmetros formais em sua definição. Para determinar o valor atual de uma variável, pode ser necessário saber quando a variável foi vinculada ao armazenamento e a qual sentença (ou sentenças).

5.4.1 Vinculação de atributos a variáveis

Uma vinculação é **estática** se ocorre pela primeira vez antes do tempo de execução e permanece inalterada ao longo da execução do programa. Se a vinculação ocorre pela primeira vez durante o tempo de execução ou pode ser mudada ao longo do curso da execução do programa, é chamada de **dinâmica**. A vinculação física de uma variável a uma célula de armazenamento em um ambiente de memória virtual é complexa, porque a página ou o segmento do espaço de endereçamento no qual a célula reside pode ser movido para dentro ou para fora da memória muitas vezes, durante a execução do

programa. De certa forma, tais variáveis são vinculadas e desvinculadas repetidamente. Contudo, essas vinculações são mantidas pelo hardware do computador, e as alterações são invisíveis para o programa e para o usuário. Como elas não são importantes para a discussão, não vamos nos preocupar com essas vinculações de hardware. O ponto essencial é distinguir entre vinculações estáticas e dinâmicas.

5.4.2 Vinculações de tipos

Antes de uma variável poder ser referenciada em um programa, ela deve ser vinculada a um tipo de dados. Os dois aspectos importantes dessa vinculação são: como o tipo é especificado e quando a vinculação ocorre. Os tipos podem ser especificados estaticamente por alguma forma de declaração explícita ou implícita.

5.4.2.1 Vinculação de tipos estática

Uma **declaração explícita** é uma sentença em um programa que lista nomes de variáveis e especifica que elas são de certo tipo. Uma **declaração implícita** é uma forma de associar variáveis a tipos por meio de convenções padronizadas, em vez de por sentenças de declaração. Nesse caso, a primeira aparição de um nome de variável em um programa constitui sua declaração implícita. Tanto as declarações explícitas quanto implícitas criam vinculações estáticas a tipos.

A maioria das linguagens de programação amplamente usadas, que empregam exclusivamente vinculação estática a tipos e foram projetadas desde meados dos anos 1960, exige declarações explícitas de todas as variáveis (Visual Basic e ML são duas exceções).

A vinculação de tipo de variável implícita é feita pelo processador da linguagem, por um compilador ou um interpretador. Existem várias bases diferentes para as vinculações implícitas de tipo de variável. A mais simples delas é a convenção de atribuição de nomes. Nesse caso, o compilador ou o interpretador vincula uma variável a um tipo com base na forma sintática do nome da variável.

Apesar de serem uma pequena conveniência para os programadores, as declarações implícitas podem ser prejudiciais à confiabilidade, pois impedem o processo de compilação de detectar alguns erros de programação e de digitação.

Alguns dos problemas das declarações implícitas podem ser evitados obrigando-se os nomes para tipos específicos a começarem com determinados caracteres especiais. Por exemplo, em Perl, qualquer nome que começa com `$` é um escalar, o qual pode armazenar uma cadeia ou um valor numérico. Se um nome começa com `@`, é um vetor; se começa com `%`, é uma estrutura de dispersão (*hash*).[3] Isso cria diferentes espaços de nomes para diferentes variáveis de tipo. Nesse cenário, os nomes `@apple` e `%apple` não são relacionados, porque cada um forma um espaço de nomes diferente. Além disso, um leitor de um programa sempre sabe o tipo de uma variável ao ler seu nome.

Outras classes de declarações de tipo implícitas utilizam o contexto. Às vezes isso é chamado de **inferência de tipos**. No caso mais simples, o contexto é o tipo do valor atribuído à variável em uma sentença de declaração. Por exemplo, em C#, uma declaração **`var`** de uma variável deve incluir um valor inicial, cujo tipo é admitido como o da variável. Considere as declarações:

[3]Tanto vetores como dispersões são considerados tipos – ambos podem armazenar qualquer valor escalar em seus elementos.

```
var sum = 0;
var total = 0.0;
var name = "Fred";
```

Os tipos de `sum`, `total` e `name` são **`int`**, **`float`** e **`string`**, respectivamente. Lembre-se de que essas variáveis são tipadas estaticamente – seus tipos são fixos durante toda a vida da unidade na qual são declaradas.

O Visual Basic e as linguagens funcionais ML, Haskell, OCaml e F# também usam inferência de tipos. Nessas linguagens funcionais, o contexto do aparecimento de um nome é a base para determinar seu tipo. Essa espécie de inferência de tipos é discutida em detalhes no Capítulo 15.

5.4.2.2 Vinculação de tipos dinâmica

Com a vinculação de tipos dinâmica, o tipo de uma variável não é especificado por uma sentença de declaração, nem pode ser determinado pelo nome da variável. Em vez disso, a variável é vinculada a um tipo quando é atribuído um valor a ela em uma sentença de atribuição. Quando a sentença de atribuição é executada, a variável que recebe um valor atribuído é vinculada ao tipo do valor da expressão no lado direito da atribuição. Tal atribuição também pode vincular a variável a um endereço e a uma célula de memória, pois diferentes valores de tipo podem exigir diferentes quantidades de armazenamento. Qualquer variável pode receber qualquer valor de tipo. Além disso, o tipo de uma variável pode mudar qualquer número de vezes durante a execução do programa. É importante perceber que o tipo de uma variável, quando é vinculado dinamicamente, pode ser temporário.

Quando o tipo de uma variável é vinculado estaticamente, pode-se considerar que o nome dela é vinculado a um tipo, pois o tipo e o nome de uma variável são vinculados simultaneamente. No entanto, quando o tipo de uma variável é vinculado dinamicamente, pode-se considerar que seu nome é vinculado a um tipo apenas temporariamente. Na realidade, os nomes de variáveis nunca são vinculados a tipos. Os nomes podem ser vinculados a variáveis e estas podem ser vinculadas a tipos.

As linguagens nas quais os tipos são vinculados dinamicamente são drasticamente diferentes daquelas nas quais os tipos são vinculados estaticamente. A principal vantagem da vinculação dinâmica de variáveis a tipos é que ela oferece maior flexibilidade ao programador. Por exemplo, um programa para processar dados numéricos em uma linguagem que usa a vinculação de tipos dinâmica pode ser escrito como um programa genérico; ou seja, ele será capaz de tratar dados de quaisquer tipos numéricos. Qualquer tipo de dados informado será aceitável, porque a variável na qual os dados serão armazenados pode ser vinculada ao tipo correto quando eles forem atribuídos às variáveis após a entrada. Em contraste, graças à vinculação estática de tipos, não é possível escrever um programa em C para processar dados sem se conhecer o tipo desses dados.

Antes de meados dos anos 1990, a maioria das linguagens de programação comumente usadas empregava vinculação de tipos estática; as principais exceções eram algumas linguagens funcionais, como Lisp. Contudo, desde então houve uma mudança significativa para linguagens que usam vinculação dinâmica de tipos. Em Python, Ruby, JavaScript e PHP, a vinculação de tipos é dinâmica. Por exemplo, um *script* JavaScript pode conter a seguinte sentença:

```
list = [10.2, 3.5];
```

Independentemente do tipo anterior da variável chamada `list`, essa atribuição faz com que ela se torne o nome de um vetor unidimensional de tamanho 2. Se a sentença

```
list = 47;
```

seguisse a atribuição de exemplo anterior, `list` se tornaria o nome de uma variável escalar.

A opção de vinculação dinâmica de tipos foi incluída no C# 2010. Uma variável pode ser declarada para usar vinculação de tipos dinâmica, com inclusão da palavra reservada **`dynamic`** em sua declaração, como no exemplo a seguir:

```
dynamic  any;
```

Isso é semelhante, embora também diferente, a declarar `any` de forma a ser de tipo **`object`**. É semelhante no sentido de que `any` pode receber um valor de qualquer tipo, exatamente como se fosse declarada como **`object`**. É diferente no sentido de que não é útil para várias situações diferentes de interoperação; por exemplo, com linguagens tipadas dinamicamente, como IronPython e IronRuby (versões .NET de Python e Ruby, respectivamente). Contudo, é útil quando dados de tipo desconhecido entram em um programa, vindos de uma fonte externa. Membros de classe, propriedades, parâmetros de método, valores de retorno de método e variáveis locais, todos podem ser declarados com **`dynamic`**.

Em linguagens orientadas a objetos puras – por exemplo, Ruby –, todas as variáveis são referências e não possuem tipos; todos os dados são objetos e qualquer variável pode referenciar qualquer objeto. De certo modo, nessas linguagens as variáveis são todas do mesmo tipo – elas são referências. No entanto, ao contrário das referências em Java, que estão restritas a referenciar um tipo de valor específico, em Ruby as variáveis podem referenciar qualquer objeto.

Existem duas desvantagens da vinculação de tipos dinâmica. Primeiro, ela torna os programas menos confiáveis, porque a capacidade de detecção de erros do compilador é diminuída em relação a um compilador para uma linguagem com vinculações de tipo estáticas. A vinculação de tipos dinâmica permite atribuir valores de quaisquer tipos a quaisquer variáveis. Tipos incorretos de lados direitos de atribuições não são detectados como erros; em vez disso, o tipo do lado esquerdo é trocado pelo tipo incorreto. Por exemplo, suponha que em determinado programa JavaScript, `i` e `x` estivessem armazenando os nomes de variáveis numéricas escalares e `y` fosse o nome de um vetor. Além disso, suponha que o programa precise da sentença de atribuição

```
i = x;
```

mas, por causa de um erro de digitação, ele tem a seguinte sentença de atribuição

```
i = y;
```

Em JavaScript (ou em qualquer outra linguagem que use vinculação de tipos dinâmica), nenhum erro é detectado nessa sentença pelo interpretador – o tipo da variável chamada `i` simplesmente se transforma em um vetor. Mas os resultados seguintes de `i` esperam

que ele seja um escalar, e resultados corretos serão impossíveis. Em uma linguagem com vinculação de tipos estática, como Java, o compilador detectaria o erro na atribuição `i = y`, e o programa não seria executado.

Note que, até certo ponto, essa desvantagem também está presente em algumas linguagens que usam vinculação de tipos estática, como C e C++, as quais, em muitos casos, convertem automaticamente o tipo do lado direito de uma atribuição no tipo do lado esquerdo.

Talvez a principal desvantagem da vinculação de tipos dinâmica seja o custo. O custo de implementar a vinculação de atributos dinâmica é considerável, principalmente em tempo de execução. A verificação de tipos deve ser feita em tempo de execução. Além disso, cada variável deve ter um descritor em tempo de execução associado a ela, de forma a manter o tipo atual. O armazenamento usado para o valor de uma variável deve ser de tamanho variável, porque valores de tipos diferentes precisam de quantidades distintas de armazenamento.

Por fim, as linguagens que têm vinculação de tipos dinâmica são normalmente implementadas por meio de interpretadores puros, em vez de compiladores. Os computadores não contêm instruções cujos tipos dos operandos não são conhecidos em tempo de compilação. Logo, um compilador não pode construir instruções de máquina para a expressão A + B se os tipos A e B não são conhecidos em tempo de compilação. A interpretação pura tipicamente leva pelo menos 10 vezes mais tempo para executar um código de máquina equivalente. Evidentemente, se uma linguagem é implementada com um interpretador puro, o tempo para realizar a vinculação de tipos dinâmica é ocultado pelo tempo total da interpretação; assim, tal vinculação parece ser menos dispendiosa nesse ambiente. Por outro lado, as linguagens com vinculações de tipo estáticas raramente são implementadas pela interpretação pura, pois os programas nessas linguagens podem ser facilmente traduzidos para versões em código de máquina muito eficientes.

5.4.3 Vinculações de armazenamento e tempo de vida

O caráter fundamental de uma linguagem de programação imperativa é, em grande parte, determinado pelo projeto das vinculações de armazenamento para suas variáveis. Dessa forma, é importante ter um claro entendimento dessas vinculações.

A célula de memória à qual uma variável é vinculada deve ser obtida, de alguma forma, de um conjunto de células de memória disponíveis. Esse processo é chamado de **alocação**. **Liberação** é o processo de colocar uma célula de memória que foi desvinculada de uma variável de volta no conjunto de células de memória disponíveis.

O **tempo de vida** de uma variável é aquele durante o qual ela está vinculada a uma posição específica da memória. Então, o tempo de vida de uma variável começa quando ela é vinculada a uma célula específica e termina quando ela é desvinculada dessa célula. Para investigar as vinculações de armazenamento das variáveis, é conveniente separar as variáveis escalares (não estruturadas) em quatro categorias, de acordo com seus tempos de vida. As categorias são: estáticas, dinâmicas da pilha, dinâmicas do monte explícitas e dinâmicas do monte implícitas. Nas seções a seguir, discutimos as definições das quatro categorias, com seus propósitos, vantagens e desvantagens.

5.4.3.1 Variáveis estáticas

Variáveis estáticas são vinculadas a células de memória antes do início da execução de um programa e permanecem vinculadas a essas mesmas células até que a execução do programa termine. Variáveis vinculadas estaticamente têm diversas aplicações importantes em programação. Variáveis acessíveis globalmente são usadas ao longo da execução de um programa, o que implica tê-las vinculadas ao mesmo armazenamento durante essa execução. Algumas vezes é conveniente ter subprogramas sensíveis ao histórico. Tais subprogramas devem ter variáveis locais estáticas.

Uma vantagem das variáveis estáticas é a eficiência. Todo o endereçamento de variáveis estáticas pode ser direto;[4] outros tipos de variáveis frequentemente exigem endereçamento indireto, que é mais lento. Além disso, não há sobrecarga em tempo de execução para a alocação e a liberação de variáveis estáticas, apesar de esse tempo ser normalmente insignificante.

Uma desvantagem da vinculação estática para armazenamento é a redução da flexibilidade; em particular, uma linguagem que tem apenas variáveis estáticas não permite o uso de subprogramas recursivos. Outra desvantagem é o armazenamento não ser compartilhado entre variáveis. Por exemplo, suponha que um programa tem dois subprogramas que exigem vetores grandes. Suponha, também, que os dois subprogramas nunca estão ativos ao mesmo tempo. Se os vetores são estáticos, eles não podem compartilhar o mesmo armazenamento para seus vetores.

C e C++ permitem aos programadores incluírem o especificador **`static`** em uma definição de variável de uma função, fazendo as variáveis definidas serem estáticas. Note que, quando o modificador **`static`** aparece na declaração de uma variável em uma definição de classe em C++, Java e C#, isso também significa que se trata de uma variável de classe e não de uma variável de instância. As variáveis de classe são criadas estaticamente algum tempo antes de a classe ser instanciada pela primeira vez.

5.4.3.2 Variáveis dinâmicas da pilha

Variáveis dinâmicas da pilha são aquelas cujas vinculações de armazenamento são criadas quando suas sentenças de declaração são elaboradas, mas cujos tipos são vinculados estaticamente. A **elaboração** de tal declaração se refere à alocação do armazenamento e ao processo de vinculação indicado pela declaração, que ocorre quando a execução alcança o código no qual a declaração está anexada. Logo, a elaboração ocorre apenas em tempo de execução. Por exemplo, as declarações de variáveis que aparecem no início de um método Java são elaboradas quando o método é chamado, e as variáveis definidas por essas declarações são liberadas quando o método completa sua execução.

Como seu nome indica, as variáveis dinâmicas da pilha são alocadas a partir da pilha de tempo de execução.

Algumas linguagens – como C++ e Java – permitem que declarações de variáveis ocorram em qualquer lugar onde uma sentença poderia ocorrer. Em algumas implementações dessas linguagens, todas as variáveis dinâmicas da pilha declaradas em uma função ou em um método (não incluindo as declaradas em blocos aninhados) podem ser vinculadas ao armazenamento no início da execução da função ou do método, mesmo

[4] Em algumas implementações, as variáveis estáticas são endereçadas por meio de um registrador-base, o que torna o acesso a elas tão dispendioso quanto para variáveis alocadas na pilha.

que as declarações de algumas dessas variáveis não apareçam no início. Nesses casos, a variável se torna visível na declaração, mas a vinculação ao armazenamento (e a inicialização, se for especificada na declaração) ocorre quando a função ou o método inicia sua execução. O fato de a vinculação do armazenamento ocorrer antes de ele se tornar visível não afeta a semântica da linguagem.

As vantagens das variáveis dinâmicas da pilha são as seguintes: para serem úteis, ao menos na maioria dos casos, os subprogramas recursivos exigem armazenamento dinâmico local, de forma que cada cópia ativa do subprograma recursivo tenha sua própria versão das variáveis locais. Essas necessidades são convenientemente satisfeitas pelas variáveis dinâmicas da pilha. Mesmo na ausência da recursão, ter armazenamento local dinâmico na pilha para subprogramas tem méritos, porque todos os subprogramas compartilham o mesmo espaço de memória para suas variáveis locais.

As desvantagens das variáveis dinâmicas da pilha em relação às variáveis estáticas são a sobrecarga em tempo de execução da alocação e da liberação, acessos possivelmente mais lentos em função do endereçamento indireto necessário, e o fato de os subprogramas não poderem ser sensíveis ao histórico de execução. O tempo necessário para alocar e liberar variáveis dinâmicas da pilha não é significativo, porque todas as variáveis dinâmicas da pilha declaradas no início de um subprograma são alocadas e liberadas juntas, e não em operações separadas.

Em Java, C++ e C#, as variáveis definidas em métodos são, por padrão, dinâmicas da pilha.

Todos os atributos, exceto os de armazenamento, são estaticamente vinculados às variáveis escalares dinâmicas da pilha. Esse não é o caso para alguns tipos estruturados, conforme discutido no Capítulo 6. A implementação dos processos de alocação e liberação para variáveis dinâmicas da pilha é discutida no Capítulo 10.

5.4.3.3 Variáveis dinâmicas do monte explícitas

Variáveis dinâmicas do monte explícitas são células de memória não nomeadas (abstratas), alocadas e liberadas por instruções explícitas em tempo de execução escritas pelo programador. Essas variáveis, alocadas a partir do monte e liberadas para o monte, podem apenas ser referenciadas por ponteiros ou variáveis de referência. O monte (*heap*) é uma coleção de células de armazenamento altamente desorganizadas, devido à imprevisibilidade de seu uso. O ponteiro (ou a variável de referência) usado para acessar uma variável dinâmica do monte explícita é criado como qualquer outra variável escalar. Uma variável dinâmica do monte explícita é criada por meio de um operador (por exemplo, em C++) ou de uma chamada a um subprograma de sistema fornecido para esse propósito (por exemplo, em C).

Em C++, o operador de alocação, chamado **`new`**, usa um nome de tipo como operando. Quando executado, uma variável dinâmica do monte explícita do tipo do operando é criada e seu endereço é retornado. Como uma variável dinâmica do monte explícita é vinculada a um tipo em tempo de compilação, essa vinculação é estática. Entretanto, tais variáveis são vinculadas ao armazenamento no momento em que são criadas, durante o tempo de execução.

Além de um subprograma ou operador criar variáveis dinâmicas do monte explícitas, algumas linguagens incluem um subprograma ou operador para destruí-las explicitamente.

Como um exemplo de variáveis dinâmicas do monte explícitas, considere o segmento de código em C++:

```
int  *intnode;       // Cria um ponteiro
intnode =  new int; // Cria a variável dinâmica do monte
. . .
delete  intnode;     // Libera a variável dinâmica do monte
                         // para qual intnode aponta
```

Nesse exemplo, uma variável dinâmica do monte explícita do tipo **int** é criada pelo operador **new**. Essa variável pode então ser referenciada por meio do ponteiro `intnode`. Posteriormente, a variável é liberada pelo operador **delete**. C++ exige o operador de liberação explícita **delete**, porque a linguagem não usa recuperação implícita de armazenamento, como a coleta de lixo.

Em Java, todos os dados, exceto os escalares primitivos, são objetos. Objetos Java são dinâmicos do monte explícitos e acessados por meio de variáveis de referência. Java não tem uma maneira de destruir explicitamente uma variável dinâmica do monte; em vez disso, é usada a coleta de lixo implícita. A coleta de lixo é discutida no Capítulo 6.

C# tem tanto objetos dinâmicos do monte explícitos quanto dinâmicos da pilha, todos implicitamente liberados. Além disso, C# oferece suporte a ponteiros no estilo de C++. Tais ponteiros são usados para referenciar o monte, a pilha e mesmo variáveis estáticas e objetos. Esses ponteiros têm os mesmos perigos daqueles de C++, e os objetos que eles referenciam no monte não são liberados implicitamente. Ponteiros são incluídos em C# para permitir que os componentes C# interoperem com componentes C e C++. Para desestimular seu uso e também para tornar claro, para qualquer leitor do programa, que o código utiliza ponteiros, o cabeçalho de todo método que defina um ponteiro deve conter a palavra reservada **unsafe**.

Variáveis dinâmicas do monte explícitas são usadas para construir estruturas dinâmicas, como listas ligadas e árvores, que precisam crescer e/ou diminuir durante a execução. Tais estruturas podem ser construídas de maneira conveniente por meio de ponteiros ou referências e variáveis dinâmicas do monte explícitas.

As desvantagens das variáveis dinâmicas do monte explícitas são a dificuldade de usar ponteiros e variáveis de referência corretamente, o custo das referências às variáveis e a complexidade da implementação do gerenciamento de armazenamento exigido. Esse é essencialmente o problema do gerenciamento do monte, que é dispendioso e complicado. Métodos de implementação para variáveis dinâmicas do monte explícitas são amplamente discutidos no Capítulo 6.

5.4.3.4 Variáveis dinâmicas do monte implícitas

Variáveis dinâmicas do monte implícitas são vinculadas ao armazenamento no monte apenas quando são atribuídos valores a elas. De fato, todos os seus atributos são vinculados cada vez que elas recebem valores atribuídos. Por exemplo, considere a seguinte sentença de atribuição em JavaScript:

```
highs = [74, 84, 86, 90, 71];
```

Independentemente de a variável chamada `highs` ter sido usada anteriormente no programa ou de para que foi usada, ela agora é um vetor de cinco valores numéricos.

A vantagem de tais variáveis é que elas têm o mais alto grau de flexibilidade, permitindo a escrita de códigos altamente genéricos. Uma desvantagem das variáveis dinâmicas do monte implícitas é a sobrecarga em tempo de execução para manter todos os atributos dinâmicos, o que pode incluir tipos e faixas de índices de vetores, entre outros. Outra desvantagem é a perda de alguma detecção de erros pelo compilador, conforme discutido na Seção 5.4.2.2.

5.5 ESCOPO

Um dos fatores importantes para o entendimento das variáveis é o escopo. O **escopo** de uma variável é a faixa de sentenças nas quais ela é visível. Uma variável é **visível** em uma sentença se ela pode ser referenciada ou atribuída nessa sentença.

As regras de escopo de uma linguagem definem como determinada ocorrência de um nome é associada a uma variável ou, no caso de uma linguagem funcional, como um nome é associado a uma expressão. Em particular, as regras de escopo determinam como referências a variáveis declaradas fora do subprograma ou bloco em execução são associadas às suas declarações e, logo, aos seus atributos (os blocos são discutidos na Seção 5.5.2). Portanto, um claro entendimento dessas regras para uma linguagem é essencial para a habilidade de escrever ou ler programas nela.

Uma variável é **local** a uma unidade ou a um bloco de programa se for declarada lá. As variáveis não locais de uma unidade ou de um bloco de programa são aquelas visíveis dentro da unidade ou do bloco de programa, mas não declaradas nessa unidade ou nesse bloco. Variáveis globais são uma categoria especial de variáveis não locais, as quais são discutidas na Seção 5.5.4.

Questões de escopo para classes, pacotes e espaços de nomes são discutidas no Capítulo 11.

5.5.1 Escopo estático

ALGOL 60 introduziu o método de vincular nomes a variáveis não locais, chamado de **escopo estático**,[5] copiado por muitas linguagens imperativas subsequentes e também por muitas linguagens não imperativas. O escopo estático é assim chamado porque o escopo de uma variável pode ser determinado estaticamente – ou seja, antes da execução. Isso permite a um leitor de programas humano (e a um compilador) determinar o tipo de cada variável no programa simplesmente examinando seu código-fonte.

Existem duas categorias de linguagens de escopo estático: aquelas nas quais os subprogramas podem ser aninhados, as quais criam escopos estáticos aninhados, e aquelas nas quais os subprogramas não podem ser aninhados. Na última categoria, os escopos estáticos também são criados para subprogramas, mas os aninhados são criados apenas por definições de classes aninhadas ou de blocos aninhados.

Ada, JavaScript, Common Lisp, Scheme, Fortran 2003+, F# e Python permitem subprogramas aninhados, mas as linguagens baseadas em C não.

Nossa discussão sobre o uso de escopo estático nesta seção se concentra nas linguagens que permitem subprogramas aninhados. Inicialmente, supomos que *todos* os

[5] O escopo estático às vezes é chamado de *escopo léxico*.

escopos são associados a unidades de programa e que todas as variáveis não locais referenciadas são declaradas em outras unidades de programa.[6] Neste capítulo, supomos que o uso de escopos é o único método de acessar variáveis não locais nas linguagens em discussão. Isso não é verdade para todas as linguagens. Não é verdade nem mesmo para todas as linguagens que usam escopo estático, mas tal premissa simplifica a discussão aqui.

Quando o leitor de um programa encontra uma referência a uma variável, os atributos dessa variável podem ser determinados por meio da descoberta da sentença na qual ela está declarada (explícita ou implicitamente). Em linguagens de escopo estático com subprogramas, esse processo pode ser visto da forma a seguir. Suponha que uma referência é feita a uma variável `x` em um subprograma `sub1`. A declaração correta é encontrada primeiro pela busca das declarações do subprograma `sub1`. Se nenhuma declaração for encontrada para a variável lá, a busca continua nas declarações do subprograma que declarou o subprograma `sub1`, chamado de **pai estático**. Se uma declaração de `x` não for encontrada lá, a busca continua para a próxima unidade maior que envolve o subprograma em que está sendo feita a busca (a unidade que declarou o pai de `sub1`), e assim por diante, até que seja encontrada uma declaração para `x` ou a maior unidade de declaração tenha sido pesquisada sem sucesso. Nesse caso, é informado um erro de variável não declarada. O pai estático do subprograma `sub1`, seu pai estático, e assim por diante até (e incluindo) o maior subprograma que envolve os demais, são chamados de **ancestrais estáticos** de `sub1`. Técnicas de implementação reais para escopo estático, discutidas no Capítulo 10, normalmente são muito mais eficientes que o processo que acabamos de descrever.

Considere a função JavaScript a seguir, `big`, na qual as duas funções `sub1` e `sub2` são aninhadas:

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```

De acordo com o escopo estático, a referência à variável `x` em `sub2` é para o `x` declarado no procedimento `big`. Isso é verdade porque a busca por `x` começa no procedimento no qual a referência ocorre, `sub2`, mas nenhuma declaração para `x` é encontrada lá. A busca continua no pai estático de `sub2`, `big`, onde a declaração de `x` é encontrada. O `x` declarado em `sub1` é ignorado porque não está nos ancestrais estáticos de `sub2`.

[6] As variáveis não locais não definidas em outras unidades de programa são discutidas na Seção 5.5.4.

Em linguagens que usam escopo estático, independentemente de ser permitido o uso de subprogramas aninhados ou não, algumas declarações de variáveis podem ser ocultadas de outros segmentos de código. Por exemplo, considere mais uma vez a função JavaScript `big`. A variável `x` é declarada tanto em `big` quanto em `sub1`, aninhado dentro de `big`. Dentro de `sub1`, toda referência simples a `x` é para o `x` local. Portanto, o `x` mais externo é ocultado de `sub1`.

5.5.2 Blocos

Muitas linguagens permitem que novos escopos estáticos sejam definidos no meio do código executável. Esse poderoso conceito, introduzido em ALGOL 60, permite a uma seção de código ter suas próprias variáveis locais, cujo escopo é minimizado. Tais variáveis são dinâmicas da pilha, de forma que seu armazenamento é alocado quando a seção é alcançada e liberado quando a seção é abandonada. Essa seção de código é denominada **bloco**. Os blocos dão origem à frase **linguagem estruturada por blocos**.

As linguagens baseadas em C permitem que quaisquer sentenças compostas (uma sequência de sentenças envoltas em chaves correspondentes – { }) tenham declarações e, dessa forma, definam um novo escopo. Tais sentenças compostas são denominadas blocos. Por exemplo, se `list` fosse um vetor de inteiros, alguém poderia escrever o seguinte:

```
if (list[i] < list[j]) {
  int temp;
  temp = list[i];
  list[i] = list[j];
  list[j] = temp;
}
```

Os escopos criados por blocos, que podem ser aninhados em blocos maiores, são tratados exatamente como aqueles criados por subprogramas. Referências a variáveis em um bloco e que não estão declaradas lá são conectadas às declarações por meio da busca pelos escopos que o envolvem (blocos ou subprogramas), por ordem de tamanho (do menor para o maior).

Considere o esqueleto de função em C:

```
void sub() {
  int count;
  . . .
  while (. . .) {
    int count;
    count++;
    . . .
  }
  . . .
}
```

A referência a `count` no laço de repetição **`while`** é para o `count` local do laço. Nesse caso, o `count` de `sub` é ocultado do código que está dentro do laço **`while`**. Em geral, a declaração de uma variável efetivamente oculta qualquer declaração de uma variável

com o mesmo nome em um escopo envolvente maior.[7] Note que esse código é válido em C e C++, mas inválido em Java e C#. Os projetistas de Java e C# acreditavam que o reúso de nomes em blocos aninhados era muito propenso a erros para ser permitido.

Embora JavaScript utilize escopo estático para suas funções aninhadas, blocos que não são função não podem ser definidos na linguagem.

A maioria das linguagens de programação funcionais inclui uma construção relacionada aos blocos das linguagens imperativas, normalmente chamada *let*. Essas construções têm duas partes, a primeira das quais para vincular nomes a valores, normalmente especificados como expressões. A segunda parte é uma expressão que utiliza os nomes definidos na primeira parte. Programas em linguagens funcionais são compostos de expressões, em vez de sentenças. Portanto, a parte final de uma construção `let` é uma expressão, não uma sentença. Em Scheme, uma construção `let` é uma chamada à função `LET` da seguinte forma:

```
(LET (
  (nome1 expressão1)
  . . .
  (nomen expressãon))
  expressão
)
```

A semântica da chamada a `LET` é a seguinte: as primeiras *n* expressões são avaliadas e os valores são atribuídos aos nomes associados. Então, a última expressão é avaliada e o valor de retorno de `LET` é esse valor. Isso difere de um bloco em uma linguagem imperativa, pois os nomes são os de valores; eles não são variáveis no sentido imperativo. Uma vez definidos, não podem ser alterados. Contudo, eles são como variáveis locais em um bloco de uma linguagem imperativa, pois seu escopo é local à chamada de `LET`. Considere a seguinte chamada a `LET`:

```
(LET (
  (top (+ a b))
  (bottom (- c d)))
  (/ top bottom)
)
```

Essa chamada calcula e retorna o valor da expressão `(a + b) / (c - d)`.

Na ML, a forma de uma construção **`let`** é a seguinte:

```
let
  val  nome1 = expressão1
  . . .
  val  nomen = expressãon
in
  expressão
end;
```

[7] Conforme discutido na Seção 5.5.4, em C++, tais variáveis globais ocultas podem ser acessadas no escopo interno com o operador de escopo (::).

Cada sentença **val** vincula um nome a uma expressão. Como em Scheme, os nomes na primeira parte são como as constantes nomeadas das linguagens imperativas; uma vez definidos, não podem ser alterados.[8] Considere a seguinte função **let**:

```
let
  val  top = a + b
  val bottom = c - d
in
  top / bottom
end;
```

A forma geral de uma construção **let** em F# é a seguinte:

let lado_esquerdo = expressão

O elemento lado_esquerdo de **let** pode ser um nome ou um padrão de tupla (uma sequência de nomes separados por vírgulas).

O escopo de um nome definido com **let** dentro de uma definição de função abarca desde o final da definição da expressão até o final da função. O escopo de **let** pode ser limitado pela endentação do código seguinte, a qual cria um escopo local. Embora qualquer endentação funcione, a convenção é uma endentação de quatro espaços. Considere o código a seguir:

```
let n1 =
    let  n2 = 7
    let  n3 = n2 + 3
    n3;;
let n4 = n3 + n1;;
```

O escopo de `n1` se estende por todo o código. No entanto, o escopo de `n2` e `n3` termina quando a endentação acaba. Portanto, o uso de `n3` no último **let** causa um erro. A última linha do escopo de **let** `n1` é o valor vinculado a `n1`; ele poderia ser qualquer expressão.

O Capítulo 15 contém mais detalhes sobre as construções `let` em Scheme, ML, Haskell e F#.

5.5.3 Ordem de declaração

Em C89, como em algumas outras linguagens, todas as declarações de dados em uma função, exceto aquelas em blocos aninhados, devem aparecer no início da função. Entretanto, algumas linguagens – como C99, C++, Java, JavaScript e C# – permitem que as declarações de variáveis apareçam em qualquer lugar onde uma sentença poderia aparecer em uma unidade de programa. Declarações podem criar escopos não associados com sentenças compostas ou subprogramas. Por exemplo, em C99, C++ e Java, o escopo de todas as variáveis locais abarca desde suas declarações até o final dos blocos nos quais essas declarações aparecem.

[8] No Capítulo 15, vamos ver que eles podem ser redefinidos, mas que, na verdade, o processo gera um novo nome.

Em C#, o escopo de quaisquer variáveis declaradas em um bloco é o bloco inteiro, independentemente da posição da declaração, desde que ele não seja aninhado. O mesmo é válido para os métodos. Lembre-se de que C# não permite que a declaração de uma variável em um bloco aninhado tenha o mesmo nome de uma variável em um escopo que faça o aninhamento. Por exemplo, considere o código C# a seguir:

```
{int x;
   ...
   {int x;  // Inválido
   ...
   }
   ...
}
```

Como o escopo de uma declaração é o bloco inteiro, a seguinte declaração aninhada de `x` também é inválida:

```
{
   {int  x;  // Inválido
   ...
   }
   int  x;
}
```

Observe que C# ainda exige todas as variáveis declaradas antes de serem usadas. Logo, independentemente de o escopo de uma variável se estender da declaração até o topo do bloco ou subprograma no qual ela aparece, a variável ainda não pode ser usada acima de sua declaração.

Em JavaScript, variáveis locais podem ser declaradas em qualquer lugar em uma função, mas o escopo de uma variável assim é sempre a função inteira. Se for usada antes de sua declaração na função, essa variável terá o valor `undefined`. A referência não é inválida.

As sentenças **`for`** de C++, Java e C# permitem a definição de variáveis em suas expressões de inicialização. Nas primeiras versões de C++, o escopo de tal variável era de sua definição até o final do menor bloco que envolvia o **`for`**. Na versão padrão, entretanto, o escopo é restrito à construção **`for`**, como é o caso de Java e C#. Considere o esqueleto de método:

```
void fun() {
   . . .
   for (int  count = 0; count < 10; count++){
      . . .
   }
   . . .
}
```

Em versões posteriores de C++, como em Java e C#, o escopo de `count` vai da sentença **`for`** até o final de seu corpo (a chave de fechamento).

5.5.4 Escopo global

Algumas linguagens, incluindo C, C++, PHP, JavaScript e Python, permitem uma estrutura de programa que é uma sequência de definição de funções, nas quais as definições de variáveis podem aparecer fora das funções. Definições fora de funções em um arquivo criam variáveis globais, potencialmente visíveis para essas funções.

C e C++ têm tanto declarações quanto definições de dados globais. Declarações especificam tipos e outros atributos, mas não causam a alocação de armazenamento. As definições especificam atributos *e* causam a alocação de armazenamento. Para um nome global específico, um programa em C pode ter qualquer número de declarações compatíveis, mas apenas uma definição.

Uma declaração de uma variável fora das definições de funções especifica que ela é definida em um arquivo diferente. Uma variável global em C é implicitamente visível em todas as funções subsequentes no arquivo, exceto naquelas que incluem uma declaração de uma variável local com o mesmo nome. Uma variável global definida após uma função pode se tornar visível na função sendo declarada como externa, como no seguinte:

```
extern int  sum;
```

Em C99, as definições de variáveis globais normalmente têm valores iniciais. As declarações de variáveis globais nunca têm valores iniciais. Se a declaração estiver fora das definições de função, ela não precisa incluir o qualificador **`extern`**.

Essa ideia de declarações e definições é usada também para funções em C e C++, em que os protótipos declaram nomes e interfaces de funções, mas não fornecem seu código. As definições de funções, em contrapartida, são completas.

Em C++, uma variável global oculta por uma local com o mesmo nome pode ser acessada por meio do operador de escopo (::). Por exemplo, se `x` é uma variável global que está oculta em uma função por uma variável local chamada `x`, a variável global pode ser referenciada como `::x`.

Sentenças em PHP podem ser interpoladas com definições de funções. As variáveis em PHP são implicitamente declaradas quando aparecem em sentenças. Qualquer variável implicitamente declarada fora de qualquer função é global; variáveis implicitamente declaradas em funções são variáveis locais. O escopo das variáveis globais se estende desde suas declarações até o fim do programa, mas ignora quaisquer definições de funções subsequentes. Logo, as variáveis globais não são implicitamente visíveis em nenhuma função. Elas podem se tornar visíveis em funções em seu escopo de duas formas: (1) se a função contém uma variável local com o mesmo nome de uma global, a global pode ser acessada por meio do vetor `$GLOBALS`, usando-se o nome da global como índice do vetor, e (2) se na função não houver nenhuma variável local com o mesmo nome da global, a global poderá se tornar visível por meio de sua inclusão em uma sentença de declaração `global`. Considere o seguinte exemplo:

```
$day = "Monday";
$month = "January";

function calendar() {
  $day = "Tuesday";
```

```
  global $month;
  print "local day is $day  ";
  $gday = $GLOBALS['day'];
  print "global day is $gday <br \>";
  print "global month is $month  ";
}

calendar();
```

A interpretação desse código produz:

```
local day is Tuesday
global day is Monday
global month is January
```

As variáveis globais de JavaScript são muito parecidas com as de PHP, exceto que não há como acessar uma variável global em uma função que tenha declarado uma variável local com o mesmo nome.

As regras de visibilidade para variáveis globais em Python não são usuais. As variáveis não são declaradas normalmente, como em PHP. Elas são implicitamente declaradas quando aparecem como alvos de sentenças de atribuição. Uma variável global pode ser referenciada em uma função, mas pode ter valores atribuídos a ela apenas se tiver sido declarada como global na função. Considere os exemplos a seguir:

```
day = "Monday"

def tester():
print "The global day is:", day

tester()
```

Como as variáveis globais podem ser referenciadas diretamente nas funções, a saída desse *script* é:

```
The global day is: Monday
```

O *script* a seguir tenta atribuir um novo valor à variável global `day`:

```
day = "Monday"

def  tester():
  print "The global day is:", day
  day = "Tuesday"
  print "The new value of day is:", day

tester()
```

O *script* cria uma mensagem de erro do tipo `UnboundLocalError`, porque a atribuição a `day` na segunda linha do corpo da função torna `day` uma variável local – o que faz a referência a `day` na primeira linha do corpo da função se tornar uma referência inválida para a variável local.

A atribuição a `day` pode ser para a variável global se `day` for declarada como global no início da função, o que impede que a atribuição a `day` crie uma variável local. Isso é mostrado no *script* a seguir:

```
day = "Monday"

def tester():
  global  day
  print  "The global day is:", day
  day = "Tuesday"
  print  "The new value of day is:", day

tester()
```

A saída desse *script* é:

```
The global day is: Monday
The new value of day is: Tuesday
```

Em Python, as funções podem ser aninhadas. As variáveis definidas nas funções que fazem o aninhamento são acessíveis em uma função aninhada por meio do escopo estático, mas essas variáveis devem ser declaradas como **`nonlocal`** na função aninhada.[9] Um esqueleto de programa de exemplo na Seção 5.7 ilustra os acessos a variáveis não locais.

Em F#, todos os nomes definidos fora de definições de função são globais. Seu escopo se estende desde suas definições até o final do arquivo.

A ordem de declaração e as variáveis globais também são problemas que aparecem na declaração e nos membros de classes em linguagens orientadas a objetos. Isso é discutido no Capítulo 12.

5.5.5 Avaliação do escopo estático

O escopo estático fornece um método de acesso a não locais que funciona bem em muitas situações. Entretanto, isso não ocorre sem problemas. Primeiro, na maioria dos casos ele permite mais acesso que o necessário às variáveis e aos subprogramas. É simplesmente uma ferramenta rudimentar demais para especificar concisamente tais restrições. Segundo, e talvez mais importante, é um problema relacionado à evolução de programas. Sistemas de software são altamente dinâmicos – programas usados regularmente mudam com frequência. Essas mudanças normalmente resultam em reestruturação, destruindo a estrutura inicial que restringia o acesso às variáveis e aos subprogramas em uma linguagem de escopo estático. Para evitar a complexidade de manter essas restrições de acesso, os desenvolvedores normalmente descartam a estrutura quando ela começa a atrapalhar. Logo, tentar contornar as restrições do escopo estático pode levar a projetos de programas que mantêm pouca semelhança com seu original, mesmo em áreas do programa nas quais não foram feitas mudanças. Os projetistas são encorajados a usar muito mais variáveis globais que o necessário. Todos os subprogramas podem acabar aninhados no mesmo nível no programa principal,

[9] A palavra reservada **`nonlocal`** foi introduzida na Python 3.

usando globais, em vez de níveis de aninhamento mais profundos.[10] Além disso, o projeto final pode ser artificial e de difícil manipulação, e pode não refletir o projeto conceitual subjacente. Esses e outros defeitos do escopo estático são discutidos em detalhes em Clarke, Wileden e Wolf (1980). Uma alternativa ao uso do escopo estático para controlar o acesso às variáveis e aos subprogramas é a utilização de construções de encapsulamento, incluídas em muitas das novas linguagens. As construções de encapsulamento são discutidas no Capítulo 11.

5.5.6 Escopo dinâmico

O escopo de variáveis em APL, SNOBOL4 e nas primeiras versões de Lisp era dinâmico. Perl e Common Lisp também permitem que as variáveis sejam declaradas com escopo dinâmico, apesar de o mecanismo de escopo padrão dessas linguagens ser estático. O **escopo dinâmico** é baseado na sequência de chamadas de subprogramas, não em seu relacionamento espacial uns com os outros. Logo, o escopo pode ser determinado apenas em tempo de execução.

Considere mais uma vez a função `big` da Seção 5.5.1, a qual reproduzimos aqui, com exceção das chamadas de função:

```
function big() {
  function sub1() {
    var x = 7;
  }
  function sub2() {
    var y = x;
    var z = 3;
  }
  var x = 3;
}
```

Assuma que as regras de escopo dinâmico se aplicam a referências não locais. O significado do identificador `x` referenciado em `sub2` é dinâmico – ele não pode ser determinado em tempo de compilação. Ele pode referenciar qualquer uma das declarações de `x`, dependendo da sequência de chamadas.

Uma maneira pela qual o significado correto de `x` pode ser determinado durante a execução é iniciar a busca com as variáveis locais. Essa também é a maneira pela qual o processo começa no escopo estático, mas é aqui que a similaridade entre os dois tipos de escopo termina. Quando a busca por declarações locais falha, as declarações do pai dinâmico, o procedimento que o chamou, são procuradas. Se uma declaração para `x` não é encontrada lá, a busca continua no pai dinâmico dessa função, e assim por diante, até que seja encontrada uma declaração de `x`. Se nenhuma for encontrada em nenhum ancestral dinâmico, ocorre um erro em tempo de execução.

Considere as duas sequências de chamadas diferentes para `sub2` no exemplo anterior. Primeiro, `big` chama `sub1`, que chama `sub2`. Nesse caso, a busca continua a partir do procedimento local, `sub2`, para seu chamador, `sub1`, onde uma declaração de `x` é

[10] Parece a estrutura de um programa em C, não é?

encontrada. Logo, a referência a `x` em `sub2`, nesse caso, é para o `x` declarado em `sub1`. A seguir, `sub2` é chamado diretamente por `big`. Nesse caso, o pai dinâmico de `sub2` é `big`, e a referência é para o `x` declarado em `big`.

Note que, se o escopo estático fosse usado em qualquer uma das sequências de chamadas discutidas, a referência a `x` em `sub2` seria o `x` de `big`.

O escopo dinâmico em Perl não é usual – na verdade, ele não é exatamente como discutimos nesta seção, apesar de a semântica ser geralmente aquela do escopo dinâmico tradicional (veja o Exercício de Programação 1).

5.5.7 Avaliação do escopo dinâmico

O efeito do escopo dinâmico na programação é profundo. Quando o escopo dinâmico é usado, os atributos corretos das variáveis não locais visíveis a uma sentença de um programa não podem ser determinados estaticamente. Além disso, uma referência ao nome de tal variável nem sempre é para a mesma variável. Uma sentença em um subprograma que contém uma referência para uma variável não local pode se referir a diferentes variáveis não locais durante diferentes execuções do subprograma. Diversos tipos de problemas de programação aparecem devido ao escopo dinâmico.

Primeiro, durante o período de tempo iniciado quando um subprograma começa sua execução e terminado quando a execução é finalizada, as variáveis locais do subprograma são todas visíveis para qualquer outro subprograma que esteja sendo executado, independentemente de sua proximidade textual ou de como a execução chegou ao subprograma que está executando atualmente. Não existe uma forma de proteger as variáveis locais dessa acessibilidade. Os subprogramas são *sempre* executados no ambiente de todos os subprogramas chamados anteriormente e que ainda não completaram suas execuções. Assim, o escopo dinâmico resulta em programas menos confiáveis do que aqueles que usam escopo estático.

Um segundo problema do escopo dinâmico é a impossibilidade de verificar os tipos das referências a não locais estaticamente. Esse problema resulta da impossibilidade de encontrar estaticamente a declaração para uma variável referenciada como não local.

O escopo dinâmico também torna os programas muito mais difíceis de serem lidos, porque a sequência de chamadas de subprogramas deve ser conhecida para se determinar o significado das referências a variáveis não locais. Essa tarefa é praticamente impossível para um leitor humano.

Por fim, o acesso a variáveis não locais em linguagens de escopo dinâmico demora muito mais do que o acesso a não locais, quando o escopo estático é usado. A razão disso é explicada no Capítulo 10.

Por outro lado, o escopo dinâmico tem seus méritos. Em muitos casos, os parâmetros passados de um subprograma para outro são variáveis definidas no chamador. Nenhuma delas precisa ser passada em uma linguagem com escopo dinâmico, porque são implicitamente visíveis no subprograma chamado.

Não é difícil entender por que o escopo dinâmico não é tão usado quanto o escopo estático. Os programas em linguagens com escopo dinâmico são mais fáceis de ler, mais confiáveis e executam mais rapidamente do que programas equivalentes em linguagens com escopo dinâmico. Por essas razões, o escopo dinâmico foi substituído pelo escopo

estático na maioria dos dialetos atuais de Lisp. O Capítulo 10 discute métodos de implementação tanto para escopo estático quanto para escopo dinâmico.

5.6 ESCOPO E TEMPO DE VIDA

Algumas vezes, o escopo e o tempo de vida de uma variável parecem ser relacionados. Por exemplo, considere uma variável declarada em um método Java que não contém nenhuma chamada a método. O escopo dessa variável se estende desde sua declaração até o final do método. O tempo de vida dessa variável é o período de tempo que começa com a entrada no método e termina quando a execução do método chega ao final. Apesar de o escopo e o tempo de vida da variável serem diferentes, devido ao escopo estático ser um conceito textual, ou espacial, enquanto o tempo de vida é um conceito temporal, eles ao menos parecem ser relacionados nesse caso.

Esse relacionamento aparente entre escopo e tempo de vida não se mantém em outras situações. Em C e C++, por exemplo, uma variável declarada em uma função por meio do especificador **`static`** é estaticamente vinculada ao escopo dessa função e ao armazenamento. Logo, seu escopo é estático e local à função, mas seu tempo de vida se estende pela execução completa do programa do qual faz parte.

O escopo e o tempo de vida também não são relacionados quando estão envolvidas chamadas a subprogramas. Considere as funções C++ de exemplo:

```
void printheader() {
  . . .
 }  /* fim de printheader */
void compute() {
  int sum;
  . . .
  printheader();
 }  /* fim de compute */
```

O escopo da variável `sum` é completamente contido na função `compute`. Ele não se estende até o corpo da função `printheader`, apesar de `printheader` executar no meio da execução de `compute`. Entretanto, o tempo de vida de `sum` se estende por todo o período em que `printheader` é executada. Qualquer que seja a posição de armazenamento à qual `sum` está vinculada antes da chamada a `printheader`, esse vínculo continuará durante e após a execução de `printheader`.

5.7 AMBIENTES DE REFERENCIAMENTO

O **ambiente de referenciamento** de uma sentença é a coleção de todas as variáveis visíveis nela. O ambiente de referenciamento de uma sentença em uma linguagem de escopo estático é composto pelas variáveis declaradas em seu escopo local mais a coleção de todas as variáveis de seus escopos ancestrais visíveis. Em tais linguagens, o ambiente de referenciamento de uma sentença é necessário enquanto a sentença é compilada, de forma que código e estruturas necessárias possam ser criados para permitir referências às variáveis de outros escopos durante o tempo de execução. O Capítulo 10 discute téc-

nicas para implementar referências a variáveis não locais, tanto em linguagens de escopo estático quanto de escopo dinâmico.

Em Python, escopos podem ser criados por definição de função. O ambiente de referenciamento de uma sentença inclui as variáveis locais e todas as variáveis declaradas em funções nas quais a sentença está aninhada (excluindo variáveis em escopos não locais que estejam ocultas por declarações em funções mais próximas). Cada definição de função cria um escopo e, dessa forma, um novo ambiente. Considere o seguinte esqueleto de programa em Python:

```
g = 3; # Uma global

def sub1():
    a = 5; # Cria uma local
    b = 7; # Cria outra local
    . . . <------------------------------ 1
  def  sub2():
  global g;  # A global g agora pode ser atribuída aqui
    c = 9;  # Cria uma nova local
    . . . <------------------------------ 2
    def  sub3():
      nonlocal  c:  # Torna c não local visível aqui
      g = 11;  # Cria uma nova local
      . . . <------------------------------ 3
```

Os ambientes de referenciamento dos pontos de programa indicados são:

Ponto	*Ambiente de referenciamento*
1	local `a` e `b` (de `sub1`), global `g` para referência, mas não para atribuição
2	local `c` (de `sub2`), global `g` tanto para referência como para atribuição
3	não local `c` (de `sub2`), local `g` (de `sub3`)

Agora, considere as declarações de variáveis desse esqueleto de programa. Primeiro, note que, apesar de o escopo de `sub1` estar em um nível mais alto do que o de `sub3` (ele está menos profundamente aninhado), o escopo de `sub1` não é um ancestral estático de `sub3`; portanto, `sub3` não tem acesso às variáveis declaradas em `sub1`. Existe uma boa razão para isso. As variáveis declaradas em `sub1` são dinâmicas da pilha, então não estão vinculadas ao armazenamento se `sub1` não estiver em execução. Como `sub3` pode estar em execução quando `sub1` não estiver, não pode ser permitido a ela acessar variáveis em `sub1`, que não estará necessariamente vinculada ao armazenamento durante a execução de `sub3`.

Um subprograma está **ativo** se sua execução já tiver começado, mas ainda não terminado. O ambiente de referenciamento de uma sentença em uma linguagem de escopo dinâmico é composto pelas variáveis declaradas localmente, mais as variáveis de todos os subprogramas ativos. Mais uma vez, algumas variáveis em subprogramas ativos podem ser ocultadas do ambiente de referenciamento. Ativações de subprogramas recentes podem ter declarações para variáveis que ocultam variáveis com o mesmo nome em ativações anteriores de subprogramas.

Considere o seguinte programa de exemplo. Suponha que as únicas chamadas a funções sejam: `main` chama `sub2`, que chama `sub1`.

```
void sub1() {
  int a, b;
  . . . <------------ 1
}  /* fim de sub1 */
void sub2() {
  int b, c;
  . . . . <------------ 2
  sub1();
}  /* fim de sub2 */
void main() {
  int c, d;
  . . . <------------ 3
  sub2();
}  /* fim de main */
```

Os ambientes de referenciamento dos pontos de programa indicados são:

Ponto	*Ambiente de referenciamento*
1	`a` e `b` de `sub1`, `c` de `sub2`, `d` de `main`, (`c` de `main` e `b` de `sub2` estão ocultas)
2	`b` e `c` de `sub2`, `d` de `main`, (`c` de `main` está oculta)
3	`c` e `d` de `main`

5.8 CONSTANTES NOMEADAS

Uma **constante nomeada** é uma variável vinculada a um valor apenas uma vez. Constantes nomeadas são úteis para auxiliar a legibilidade e a confiabilidade dos programas. A legibilidade pode ser melhorada, por exemplo, ao ser usado o nome `pi` em vez da constante `3.14159265`.

Outro uso importante de constantes nomeadas é na parametrização de um programa. Por exemplo, considere um que processa valores de dados um número fixo de vezes, digamos, 100. Tal programa normalmente usa a constante `100` em diversos locais para declarar as faixas de índices de vetores e para controlar os limites dos laços de repetição. Considere o seguinte segmento do esqueleto de um programa Java:

```
void example() {
  int[] intList = new int[100];
  String[] strList =  new  String[100];
  . . .
  for  (index = 0; index < 100; index++) {
    . . .
  }
  . . .
  for  (index = 0; index < 100; index++) {
    . . .
  }
  . . .
  average = sum / 100;
```

```
  . . .
}
```

Quando esse programa for modificado para lidar com um número diferente de valores de dados, todas as ocorrências de `100` devem ser encontradas e modificadas. Em um programa extenso, isso pode ser tedioso e propenso a erros. Um método mais fácil e confiável é usar uma constante nomeada como um parâmetro de programa, como segue:

```
void example() {
  final int  len = 100;
  int[] intList =  new int[len];
  String[] strList =  new  String[len];
  . . .
  for (index = 0; index < len; index++) {
    . . .
  }
  . . .
  for (index = 0; index < len; index++) {
    . . .
  }
  . . .
  average = sum / len;
  . . .
}
```

Agora, quando o tamanho precisar ser alterado, apenas uma linha deverá ser modificada (a variável `len`), independentemente do número de vezes em que ela é usada no programa. Esse é outro exemplo das vantagens da abstração. O nome `len` é uma abstração para o número de elementos em alguns vetores e para o número de iterações em alguns laços de repetição. Isso ilustra como constantes nomeadas podem auxiliar na facilidade de modificação.

C++ também permite a vinculação dinâmica de valores a constantes nomeadas. Isso permite que expressões contendo variáveis sejam atribuídas às constantes nas declarações. Por exemplo, a sentença C++

```
const int  result = 2 * width + 1;
```

declara `result` como uma constante nomeada do tipo inteiro, cujo valor é informado como o da expressão `2 * width + 1`, onde o valor da variável `width` deve ser visível quando `result` for alocado e vinculado ao valor da expressão.

Java também permite a vinculação dinâmica de valores a constantes nomeadas. Em Java, constantes nomeadas são definidas com a palavra reservada **`final`** (como no exemplo anterior). O valor inicial pode ser dado na sentença de declaração ou em uma sentença de atribuição subsequente. O valor atribuído pode ser especificado com qualquer expressão.

C# tem dois tipos de constantes nomeadas: definidas com **`const`** e definidas com **`readonly`**. As constantes nomeadas **`const`**, implicitamente **`static`**, são estaticamente vinculadas a valores; são vinculadas aos valores em tempo de compilação; ou seja, esses valores podem ser especificados apenas com literais ou outros membros **`const`**. As constantes nomeadas **`readonly`**, vinculadas a valores dinamicamente, po-

dem ser atribuídas na declaração ou com um construtor estático.[11] Então, se um programa precisa de um objeto de valor constante, cujo valor é o mesmo em cada uso de um programa, uma constante **const** é usada. Entretanto, se um programa precisa de um objeto de valor constante, cujo valor é determinado apenas quando o objeto é criado e pode ser diferente para execuções diversas do programa, uma constante **readonly** é usada.

A discussão sobre valores vinculados a constantes nomeadas naturalmente leva ao assunto da inicialização, porque vincular um valor a uma constante nomeada é o mesmo processo, exceto por ser permanente.

Em muitos casos, é conveniente para as variáveis ter valores antes de o código do programa ou subprograma no qual elas são declaradas começar a executar. A vinculação de uma variável a um valor no momento em que ela é vinculada ao armazenamento é chamada de **inicialização**. Se a variável é estaticamente vinculada ao armazenamento, a vinculação e a inicialização ocorrem antes do tempo de execução. Nesses casos, o valor inicial deve ser especificado como um literal, ou como uma expressão cujos operandos não literais sejam constantes nomeadas já definidas. Se a vinculação ao armazenamento for dinâmica, a inicialização também é dinâmica e os valores iniciais podem ser quaisquer expressões.

Na maioria das linguagens, a inicialização é especificada na declaração que cria a variável. Por exemplo, em C++, poderíamos ter

```
int sum = 0;
int* ptrSum = &sum;
char name[] = "George Washington Carver";
```

RESUMO

A distinção entre maiúsculas e minúsculas e o uso de sublinhados são questões de projeto para nomes.

As variáveis podem ser caracterizadas por seis atributos: nome, endereço, valor, tipo, tempo de vida e escopo.

Apelidos são duas ou mais variáveis vinculadas ao mesmo endereço de armazenamento. Eles são considerados prejudiciais à confiabilidade, mas são difíceis de serem eliminados completamente de uma linguagem.

A vinculação é a associação de atributos a entidades de programa. O conhecimento dos tempos de vinculação de atributos a entidades é essencial para se entender a semântica das linguagens de programação. A vinculação pode ser estática ou dinâmica. Declarações, tanto explícitas quanto implícitas, fornecem uma forma de especificar a vinculação estática de variáveis a tipos. Em geral, a vinculação dinâmica permite maior flexibilidade, à custa de legibilidade, eficiência e confiabilidade.

Variáveis escalares podem ser separadas em quatro categorias, considerando seus tempos de vida: estáticas, dinâmicas da pilha, dinâmicas do monte explícitas e dinâmicas do monte implícitas.

[11] Em C#, os construtores estáticos são executados em algum momento indeterminado, antes que a classe seja instanciada.

O escopo estático é um recurso central de ALGOL 60 e de alguns de seus descendentes. Ele fornece um método simples, confiável e eficiente de permitir visibilidade a variáveis não locais em subprogramas. O escopo dinâmico fornece mais flexibilidade do que o escopo estático, mas à custa de legibilidade, confiabilidade e eficiência.

A maioria das linguagens funcionais permite que o usuário crie escopos locais com construções `let`, as quais limitam o escopo dos nomes definidos.

O ambiente de referenciamento de uma sentença é a coleção de todas as variáveis visíveis para aquela sentença.

Constantes nomeadas são simplesmente variáveis vinculadas a valores apenas uma vez.

QUESTÕES DE REVISÃO

1. Quais são as questões de projeto para nomes?
2. Qual é o perigo em potencial dos nomes sensíveis à distinção maiúsculas/minúsculas?
3. O que é um apelido?
4. Que categoria de variáveis de referência em C++ é sempre composta de apelidos?
5. O que é o lado *esquerdo* de uma variável? O que é o lado *direito*?
6. Defina *vinculação* e *tempo de vinculação*.
7. Após o projeto e a implementação de uma linguagem, quais são os quatro tipos de vinculações que podem ocorrer em um programa?
8. Defina *vinculação estática* e *vinculação dinâmica*.
9. Quais são as vantagens e desvantagens das declarações implícitas?
10. Quais são as vantagens e desvantagens da vinculação de tipos dinâmica?
11. Defina *variáveis estáticas*, *dinâmicas da pilha*, *dinâmicas do monte explícitas* e *dinâmicas do monte implícitas*. Quais são suas vantagens e desvantagens?
12. Defina *tempo de vida*, *escopo*, *escopo estático* e *escopo dinâmico*.
13. Como a referência a uma variável não local em um programa com escopo estático está conectada à sua definição?
14. Qual é o problema geral do escopo estático?
15. O que é o ambiente de referenciamento de uma sentença?
16. O que é um ancestral estático de um subprograma? O que é um ancestral dinâmico de um subprograma?
17. O que é um bloco?
18. Qual é a finalidade das construções `let` em linguagens funcionais?
19. Qual é a diferença entre os nomes definidos em uma construção `let` de ML e as variáveis declaradas em um bloco de C?

20. Descreva o encapsulamento de um `let` de F#, dentro de uma função e fora de todas as funções.
21. Quais são as vantagens e as desvantagens do escopo dinâmico?
22. Quais são as vantagens das constantes nomeadas?

PROBLEMAS

1. Qual das seguintes formas de identificador é mais legível? Argumente.

```
SumOfSales
sum_of_sales
SUMOFSALES
```

2. Algumas linguagens de programação não têm tipos. Quais são as vantagens e desvantagens óbvias da ausência de tipos em uma linguagem?
3. Escreva uma sentença de atribuição simples com um operador aritmético em alguma linguagem que você conheça. Para cada componente da sentença, liste as vinculações necessárias para determinar a semântica quando a sentença é executada. Para cada vinculação, indique o tempo de vinculação usado para a linguagem.
4. A vinculação de tipos dinâmica está muito relacionada às variáveis dinâmicas do monte. Explique esse relacionamento.
5. Descreva uma situação na qual uma variável sensível ao histórico em um subprograma é útil.
6. Considere o seguinte esqueleto de programa em JavaScript:

```
// O programa principal
var x;
function sub1() {
  var x;
  function sub2() {
    . . .
  }
}
function sub3() {
  . . .
}
```

Assuma que a execução desse programa é na seguinte ordem de unidades:

`main` chama `sub1`
`sub1` chama `sub2`
`sub2` chama `sub3`

a. Supondo escopo estático, a seguir, qual declaração de `x` é a correta para uma referência a `x`?
   i. `sub1`
   ii. `sub2`
   iii. `sub3`

b. Repita o exercício a, mas suponha escopo dinâmico.

7. Suponha que o seguinte programa em JavaScript foi interpretado por meio de regras de escopo estático. Que valor de `x` é impresso na função `sub1`? Sob regras de escopo dinâmico, qual valor de `x` é impresso na função `sub1`?

```
var x;
function  sub1() {
  document.write("x = " + x + "");
}
function  sub2() {
  var  x;
  x = 10;
  sub1();
}
x = 5;
sub2();
```

8. Considere o seguinte programa em JavaScript:

```
var x, y, z;
function sub1() {
  var a, y, z;
  function sub2() {
    var a, b, z;
    . . .
  }
  . . .
}
function  sub3() {
  var  a, x, w;
  . . .
}
```

Liste todas as variáveis, com as unidades de programa onde elas estão declaradas, visíveis nos corpos de `sub1`, `sub2` e `sub3`, supondo que é usado escopo estático.

9. Considere o seguinte programa em Python:

```
x = 1;
y = 3;
z = 5;
def sub1():
  a = 7;
  y = 9;
  z = 11;
```

```
    . . .
  def sub2():
    global  x;
    a = 13;
    x = 15;
    w = 17;
    . . .
    def sub3():
      nonlocal  a;
      a = 19;
      b = 21;
      z = 23;
      . . .
. . .
```

Liste todas as variáveis, com as unidades de programa onde elas estão declaradas, visíveis nos corpos de `sub1`, `sub2` e `sub3`, supondo que é usado escopo estático.

10. Considere o seguinte programa em C:

```
void fun(void) {
  int a, b, c; /* definição 1 */
  . . .
  while (. . .) {
    int b, c, d; /*definição 2 */
    . . . <------------ 1
    while (. . .) {
    int c, d, e; /* definição 3 */
    . . . <-------------  2
    }
   . . . <--------------  3
   }
  . . . <----------------  4
}
```

Para cada um dos quatro pontos marcados nessa função, liste cada variável visível, com o número da sentença de definição que a define.

11. Considere o seguinte esqueleto de programa em C:

```
void fun1(void);  /* protótipo */
void fun2(void);  /* protótipo */
void fun3(void);  /* protótipo */
void main() {
  int a, b, c;
  . . .
 }
void  fun1(void) {
  int  b, c, d;
  . . .
 }
void  fun2(void) {
```

```
  int  c, d, e;
  . . .
 }
void  fun3(void) {
  int  d, e, f;
  . . .
 }
```

Dadas as seguintes sequências de chamadas e supondo que o escopo dinâmico é usado, que variáveis são visíveis durante a execução da última função chamada? Inclua com cada variável visível o nome da função na qual foi definida.

a. `main` chama `fun1`; `fun1` chama `fun2`; `fun2` chama `fun3`.
b. `main` chama `fun1`; `fun1` chama `fun3`.
c. `main` chama `fun2`; `fun2` chama `fun3`; `fun3` chama `fun1`.
d. `main` chama `fun3`; `fun3` chama `fun1`.
e. `main` chama `fun1`; `fun1` chama `fun3`; `fun3` chama `fun2`.
f. `main` chama `fun3`; `fun3` chama `fun2`; `fun2` chama `fun1`.

12. Considere o programa a seguir, escrito em sintaxe do tipo JavaScript:

```
// programa principal
var x, y, z;

function sub1() {
var a, y, z;
. . .
}
function sub2() {
  var a, b, z;
  . . .
}
function sub3() {
  var a, x, w;
  . . .
}
```

Dadas as seguintes sequências de chamadas e supondo que é usado escopo dinâmico, quais variáveis são visíveis durante a execução do último subprograma ativado? Inclua com cada variável visível o nome da função na qual ela foi declarada.

a. `main` chama `sub1`; `sub1` chama `sub2`; `sub2` chama `sub3`.
b. `main` chama `sub1`; `sub1` chama `sub3`.
c. `main` chama `sub2`; `sub2` chama `sub3`; `sub3` chama `sub1`.
d. `main` chama `sub3`; `sub3` chama `sub1`.
e. `main` chama `sub1`; `sub1` chama `sub3`; `sub3` chama `sub2`.
f. `main` chama `sub3`; `sub3` chama `sub2`; `sub2` chama `sub1`.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Perl permite tanto escopo estático quanto um tipo de escopo dinâmico. Escreva um programa em Perl que use ambos e mostre a diferença real entre os dois. Explique a diferença entre o escopo dinâmico descrito neste capítulo e o implementado em Perl.
2. Escreva um programa em Common Lisp que mostre a diferença entre escopo estático e dinâmico.
3. Escreva um *script* em JavaScript que tenha subprogramas aninhados em três níveis de profundidade e nos quais cada subprograma aninhado referencie variáveis definidas em todos os seus subprogramas que o envolvem no aninhamento.
4. Repita o exercício anterior com Python.
5. Escreva uma função em C99 que inclua a seguinte sequência de sentenças:

```
x = 21;
int x;
x = 42;
```

   Execute o programa e explique os resultados. Reescreva o mesmo código em C++ e Java e compare os resultados.

12. Escreva programas de teste em C++, Java e C# para determinar o escopo de uma variável declarada em uma sentença **`for`**. Especificamente, o código deve determinar se tal variável é visível após o corpo da sentença **`for`**.
13. Escreva três funções em C ou C++: uma que declare um vetor grande estaticamente, outra que declare o mesmo vetor grande na pilha e outra que crie o mesmo vetor grande no monte. Chame cada um desses subprogramas um grande número de vezes (ao menos 100 mil vezes) e mostre na tela o tempo exigido para cada um. Explique os resultados.

6

Tipos de dados

6.1 Introdução
6.2 Tipos de dados primitivos
6.3 Cadeias de caracteres
6.4 Tipos enumeração
6.5 Tipos de matrizes
6.6 Matrizes associativas
6.7 Registros
6.8 Tuplas
6.9 Listas
6.10 Uniões
6.11 Ponteiros e referências
6.12 Verificação de tipos
6.13 Tipagem forte
6.14 Equivalência de tipos
6.15 Teoria e tipos de dados

Este capítulo primeiro apresenta o conceito de tipo de dados e as características dos tipos de dados primitivos mais comuns. Então, são discutidos os projetos de enumerações e de tipos subfaixas. A seguir, são investigados os detalhes dos tipos de dados estruturados – especificamente matrizes, matrizes associativas, registros, tuplas, listas e uniões. Esta seção é seguida por uma visão aprofundada sobre os ponteiros e as referências.

Para cada uma das várias categorias de tipos de dados, são enunciadas as questões de projeto e descritas as escolhas feitas pelos projetistas de algumas linguagens comuns. Esses projetos são então avaliados.

As próximas três seções fornecem uma minuciosa investigação da verificação de tipos, da tipagem forte e das regras de equivalência de tipos. A última seção do capítulo apresenta noções básicas sobre a teoria de tipos de dados.

Métodos de implementação para tipos de dados algumas vezes têm impacto significativo no projeto desses tipos. Logo, a implementação dos diversos tipos de dados é outro tema importante deste capítulo, especialmente para matrizes.

6.1 INTRODUÇÃO

Um **tipo de dado** define uma coleção de valores de dados e um conjunto de operações predefinidas sobre eles. Programas de computador produzem resultados por meio da manipulação de dados. Um fator importante para determinar a facilidade com a qual eles realizam suas tarefas é quão bem os tipos de dados disponíveis na linguagem usada casam com os objetos do mundo real do problema tratado. Logo, é crucial uma linguagem oferecer suporte para uma coleção apropriada de tipos e estruturas de dados.

Os conceitos contemporâneos de tipagem de dados evoluíram nos últimos 60 anos. Nas primeiras linguagens, todas as estruturas de dados do espaço do problema tinham de ser modeladas com apenas poucas estruturas de dados suportadas pela linguagem. Por exemplo, nas versões de Fortran pré-90, as listas ligadas e as árvores binárias eram implementadas com vetores.

As estruturas de dados de COBOL deram o primeiro passo na direção oposta à do modelo de Fortran I ao permitir que os programadores especificassem a precisão dos valores de dados decimais, e também por fornecer um tipo de dados estruturado para registros de informação. PL/I estendeu a capacidade da especificação de precisão para os tipos inteiros e de ponto flutuante. Os projetistas de PL/I incluíam muitos tipos de dados, com a intenção de suportar uma grande faixa de aplicações. Uma abordagem melhor, introduzida em ALGOL 68, é fornecer alguns tipos básicos e operadores de definição de estrutura flexíveis que permitam a um programador projetar uma estrutura de dados para cada necessidade. Claramente, essa foi uma das inovações mais importantes na evolução do projeto de tipos de dados. Tipos de dados definidos pelo usuário também fornecem uma legibilidade melhorada, por meio do uso de nomes significativos para os tipos. Eles permitem a verificação de tipos das variáveis de uma categoria especial de uso que, de outra forma, seria impossível. Tipos de dados definidos pelo usuário também ajudam na facilidade de fazer modificações: um programador pode modificar o tipo de uma categoria de variáveis em um programa trocando apenas uma sentença de definição de tipo.

Levando o conceito de tipo definido pelo usuário um passo adiante, chegamos aos tipos abstratos de dados, suportados pela maioria das linguagens de programação

projetadas desde a metade dos anos 1980. A ideia fundamental de um tipo abstrato de dados é que a interface de um tipo, visível para o usuário, é separada da representação e do conjunto de operações sobre valores desse tipo, ocultos do usuário. Todos os tipos fornecidos por uma linguagem de programação de alto nível são tipos de dados abstratos. Tipos de dados abstratos definidos pelo usuário são discutidos em detalhes no Capítulo 11.

Existem diversos usos para o sistema de tipos de uma linguagem de programação. O mais prático é a detecção de erros. O processo e o valor da verificação de tipos, guiada pelo sistema de tipos da linguagem, são discutidos na Seção 6.12. Um segundo uso para um sistema de tipos é o auxílio prestado para a modularização de programas. Isso resulta da verificação de tipos em diferentes módulos, o que garante a consistência das interfaces entre eles. Outro uso para um sistema de tipos é a documentação. As declarações de tipo em um programa documentam informações sobre seus dados e fornecem pistas sobre o comportamento de um programa.

O sistema de tipos de uma linguagem de programação define como um tipo é associado a cada expressão na linguagem e inclui suas regras para equivalência e compatibilidade de tipos. Certamente, uma das etapas mais importantes para se entender a semântica de uma linguagem de programação é entender seu sistema de tipos.

Os dois tipos mais comuns de dados estruturados (não escalares) nas linguagens imperativas são as matrizes e os registros, apesar de a popularidade das matrizes associativas ter aumentado nos últimos anos. As listas são uma parte fundamental das linguagens de programação funcionais desde que a primeira dessas linguagens surgiu, em 1959 (Lisp). No decorrer da última década, a crescente popularidade da programação funcional levou à adição de listas nas principais linguagens imperativas, como Python e C#.

Os tipos de dados estruturados são definidos por meio de operadores de tipos, ou construtores, usados para formar expressões de tipos. Por exemplo, C usa colchetes e asteriscos como operadores de tipos para especificar matrizes e ponteiros.

É conveniente, tanto logicamente quanto concretamente, pensarmos em variáveis em termos de descritores. Um **descritor** é a coleção de atributos de uma variável. Em uma implementação, um descritor é uma área de memória que armazena os atributos de uma variável. Se os atributos são todos estáticos, os descritores são necessários apenas em tempo de compilação. Tais descritores são construídos pelo compilador, normalmente como parte da tabela de símbolos, e usados durante a compilação. Para atributos dinâmicos, entretanto, parte ou todo o descritor deve ser mantido durante a execução. Nesse caso, o descritor é usado pelo sistema de tempo de execução. Em todos os casos, os descritores são usados para verificação de tipos e na construção do código para as operações de alocação e liberação.

Deve-se tomar cuidado ao usar o termo *variável*. Quem usa apenas linguagens imperativas tradicionais pode considerar identificadores como variáveis, mas isso pode causar confusão ao se considerar os tipos de dados. Em algumas linguagens de programação, os identificadores não têm tipos de dados. É prudente lembrar que identificadores são apenas um dos atributos de uma variável.

A palavra *objeto* é normalmente associada ao valor de uma variável e ao espaço que ocupa. Neste livro, entretanto, reservamos a palavra *objeto* exclusivamente para instâncias de tipos de dados abstratos definidos pelo usuário e pela linguagem, em vez de também usá-la para os valores de todas as variáveis de programa de tipos predefinidos. Objetos são discutidos em detalhes nos Capítulos 11 e 12.

Nas seções seguintes são discutidos muitos tipos de dados comuns. Para a maioria, são levantadas questões particulares ao tipo. Para todos, um ou mais exemplos de projeto são descritos. Uma questão de projeto é fundamental para todos os tipos de dados: quais operações são fornecidas para variáveis de tipo e como elas são especificadas?

6.2 TIPOS DE DADOS PRIMITIVOS

Tipos de dados não definidos em termos de outros são chamados de **tipos de dados primitivos**. Praticamente todas as linguagens de programação fornecem um conjunto de tipos de dados primitivos. Alguns dos tipos primitivos são meramente reflexos do hardware – por exemplo, a maioria dos tipos inteiros. Outros exigem apenas um pouco de suporte externo ao hardware para sua implementação.

Para especificar os tipos estruturados, são usados os tipos de dados primitivos de uma linguagem, junto com um ou mais construtores de tipo.

6.2.1 Tipos numéricos

Algumas das primeiras linguagens de programação tinham apenas tipos primitivos numéricos. Tipos numéricos ainda desempenham um papel central entre as coleções de tipos suportadas pelas linguagens contemporâneas.

6.2.1.1 Inteiro

O tipo de dados primitivo numérico mais comum é o **`inteiro`**. O hardware de muitos computadores suporta vários tamanhos de inteiros. Esses tamanhos de inteiros – e frequentemente alguns outros – são suportados por algumas linguagens de programação. Por exemplo, Java inclui quatro tamanhos de inteiro com sinal: **`byte`**, **`short`**, **`int`** e **`long`**. Algumas linguagens, por exemplo, C++ e C#, incluem tipos inteiros sem sinal, que são tipos para valores inteiros sem sinal. Tipos sem sinal são geralmente usados para dados binários.

Um valor inteiro com sinal é representado em um computador como uma cadeia de bits, com um dos bits (normalmente o mais à esquerda) representando o sinal. A maioria dos tipos inteiros é suportada diretamente pelo hardware. Um exemplo de um tipo inteiro não suportado diretamente pelo hardware é o tipo inteiro longo de Python (F# também fornece tais inteiros). Valores desse tipo podem ter um tamanho ilimitado. Valores inteiros longos podem ser especificados como literais, como no seguinte exemplo:

```
243725839182756281923L
```

As operações aritméticas inteiras em Python, que produzem valores muito grandes para serem representados com o tipo **`int`**, os armazenam como valores do tipo inteiro longo.

Um inteiro negativo pode ser armazenado na notação sinal-magnitude, na qual o bit de sinal é usado para indicar números negativos e o restante da cadeia de bits representa o valor absoluto do número. A notação sinal-magnitude, entretanto, não é usada para aritmética computacional. A maioria dos computadores usa agora uma notação chamada **complemento de dois** para armazenar inteiros negativos, a qual é conveniente para a

adição e a subtração. Na notação de complemento de dois, a representação de um inteiro negativo é formada ao tomarmos o complemento lógico da versão positiva do número e adicionarmos um. A notação de complemento de um ainda é usada por alguns computadores. Nela, o negativo de um inteiro é armazenado como o complemento lógico de seu valor absoluto. A notação de complemento de um tem a desvantagem de ter duas representações para zero. Procure um livro sobre programação em linguagem de montagem para saber mais detalhes sobre as representações inteiras.

6.2.1.2 Ponto flutuante

Tipos de dados de **ponto flutuante** modelam números reais, mas as representações são apenas aproximações para muitos valores reais. Por exemplo, nenhum dos números fundamentais π ou e (a base para logaritmos naturais) pode ser corretamente representado em notação de ponto flutuante. Evidentemente, nenhum desses números pode ser representado precisamente em qualquer quantidade finita de memória de computador. Na maioria dos computadores, os números de ponto flutuante são armazenados em binário, o que agrava o problema. Por exemplo, mesmo o valor 0,1 em decimal não pode ser representado por um número finito de dígitos binários.[1] Outro problema dos tipos de ponto flutuante é a perda de precisão em operações aritméticas. Para mais informações sobre os problemas na notação de ponto flutuante, consulte um livro sobre análise numérica.

Valores de ponto flutuante são representados como frações e expoentes, uma forma emprestada da notação científica. Os computadores mais antigos usavam uma variedade de diferentes representações para valores de ponto flutuante. Entretanto, a maioria das máquinas mais novas usa o formato padrão para ponto flutuante da IEEE, chamado IEEE Padrão de Ponto Flutuante 754 (IEEE Floating-Point Standard 754). Os implementadores de linguagens usam quaisquer representações suportadas pelo hardware. A maioria das linguagens inclui dois tipos de ponto flutuante, chamados **float** e **double**. O tipo float é o tamanho padrão, e normalmente é armazenado em quatro bytes de memória. O tipo double é fornecido para situações nas quais são necessárias partes fracionárias maiores e/ou uma faixa de expoentes maior. Variáveis de precisão dupla normalmente ocupam o dobro de armazenamento das variáveis float e fornecem ao menos o dobro do número de bits de fração.

A coleção de valores que podem ser representados por um tipo de ponto flutuante é definida em termos de precisão e faixa. **Precisão** é a exatidão da parte fracionária de um valor, medida como o número de bits. **Faixa** é a combinação da faixa de frações e, mais importante, de expoentes.

A Figura 6.1 mostra o formato IEEE Padrão de Ponto Flutuante 754 para representação de precisão simples e dupla (IEEE, 1985). Detalhes dos formatos IEEE podem ser encontrados em Tanenbaum (2005).

6.2.1.3 Complexo

Algumas linguagens de programação suportam um tipo de dados complexo – por exemplo, Fortran e Python. Valores complexos são representados como pares ordenados de

[1] 0,1 em decimal equivale a 0.0001100110011... em binário.

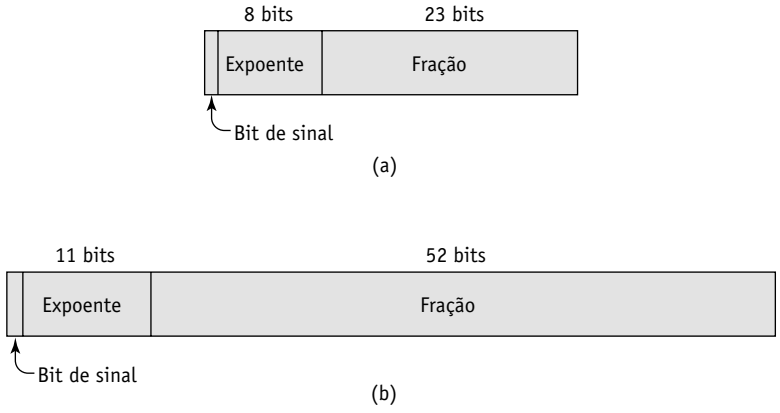


FIGURA 6.1
Formatos de ponto flutuante da IEEE: (a) precisão simples, (b) precisão dupla.

valores de ponto flutuante. Em Python, a parte imaginária de um literal complexo é especificada seguindo-a por `j` ou `J` – por exemplo,

```
(7 + 3j)
```

Linguagens que suportam um tipo complexo incluem operações para aritmética em valores complexos.

6.2.1.4 Decimal

A maioria dos computadores de grande porte projetados para suportar aplicações de sistemas de negócios tem suporte em hardware para tipos de dados **decimais**. Tipos de dados decimais armazenam um número fixo de dígitos decimais, com o ponto decimal implícito em uma posição fixa no valor. Esses são os tipos de dados primários para processamento de dados de negócios e, dessa forma, são essenciais para COBOL. C# e F# também têm tipos de dados decimais.

Tipos decimais têm a vantagem de serem capazes de armazenar precisamente valores decimais, ao menos dentro de uma faixa restrita, o que não pode ser feito com tipos de ponto flutuante. Por exemplo, o número 0,1 (em decimal) pode ser representado exatamente em um tipo decimal, mas não em um tipo de ponto flutuante, como mencionado na Seção 6.2.1.2. As desvantagens dos tipos decimais são que a faixa de valores é restrita porque nenhum expoente é permitido, e sua representação em memória é um pouco dispendiosa, pelas razões discutidas no parágrafo a seguir.

Tipos decimais são armazenados de maneira muito parecida com as cadeias de caracteres, usando códigos binários para os dígitos decimais. Essas representações são chamadas de **decimais codificados em binário** (BCD – binary coded decimal). Em alguns casos, eles são armazenados um dígito por byte, mas em outros são agrupados em dois dígitos por byte. De qualquer forma, ocupam mais armazenamento do que as representações binárias. São necessários pelo menos quatro bits para codificar um dígito decimal. Logo, armazenar um número decimal codificado de seis dígitos requer 24 bits de memória. Contudo, são necessários apenas 20 bits para armazenar o mesmo número

em binário.[2] As operações em valores decimais são efetuadas no hardware, em máquinas que têm esses recursos; caso contrário, são simuladas no software.

6.2.2 Tipos booleanos

Os tipos **booleanos** talvez sejam os mais simples. Sua faixa de valores tem apenas dois elementos: um para verdadeiro e outro para falso. Eles foram introduzidos em ALGOL 60 e incluídos na maioria das linguagens de propósito geral projetadas desde 1960. Uma exceção popular é C89, em que expressões numéricas são usadas como condicionais. Em tais expressões, todos os operandos com valores diferentes de zero são considerados verdadeiros, e zero é considerado falso. Apesar de C99 e C++ terem um tipo booleano, também permitem que expressões numéricas sejam usadas como se fossem booleanas. Esse não é o caso nas linguagens subsequentes, como Java e C#.

Tipos booleanos são geralmente usados para representar escolhas ou como *flags* em programas. Apesar de outros tipos, como inteiros, poderem ser usados para tais propósitos, o uso de tipos booleanos é mais legível.

Um valor booleano poderia ser representado por um único bit, mas, como um único bit de memória não pode ser acessado de maneira eficiente em muitas máquinas, eles são armazenados na menor célula de memória eficientemente endereçável, um byte.

6.2.3 Caracteres

Dados na forma de caracteres são armazenados nos computadores como codificações numéricas. Tradicionalmente, a codificação mais usada era o Padrão Americano de Codificação para Intercâmbio de Informação (ASCII – American Standard Code for Information Interchange) de 8 bits, que usa os valores de 0 a 127 para codificar 128 caracteres diferentes. O ISO 8859-1 é outra codificação de 8 bits para caracteres, mas ele permite 256 caracteres diferentes.

Devido à globalização dos negócios e da necessidade de os computadores se comunicarem com outros computadores pelo mundo, o conjunto de caracteres ASCII se tornou inadequado. Em resposta, em 1991, o Consórcio Unicode publicou o padrão UCS-2, um conjunto de caracteres de 16 bits. Essa codificação de caracteres é geralmente chamada de Unicode. Unicode inclui os caracteres da maioria das linguagens naturais do mundo. Por exemplo, Unicode inclui o alfabeto Cirílico, usado na Sérvia, e os dígitos tailandeses. Os primeiros 128 caracteres do Unicode são idênticos àqueles do ASCII. Java foi a primeira linguagem amplamente usada a empregar o conjunto de caracteres Unicode. Desde então, ele foi adotado em JavaScript, Python, Perl, C# e F#.

Após 1991, o Consórcio Unicode, em cooperação com a Organização Internacional de Padrões (ISO – International Standards Organization) desenvolveu uma codificação de caracteres de 4 bytes chamada UCS-4 ou UTF-32, que é descrita pelo padrão ISO/IEC 10646, publicado em 2000.

Para fornecer os meios de processar codificações de caracteres simples, a maioria das linguagens de programação inclui um tipo primitivo para eles. Entretanto, Python suporta caracteres únicos apenas como cadeias de caracteres de tamanho 1.

[2] Evidentemente, a não ser que um programa precise manter um número extenso de valores decimais grandes, a diferença é insignificante.

6.3 CADEIAS DE CARACTERES

Um **tipo cadeia de caracteres** é aquele no qual os valores consistem em sequências de caracteres. Constantes do tipo cadeias de caracteres são usadas para rotular a saída, e a entrada e a saída de todos os tipos de dados é geralmente feita em termos de cadeias. É claro que cadeias de caracteres também são um tipo essencial para todos os programas que realizam manipulação de caracteres.

6.3.1 Questões de projeto

As duas questões de projeto mais importantes, específicas às cadeias de caracteres, são:

- As cadeias devem ser um tipo especial de vetor de caracteres ou um tipo primitivo?
- As cadeias devem ter tamanho estático ou dinâmico?

6.3.2 Cadeias e suas operações

As operações mais comuns em cadeias são: atribuição, concatenação, referência a subcadeias, comparação e casamento de padrões.

Uma **referência a subcadeias** é uma referência a uma subcadeia de determinada cadeia. As referências a subcadeias são discutidas no contexto mais geral das matrizes, no qual são denominadas **fatias**.

Em geral, tanto a operação de atribuição quanto a de comparação de cadeias de caracteres são complicadas pela possibilidade de operandos de diferentes tamanhos. Por exemplo, o que acontece quando uma cadeia maior é atribuída a uma cadeia menor, ou vice-versa? Normalmente, escolhas simples e sensíveis são feitas para tais situações, apesar de os programadores terem dificuldade de lembrar delas.

Em algumas linguagens, o casamento de padrões é suportado diretamente na linguagem. Em outras, é fornecido por uma função ou biblioteca de classes.

Se as cadeias não são definidas como um tipo primitivo, os dados da cadeia são normalmente armazenados em vetores de caracteres e referenciados como tal na linguagem. Essa é a estratégia adotada por C e C++, que utilizam vetores de **`char`** para armazenar cadeias de caracteres. Essas linguagens fornecem uma coleção de operações em cadeias por meio de bibliotecas padrão. Muitos usuários de cadeias e muitas das funções das bibliotecas usam a convenção de que as cadeias de caracteres são terminadas com um caractere especial, nulo, representado por um zero. Essa é uma alternativa em relação a manter o tamanho das variáveis do tipo cadeia. As operações de bibliotecas simplesmente efetuam suas operações até que o caractere nulo apareça na cadeia operada. Funções de biblioteca que produzem cadeias normalmente fornecem o caractere nulo. Os literais de cadeias construídos pelo compilador também têm o caractere nulo. Por exemplo, considere a declaração:

```
char str[] = "apples";
```

Nesse exemplo, `str` é um vetor de elementos do tipo **`char`**, especificamente `apples0`, onde `0` é o caractere nulo.

Algumas das funções de biblioteca mais usadas para cadeias de caracteres em C e C++ são: `strcpy`, que move cadeias; `strcat`, que concatena uma cadeia com outra; `strcmp`, que compara duas cadeias lexicograficamente (pela ordem dos códigos de caracteres); e `strlen`, que retorna o número de caracteres em uma cadeia, não contando o caractere nulo. Os parâmetros e os valores de retorno para a maioria das funções de manipulações de cadeias são ponteiros do tipo **`char`**, que apontam para vetores de caracteres (**`char`**). Os parâmetros também podem ser cadeias literais.

As funções de manipulação de cadeias da biblioteca padrão do C, também disponíveis em C++, são inerentemente inseguras e têm levado a diversos erros de programação. O problema é que as funções dessa biblioteca que movem dados de cadeias não verificam transbordamentos (*overflows*) no destino. Por exemplo, considere a seguinte chamada a `strcpy`:

```
strcpy(dest, src);
```

Se o tamanho de `dest` for 20 e o de `src` for 50, `strcpy` escreverá sobre os 30 bytes subsequentes a `dest`. A questão é que `strcpy` não sabe o tamanho de `dest`, então não pode garantir que a memória subsequente não será sobrescrita. O mesmo problema pode ocorrer em várias das outras funções da biblioteca de cadeias de C. Além das cadeias no estilo C, C++ também suporta cadeias por meio de sua biblioteca de classes padrão, a qual também é semelhante à de Java. Em função das inseguranças da biblioteca de cadeias de C, programadores C++ devem usar a classe `string` da biblioteca padrão, em vez de vetores de **`char`** e da biblioteca de cadeias de C.

Em Java, cadeias são suportadas pelas classes `String`, cujos valores são cadeias constantes, e `StringBuffer`, cujos valores são modificáveis e são mais parecidos com vetores de caracteres. Esses valores são especificados com métodos da classe `StringBuffer`. C# e Ruby incluem classes similares às de Java para representar cadeias.

Python tem cadeias como um tipo primitivo e operações para referência a subcadeias, concatenação, indexação para acessar caracteres individuais, assim como métodos para busca e substituição. Existe também uma operação para verificar se um caractere pertence a uma cadeia. Então, mesmo que as cadeias de Python sejam tipos primitivos, para caracteres e referências a subcadeias, elas agem de maneira bastante parecida à das matrizes de caracteres. Entretanto, as cadeias em Python são imutáveis, de modo similar aos objetos da classe `String` em Java.

Em F#, cadeias são uma classe. Os caracteres individuais, que são representados em Unicode UTF-16, podem ser acessados, mas não alterados. Cadeias podem ser concatenadas com o operador `+`. Em ML, cadeia é um tipo primitivo imutável. Ela usa `^` como operador de concatenação e contém funções para referenciar subcadeias e obter o tamanho de uma cadeia.

Perl, JavaScript, Ruby e PHP incluem operações de casamento de padrões predefinidas. Nessas linguagens, as expressões de casamento de padrões são levemente baseadas em expressões regulares matemáticas. De fato, são chamadas de **expressões regulares**. Elas evoluíram desde o primeiro editor de linhas do UNIX, o `ed`, para se tornarem parte das linguagens de interpretação de comandos do UNIX. Por fim, elas cresceram para sua forma complexa atual. Existe ao menos um livro com-

» *nota histórica*

SNOBOL 4 foi a primeira linguagem bastante conhecida a suportar o casamento de padrões.

pleto sobre esse tipo de expressões de casamento de padrões (Friedl, 2006). Nesta seção, fornecemos um breve panorama do estilo dessas expressões por meio de dois exemplos relativamente simples.

Considere a expressão de padrão:

```
/[A-Za-z][A-Za-z\d]+/
```

Esse padrão casa (ou descreve) o formato de nomes típico em linguagens de programação. Os colchetes envolvem as classes de caracteres. A primeira classe de caracteres especifica todas as letras; a segunda especifica todas as letras e dígitos (um dígito é especificado com a abreviação `\d`). Se apenas a segunda classe de caracteres fosse incluída, não poderíamos impedir que um nome iniciasse com um dígito. O operador de adição seguinte à segunda categoria especifica que deve existir um ou mais caracteres desta. Então, o padrão como um todo casa com cadeias que iniciam com uma letra, seguida de uma ou mais letras ou dígitos.

Agora, considere a seguinte expressão de padrão:

```
/\d+\.?\d*|\.\d+/
```

Esse padrão casa com literais numéricos. O `\.` especifica um ponto decimal literal.[3] O ponto de interrogação quantifica o que o precede com zero ou mais aparições. A barra vertical (`|`) separa duas alternativas no padrão como um todo. A primeira alternativa casa com cadeias de um ou mais dígitos, possivelmente seguidos por um ponto decimal, seguido de zero ou mais dígitos; a segunda alternativa casa com cadeias que começam com um ponto decimal, seguidas por um ou mais dígitos.

Recursos para casamento de padrões por meio de expressões regulares são incluídos nas bibliotecas de classes de C++, Java, Python, C# e F#.

6.3.3 Opções de tamanho de cadeias

Existem diversas escolhas de projeto em relação ao tamanho dos valores das cadeias. Primeiro, o tamanho pode ser estático e definido quando a cadeia é criada. Tal cadeia é chamada de **cadeia de tamanho estático**. Essa é a escolha para as cadeias de Python, para os objetos imutáveis da classe `String` de Java, assim como para as classes similares na biblioteca de classes padrão de C++, para a classe predefinida `String` em Ruby e para a biblioteca de classes do .NET disponível para C# e F#.

A segunda opção é permitir que as cadeias tenham tamanhos variáveis, até um máximo declarado e fixado pela definição da variável, como exemplificado pelas cadeias em C e pelas cadeias no estilo de C em C++. Essas são chamadas de **cadeias de tamanho dinâmico limitado**. Tais variáveis podem armazenar qualquer número de caracteres entre zero e o máximo. Lembre-se de que as cadeias em C usam um caractere especial para indicar o final da cadeia, em vez de manter o tamanho dela.

A terceira opção é permitir que as cadeias tenham tamanho variado, sem um máximo, como em JavaScript, Perl e a biblioteca padrão de C++. Essas são chamadas de

[3] Deve ser empregado um "caractere de escape" para o ponto, usando-se uma barra invertida, pois o ponto tem um significado especial em uma expressão regular.

cadeias de tamanho dinâmico. Essa opção exige a sobrecarga da alocação e a liberação de armazenamento dinâmico, mas oferece a máxima flexibilidade.

6.3.4 Avaliação

Os tipos que representam cadeias são importantes para a facilidade de escrita de uma linguagem. Trabalhar com cadeias na forma de vetores pode ser mais trabalhoso do que com um tipo primitivo que representa cadeias. Por exemplo, considere uma linguagem que trata cadeias como vetores de caracteres e não tem uma função predefinida que faz o que `strcpy` em C faz. Então, uma simples atribuição de uma cadeia para outra necessitaria de um laço de repetição. A adição de cadeias como um tipo primitivo de uma linguagem não é dispendiosa, nem em termos da linguagem nem da complexidade do compilador. Logo, é difícil justificar a omissão de tipos primitivos para cadeias em algumas linguagens contemporâneas. É claro que fornecer cadeias por meio de uma biblioteca padrão é quase tão conveniente quanto tê-las como um tipo primitivo.

Operações sobre cadeias, como casamento de padrões simples e concatenação, são essenciais e devem ser incluídas para valores do tipo cadeia. Apesar de as cadeias de tamanho dinâmico serem obviamente as mais flexíveis, a sobrecarga de sua implementação deve ser pesada em relação à flexibilidade adicional.

6.3.5 Implementação de tipos cadeias de caracteres

Os tipos que representam cadeias de caracteres poderiam ser suportados diretamente em hardware; mas, na maioria dos casos, o armazenamento, a recuperação e a manipulação de cadeias são feitos em software. Quando os tipos cadeias de caracteres são representados como vetores de caracteres, a linguagem fornece poucas operações.

Um descritor para um tipo que representa cadeias de caracteres, necessário apenas durante a compilação, tem três campos. O primeiro campo de cada descritor é o nome do tipo. No caso de cadeias de caracteres estáticas, o segundo campo é o tamanho do tipo (em caracteres). O terceiro é o endereço do primeiro caractere. Esse descritor é mostrado na Figura 6.2. Cadeias de tamanho dinâmico limitado exigem um descritor em tempo de execução para armazenar tanto o tamanho máximo fixado quanto o tamanho atual, como mostrado na Figura 6.3. Cadeias de tamanho dinâmico exigem um descritor em tempo de execução mais simples, porque apenas o tamanho atual precisa ser armazenado. Apesar de mostrarmos os descritores como blocos de armazenamento independentes, na maioria dos casos eles são armazenados na tabela de símbolos.

As cadeias de tamanho dinâmico limitado de C e C++ não exigem descritores em tempo de execução, porque o final de uma cadeia é marcado com o caractere nulo. Elas não precisam do tamanho máximo, porque os valores de índices em referências a vetores não são verificados em relação à sua faixa nessas linguagens.

Cadeias de tamanho estático e de tamanho dinâmico limitado não exigem alocação dinâmica especial. No caso das cadeias de tamanho dinâmico limitado, um armazenamento suficiente para o tamanho máximo é alocado quando a variável do tipo cadeia é vinculada a ele; então, é necessário apenas um processo de alocação simples.

Cadeias de tamanho dinâmico exigem um gerenciamento de armazenamento mais complexo. O tamanho de uma cadeia e, dessa forma, o armazenamento ao qual ela está vinculada devem crescer e encolher dinamicamente.

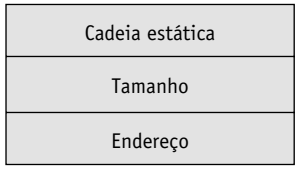


FIGURA 6.2
Descritor em tempo de compilação para cadeias estáticas.

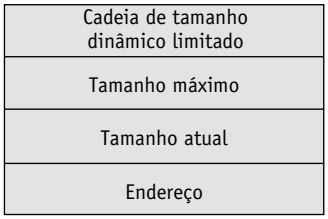


FIGURA 6.3
Descritor em tempo de execução para cadeias de tamanho dinâmico limitado.

Existem três abordagens para suportar a alocação e a liberação dinâmica necessárias para as cadeias de tamanho dinâmico. Primeiro, as cadeias podem ser armazenadas em uma lista encadeada, de forma que, quando uma cadeia cresce, as novas células necessárias podem vir de qualquer lugar do monte. As desvantagens desse método são o armazenamento extra ocupado pelas ligações na representação das listas e a complexidade das operações sobre cadeias.

A segunda abordagem é armazenar as cadeias como vetores de ponteiros para caracteres individuais, alocados no monte. Esse método ainda usa memória extra, mas o processamento de cadeias pode ser mais rápido do que com a abordagem que usa listas encadeadas.

A terceira alternativa é armazenar cadeias completas em células de armazenamento adjacentes. O problema desse método surge quando uma cadeia cresce: como o armazenamento adjacente às células existentes continua a ser alocado para a variável do tipo cadeia? Frequentemente, tal armazenamento não está disponível. Em vez disso, uma nova área de memória é encontrada para que possa armazenar a nova cadeia completa, e a parte antiga é movida para essa área. Então, as células de memória usadas para a cadeia antiga são liberadas. Essa última abordagem é a comumente usada. O problema geral de gerenciar a alocação e a liberação de segmentos de tamanhos variáveis é discutido na Seção 6.11.7.3.

Apesar de o método que usa listas encadeadas exigir mais armazenamento, os processos de alocação e liberação associados são simples. Entretanto, algumas operações sobre cadeias são mais lentas em função da necessidade de percorrer os ponteiros. Por outro lado, usar memória adjacente para cadeias completas resulta em operações sobre cadeias mais rápidas e requer significativamente menos armazenamento, mas os processos de alocação e liberação são mais lentos.

6.4 TIPOS ENUMERAÇÃO

Um **tipo enumeração** é aquele no qual todos os valores possíveis, os quais são constantes nomeadas, são fornecidos, ou enumerados, na definição. Tipos enumeração fornecem uma maneira de definir e agrupar coleções de constantes nomeadas, chamadas de **constantes de enumeração**. A definição de um tipo enumeração típico é mostrada neste exemplo em C#:

```
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

As constantes de enumeração são preenchidas implicitamente por atribuições de valores inteiros, 0, 1, . . ., mas podem ser atribuídos explicitamente quaisquer literais inteiros na definição do tipo.

6.4.1 Questões de projeto

As questões de projeto para tipos enumeração são as seguintes:

- Uma constante de enumeração pode aparecer em mais de uma definição de tipo? Se pode, como o tipo de uma ocorrência de tal constante é verificado no programa?
- Os valores de enumeração são convertidos em inteiros?
- Existem outros tipos que são convertidos em um tipo enumeração?

Todas essas questões de projeto são relacionadas à verificação de tipos. Se uma variável de enumeração é convertida em um tipo numérico, então existe pouco controle sobre sua faixa de operações válidas ou sobre sua faixa de valores. Se um valor do tipo `int` é convertido em um tipo enumeração, então uma variável do tipo enumeração pode ter quaisquer valores inteiros atribuídos a ela, independentemente de representar uma constante de enumeração ou não.

6.4.2 Projetos

Em linguagens que não têm tipos enumeração, os programadores normalmente simulam enumerações usando valores inteiros. Por exemplo, suponha que precisássemos representar cores em um programa em C e a linguagem não tivesse um tipo enumeração. Poderíamos usar 0 para representar o azul (blue), 1 para representar o vermelho (red) e assim por diante. Esses valores poderiam ser definidos como segue:

```
int red = 0, blue = 1;
```

No programa, poderíamos usar red e blue como se fossem tipos de cores. O problema dessa abordagem é que, como não definimos um tipo para nossas cores, não existe uma verificação de tipos quando elas são usadas. Por exemplo, seria permitido somar as duas, apesar de essa raramente ser uma operação desejada. Elas poderiam ser combinadas com qualquer outro operando de tipo numérico. Além disso, como são apenas variáveis, podem ser atribuídos quaisquer valores inteiros a elas, destruindo o relacionamento com as cores. Esse último problema pode ser evitado tornando-as constantes nomeadas.

C e Pascal foram as primeiras linguagens amplamente usadas a incluírem um tipo de dados de enumeração. C++ inclui os tipos de enumeração de C. Em C++ poderíamos ter:

```
enum colors {red, blue, green, yellow, black};
colors myColor = blue, yourColor = red;
```

O tipo `colors` usa os valores padrões internos para as constantes de enumeração, 0, 1, . . ., apesar de as constantes poderem receber qualquer literal inteiro (ou qualquer expressão com valores constantes). Os valores de enumeração são convertidos em **`int`** quando inseridos em um contexto inteiro. Isso permite seu uso em qualquer expressão numérica. Por exemplo, se o valor atual de `myColor` é blue, então a expressão

```
myColor++
```

atribuiria `green` a `myColor`.

C++ também permite que as constantes de enumeração sejam atribuídas a variáveis de qualquer tipo numérico, apesar de isso provavelmente ser um erro. Entretanto, nenhum outro valor de tipo é convertido em um tipo enumeração em C++. Por exemplo,

```
myColor = 4;
```

é inválido em C++. Essa atribuição seria válida se o lado direito tivesse sido convertido no tipo `colors`. Isso evita alguns erros em potencial.

Constantes de enumeração em C++ podem aparecer em apenas um tipo enumeração no mesmo ambiente de referenciamento.

Em 2004, um tipo enumeração foi adicionado à linguagem Java em sua versão 5.0. Todos os tipos enumeração em Java são subclasses implícitas da classe predefinida `Enum`. Como os tipos enumeração são classes, eles podem ter atributos de dados de instância, construtores e métodos. Sintaticamente, as definições de tipos enumeração em Java se parecem com aquelas de C++, exceto pelo fato de poderem incluir atributos, construtores e métodos. Os valores possíveis de uma enumeração são as únicas instâncias da classe possíveis. Todos os tipos enumeração herdam `toString`, assim como alguns outros métodos. Um vetor das instâncias de um tipo enumeração pode ser obtido com o método estático `values`. O valor numérico interno de uma variável de enumeração pode ser obtido com o método `ordinal`. Nenhuma expressão de nenhum outro tipo pode ser atribuída a uma variável de enumeração. Além disso, uma variável de enumeração nunca é convertida em qualquer outro tipo.

Tipos enumeração em C# são parecidos com os de C++, exceto pelo fato de que nunca são convertidos em inteiros. Então, as operações em tipos enumeração estão restritas àquelas que fazem sentido. Além disso, a faixa de valores é restringida àquela do tipo enumeração em particular.

Na ML, os tipos de enumeração são definidos como novos tipos, com declarações **`datatype`**. Por exemplo, poderíamos ter o seguinte:

```
datatype weekdays = Monday | Tuesday | Wednesday |
Thursday | Friday
```

O tipo dos elementos de `weekdays` é inteiro.

F# tem tipos de enumeração semelhantes aos de ML, exceto que a palavra reservada **`type`** é usada em lugar de **`datatype`** e que o primeiro valor é precedido por um operador OU (`|`).

Curiosamente, nenhuma das linguagens de *scripting* relativamente recentes contêm tipos de enumeração. Isso inclui Perl, JavaScript, PHP, Python, Ruby e Lua. Mesmo Java já tinha 10 anos quando os tipos enumeração foram adicionados.

6.4.3 Avaliação

Tipos enumeração podem oferecer vantagens tanto em relação à legibilidade quanto à confiabilidade. A legibilidade é melhorada de maneira muito direta: valores nomeados são facilmente reconhecidos, enquanto valores codificados não.

Na área da confiabilidade, os tipos enumeração de C#, F# e Java 5.0 oferecem duas vantagens: (1) nenhuma operação aritmética é permitida em tipos enumeração – isso impede a soma de dias da semana, por exemplo – e (2) nenhuma variável de enumeração pode ter um valor atribuído a ela fora de sua faixa definida.[4] Se a enumeração `colors` tem 10 constantes de enumeração e usa `0..9` como seus valores internos, nenhum número maior que `9` pode ser atribuído à variável do tipo `colors`.

Como C trata as variáveis de enumeração como variáveis inteiras, ele não fornece nenhuma dessas duas vantagens.

C++ é um pouco melhor. Valores numéricos podem ser atribuídos à variáveis do tipo enumeração apenas se eles puderem ser convertidos no tipo da variável recebe a atribuição. Valores numéricos atribuídos a variáveis do tipo enumeração são verificados para determinar se estão na faixa dos valores internos do tipo enumeração. Infelizmente, se o usuário usar uma ampla faixa de valores atribuídos explicitamente, essa verificação não é eficiente. Por exemplo,

```
enum colors {red = 1, blue = 1000, green = 100000}
```

Nesse exemplo, um valor atribuído a uma variável do tipo `colors` será verificado apenas para determinar se está na faixa de `1..100000`.

6.5 TIPOS DE MATRIZES

Uma **matriz** é um agregado homogêneo de elementos de dados no qual um elemento individual é identificado por sua posição na agregação, relativamente ao primeiro elemento. Os elementos de dados individuais de uma matriz são do mesmo tipo. As referências aos elementos são especificadas com expressões de índices. Se qualquer uma das expressões de índices em uma referência incluir variáveis, então a referência exigirá um cálculo adicional em tempo de execução para determinar o endereço da posição de memória referenciada.

Em muitas linguagens, como C, C++, Java e C#, todos os elementos de uma matriz precisam ser do mesmo tipo. Nelas, ponteiros e referências são restritos para que possam

[4]Em C# e F#, um valor inteiro pode ser convertido em um tipo de enumeração e atribuído ao nome de uma variável de enumeração. Tais valores devem ser testados com o método `Enum.IsDefined` antes de serem atribuídos ao nome de uma variável de enumeração.

apontar ou referenciar um único tipo. Assim, os objetos ou valores de dados apontados ou referenciados são também de um tipo único. Em outras linguagens, como JavaScript, Python e Ruby, as variáveis são referências sem tipo para objetos ou valores de dados. Nesses casos, as matrizes ainda consistem em elementos de um único tipo, mas os elementos podem referenciar objetos ou valores de dados de tipos diferentes. Tais vetores ainda são homogêneos, porque os elementos da matriz são do mesmo tipo.

C# e Java 5.0 fornecem matrizes genéricas, ou seja, matrizes cujos elementos são referências a objetos, por meio de suas bibliotecas de classe. Isso é discutido na Seção 6.5.3.

6.5.1 Questões de projeto

As principais questões de projeto específicas às matrizes são:

- Que tipos são permitidos para índices?
- As expressões de índices em referências a elementos são verificadas em relação à faixa?
- Quando as faixas de índices são vinculadas?
- Quando ocorre a alocação da matriz?
- As matrizes multidimensionais irregulares ou retangulares são permitidas, ou ambas?
- As matrizes podem ser inicializadas quando têm seu armazenamento alocado?
- Que tipos de fatias são permitidas (caso sejam)?

Nas seções seguintes são discutidos exemplos de escolhas de projeto feitas para as matrizes das linguagens de programação mais comuns.

6.5.2 Matrizes e índices

Elementos específicos de uma matriz são referenciados por meio de um mecanismo sintático de dois níveis, em que a primeira parte é o nome agregado e a segunda é um seletor, possivelmente dinâmico, que consiste em um ou mais itens conhecidos como índices ou **subscritos**. Se todos os índices em uma referência são constantes, o seletor é estático; senão, ele é dinâmico. A operação de seleção pode ser pensada como um mapeamento do nome de uma matriz e o conjunto de valores de índices para um elemento no agregado. Na verdade, as matrizes são algumas vezes chamadas de **mapeamentos finitos**. Simbolicamente, esse mapeamento pode ser mostrado como

array_name(subscript_value_list) → element

A sintaxe das referências a uma matriz é bastante universal: o nome da matriz é seguido por uma lista de índices, envoltos em parênteses ou em colchetes. Em algumas linguagens que fornecem matrizes multidimensionais como matrizes de matrizes, cada índice aparece em seu próprio colchete. Um problema do uso de parênteses para envolver

» *nota histórica*

Os projetistas de Fortran pré-90 e de PL/I escolheram os parênteses para os índices de matrizes porque nenhum outro caractere adequado estava disponível na época. Cartões perfurados não incluíam caracteres para representar colchetes.

» *nota histórica*

Fortran I limitou o número de índices de matrizes a três porque, na época de seu projeto, a eficiência de execução era uma das principais preocupações. Os seus projetistas desenvolveram um método muito rápido para acessar os elementos de matrizes de até três dimensões, usando os três registradores de índices do IBM 704. Fortran IV foi implementado pela primeira vez em um IBM 7094, que tinha sete registradores de índices. Isso permitia aos projetistas de Fortran IV tolerar matrizes com até sete índices. A maioria das outras linguagens contemporâneas não determina esses limites.

as expressões de índices é que eles são usados também para envolver os parâmetros em chamadas de subprogramas; esse uso faz as referências a matrizes se parecerem exatamente com essas chamadas. Por exemplo, considere a seguinte sentença de atribuição em Ada:

```
Sum := Sum + B(I);
```

Como os parênteses são usados tanto para parâmetros de subprogramas quanto para índices de matrizes em Ada, tanto os leitores de programas como os compiladores são forçados a usar outra informação para determinar se `B(I)` nessa atribuição é uma chamada a uma função ou uma referência a um elemento de uma matriz. Isso resulta em uma redução na legibilidade.

Os projetistas de Ada escolheram especificamente os parênteses para envolver índices, de forma que houvesse uniformidade entre referências a matrizes e chamadas a funções em expressões, apesar dos problemas de legibilidade em potencial. Essa escolha foi feita em parte porque as referências a elementos de matrizes e chamadas a funções são mapeamentos. As referências mapeiam os índices para determinado elemento da matriz e as chamadas mapeiam os parâmetros reais para a definição da função e, por fim, um valor funcional.

A maioria das linguagens, exceto Fortran e Ada, usa colchetes para delimitar seus índices de matrizes.

Dois tipos distintos estão envolvidos em um tipo de matriz: o tipo de elemento e o tipo dos índices. Frequentemente, o tipo dos índices é inteiro.

As primeiras linguagens de programação não especificavam que as faixas de índices deveriam ser implicitamente verificadas. Erros de faixas em índices são comuns em programas; então, exigir a verificação de faixas é um fator importante na confiabilidade. Muitas linguagens contemporâneas também não especificam verificação de faixa de índices, mas Java, ML e C#, sim.

Índices em Perl são um pouco não usuais, pois, apesar de os nomes de todas as matrizes começarem com arroba (`@`), devido ao fato de os elementos de matrizes serem sempre escalares e os nomes dos escalares sempre começarem com cifrão (`$`), as referências a elementos de matrizes usam cifrões em vez de arrobas em seus nomes. Por exemplo, para a matriz `@list`, o segundo elemento é referenciado como `$list`[1].

É possível referenciar um elemento de uma matriz em Perl com um índice negativo – no caso, o valor do índice é um deslocamento a partir do fim da matriz. Por exemplo, se a matriz `@list` tem cinco elementos com índices 0..4, `$list`[-2] referencia o elemento com índice 3. Uma referência a um elemento não existente em Perl leva a um valor **`undef`**, mas nenhum erro é informado.

6.5.3 Vinculações de índices e categorias de matrizes

A vinculação do tipo do índice a uma variável matriz é normalmente estática, mas a faixa de valores do índice algumas vezes é vinculada dinamicamente.

Em algumas linguagens, o limite inferior da faixa de índices é implícito. Por exemplo, nas baseadas em C, o limite inferior de todas as faixas de índices é fixado em 0. Em outras, os limites inferiores podem ser especificados pelo programador.

Existem quatro categorias de matrizes, baseadas na vinculação das faixas de índices, na vinculação ao armazenamento e em a partir de onde o armazenamento é alocado. Os nomes das categorias indicam as escolhas de projeto para essas três questões. Nas primeiras três dessas categorias, uma vez que as faixas de índices são vinculadas e o armazenamento é alocado, elas permanecem fixas pelo tempo de vida de uma variável. É claro que, quando as faixas de índices são fixas, a matriz não pode modificar seu tamanho.

Uma **matriz estática** é aquela na qual as faixas de índices são vinculadas estaticamente e a alocação de armazenamento é estática (feita antes do tempo de execução). A vantagem das matrizes estáticas é a eficiência: nenhuma alocação ou liberação dinâmica é necessária. A desvantagem é que o armazenamento para a matriz é fixo por todo o tempo de execução do programa.

Uma **matriz dinâmica da pilha fixa** é aquela na qual as faixas de índices são vinculadas estaticamente, mas a alocação é feita em tempo de elaboração da declaração, durante a execução. A vantagem das matrizes dinâmicas da pilha fixas em relação às estáticas é a eficiência de espaço. Uma matriz grande em um subprograma pode usar o mesmo espaço que uma matriz grande em um subprograma diferente, desde que ambos os subprogramas não estejam ativos ao mesmo tempo. O mesmo é verdade caso as duas matrizes estejam em diferentes blocos não ativos ao mesmo tempo. A desvantagem é o tempo necessário para a alocação e a liberação.

Uma **matriz dinâmica do monte fixa** é similar a uma dinâmica da pilha fixa, no sentido de que as faixas de índices e a vinculação ao armazenamento são fixas após este ser alocado. As diferenças são que tanto as faixas de índices quanto as vinculações de armazenamento são feitas quando o programa de usuário requisitá-las durante a execução, e o armazenamento é alocado a partir do monte, em vez de a partir da pilha. A vantagem das matrizes dinâmicas do monte fixas é sua flexibilidade – o tamanho da matriz sempre se encaixa no problema. A desvantagem é o tempo de alocação a partir do monte, que é maior que o tempo de alocação a partir da pilha.

Uma **matriz dinâmica do monte** é aquela na qual a vinculação das faixas de índices e da alocação de armazenamento é dinâmica e pode mudar qualquer número de vezes durante seu tempo de vida. A vantagem das matrizes dinâmicas do monte em relação às outras é a flexibilidade: elas podem crescer e encolher durante a execução de um programa conforme a necessidade de mudanças de espaço. A desvantagem é que a alocação e a liberação levam mais tempo e podem acontecer muitas vezes durante a execução do programa. Exemplos das quatro categorias são dados nos parágrafos a seguir.

Matrizes declaradas em funções C e C++ que incluem o modificador **`static`** são estáticas.

Matrizes declaradas em funções C e C++ sem o especificador **`static`** são dinâmicas da pilha fixas.

C e C++ também fornecem matrizes dinâmicas do monte fixas. Funções da biblioteca padrão `malloc` e `free`, operações gerais de alocação e liberação no monte, respectivamente, podem ser usadas para matrizes em C. C++ usa os operadores **`new`** e **`delete`** para gerenciar o armazenamento no monte. Uma matriz é tratada como um ponteiro para uma coleção de células de armazenamento, onde o ponteiro pode ser indexado, como discutido na Seção 6.11.5.

Em Java, todas as matrizes não genéricas são dinâmicas do monte fixas. Uma vez criadas, elas mantêm as mesmas faixas de índices e o mesmo armazenamento. C# também fornece matrizes dinâmicas do monte fixas.

Objetos da classe `List` de C# são matrizes dinâmicas do monte genéricas. Esses objetos são criados sem elementos, como em

```
List<String> stringList = new List<String>();
```

Elementos são adicionados a esse objeto por meio do método `Add`, como em

```
stringList.Add("Michael");
```

O acesso aos elementos dessas matrizes é feito por meio de índices.

Java inclui uma classe genérica semelhante à `List` de C#, chamada `ArrayList`. Ela é diferente da `List` de C# porque o uso de índices não é suportado – para acessar os elementos, devem ser usados métodos de acesso (`get` e `set`).

Uma matriz em Perl pode ser aumentada por meio das operações `push` (inclui um ou mais elementos novos no fim da matriz) e `unshift` (inclui um ou mais elementos novos no início da matriz), ou da atribuição de um valor para a matriz, especificando um índice além do de maior valor da matriz. Uma matriz pode ser reduzida a nenhum elemento atribuindo-se a ela a lista vazia, `()`. O tamanho de uma matriz é definido como o maior índice acrescido de um.

Como Perl, JavaScript permite que as matrizes cresçam com os métodos `push` e `unshift` e encolham atribuindo-se a elas a lista vazia. Entretanto, índices negativos não são suportados.

Matrizes em JavaScript podem ser esparsas, ou seja, os valores dos índices não precisam ser adjacentes. Por exemplo, suponha que tenhamos uma matriz chamada `list` que tem 10 elementos com os índices `0..9`.[5] Considere a seguinte sentença de atribuição:

```
list[50] = 42;
```

Agora, `list` tem 11 elementos e tamanho 51. Os elementos com índices `11..49` não são definidos e, dessa forma, não exigem armazenamento. Uma referência a um elemento não existente em uma matriz JavaScript leva a **`undefined`**.

Matrizes em Python, Ruby e Lua podem ser aumentadas apenas por métodos para a adição de elementos ou concatenação com outras matrizes. Ruby e Lua suportam índices negativos, mas Python não. Em Python, Ruby e Lua, um elemento ou fatia de uma matriz pode ser apagado. Uma referência a um elemento não existente em Python resulta

[5] A faixa de índices poderia facilmente ter sido 1000 .. 1009.

em um erro em tempo de execução, enquanto uma referência similar em Ruby e Lua leva a **nil** e nenhum erro é informado.

Embora a definição de ML não inclua matrizes, sua implementação amplamente usada, SML/NJ, inclui.

O único tipo de coleção predefinido que faz parte de F# é a matriz (outros tipos de coleção são fornecidos por meio da .NET Framework Library). Essas matrizes são parecidas com as de C#. É incluída uma sentença **foreach** na linguagem para processamento de matrizes.

6.5.4 Inicialização de matrizes

Algumas linguagens fornecem os meios para inicializar matrizes no momento em que seu armazenamento é alocado. C, C++, Java e C# permitem inicialização de suas matrizes. Considere a seguinte declaração em C:

```
int list [] = {4, 5, 7, 83};
```

A matriz `list` é criada e inicializada com os valores `4`, `5`, `7` e `83`. O compilador também define o tamanho da matriz. Isso supostamente seria uma conveniência, mas tem seu custo. Ela efetivamente elimina a possibilidade de o sistema poder detectar alguns tipos de erros de programação, como deixar um valor de fora da lista por engano.

Conforme discutido na Seção 6.3.2, cadeias de caracteres em C e C++ são implementadas como matrizes (unidimensionais) de **char**. Essas matrizes podem ser inicializadas com constantes que representam cadeias, como em

```
char name [] = "freddie";
```

A matriz `name` terá oito elementos, visto que todas as cadeias são terminadas com um caractere nulo (zero), fornecido implicitamente pelo sistema para constantes do tipo cadeia.

Matrizes de cadeias em C e C++ também podem ser inicializadas com literais do tipo cadeia. Por exemplo,

```
char *names [] = {"Bob", "Jake", "Darcie"};
```

Esse exemplo ilustra a natureza dos literais de caracteres em C e C++. No exemplo anterior, que mostra um literal do tipo cadeia usado para inicializar a matriz de **char** `name`, o literal é considerado uma matriz de **char**. Mas, no último exemplo (`names`), os literais são considerados ponteiros para caracteres; então, a matriz é de ponteiros para caracteres. Por exemplo, `names[0]` é um ponteiro para a letra '`B`' na matriz de caracteres literal que contém os caracteres '`B`', '`o`', '`b`', e o caractere nulo (null).

Em Java, uma sintaxe similar é usada para definir e inicializar uma matriz de referências a objetos `String`. Por exemplo,

```
String[] names = ["Bob", "Jake", "Darcie"];
```

6.5.5 Operações de matrizes

Uma operação de matriz atua nesta como uma unidade. As mais comuns são a atribuição, a concatenação, a comparação por igualdade e diferença e as fatias, as quais são discutidas separadamente na Seção 6.5.7.

As linguagens baseadas em C não fornecem operações de matrizes, exceto por meio de métodos em Java, C++ e C#. Perl oferece suporte à atribuição de matrizes, mas não para comparações.

As matrizes em Python são chamadas de listas, apesar de terem todas as características das matrizes dinâmicas. Como os objetos podem ser de qualquer tipo, essas matrizes são heterogêneas. Python fornece atribuição de matrizes, apesar de ser apenas uma mudança de referência. Python também tem operações para concatenação de matrizes (+) e para verificar se um elemento pertence à matriz (**`in`**). Ela inclui dois operadores de comparação: um que determina se duas variáveis referenciam o mesmo objeto (**`is`**) e outro que compara em relação à igualdade (==) todos os objetos correspondentes nos objetos referenciados, independentemente de quão profundamente estejam aninhados.

Como em Python, os elementos das matrizes de Ruby são referências a objetos. E, como em Python, quando um operador == é usado entre duas matrizes, o resultado é verdadeiro somente se elas têm o mesmo tamanho e os elementos correspondentes são iguais. As matrizes de Ruby podem ser concatenadas com um método `Array`.

F# contém muitos operadores de matrizes em seu módulo `Array`. Entre eles estão `Array.append`, `Array.copy` e `Array.length`.

Matrizes e suas operações estão no coração de APL; ela é a linguagem de processamento de matrizes mais poderosa já desenvolvida. Devido ao seu relativo anonimato e à sua pouca influência em linguagens subsequentes, entretanto, apresentamos apenas um relance de suas operações de matrizes.

Em APL, as quatro operações aritméticas básicas são definidas para vetores (matrizes de dimensão única) e para matrizes, bem como para operandos escalares. Por exemplo,

```
A + B
```

é uma expressão válida, independentemente de `A` e `B` serem variáveis escalares, vetores ou matrizes.

APL inclui uma coleção de operadores unários para vetores e matrizes, alguns dos quais são os seguintes (onde `V` é um vetor e `M` é uma matriz):

ϕV	inverte os elementos de V
ϕM	inverte as colunas de M
θM	inverte as linhas de M
ØM	transpõe M (suas linhas viram suas colunas e vice-versa)
÷M	inverte M

APL também inclui diversos operadores especiais que recebem outros operadores como operandos. Um deles é o operador de produto interno, especificado com um ponto (.). Ele recebe dois operandos, os quais são operadores binários. Por exemplo,

```
+.×
```

é um novo operador que recebe dois argumentos, vetores ou matrizes. Ele primeiro multiplica os elementos correspondentes de dois argumentos e, então, soma os resultados. Por exemplo, se `A` e `B` são vetores,

```
A × B
```

é o produto interno matemático de `A` e `B` (um vetor dos produtos dos elementos correspondentes de `A` e `B`). A sentença

```
A +.× B
```

é a soma do produto interno de `A` e `B`. Se `A` e `B` são matrizes, essa expressão especifica a multiplicação das matrizes `A` e `B`.

Os operadores especiais de APL são, na verdade, formas funcionais, descritas no Capítulo 15.

6.5.6 Matrizes retangulares e irregulares

Uma **matriz retangular** é uma matriz multidimensional na qual todas as linhas têm o mesmo número de elementos e todas as colunas têm o mesmo número de elementos. Matrizes retangulares modelam tabelas retangulares precisamente.

Uma **matriz irregular** é aquela na qual o tamanho das linhas não precisa ser o mesmo. Por exemplo, uma matriz irregular pode ser composta de três linhas, uma com cinco elementos, uma com sete e outra com 12. Isso também se aplica às colunas e às dimensões superiores. Então, se existir uma terceira dimensão (camadas), cada camada pode ter um número diferente de elementos. Matrizes irregulares são possíveis quando as multidimensionais são matrizes de matrizes. Por exemplo, uma matriz poderia aparecer como uma matriz de matrizes de uma dimensão.

C, C++ e Java oferecem suporte para matrizes irregulares, mas não para matrizes regulares. Nessas linguagens, uma referência a um elemento de uma matriz multidimensional usa um par de colchetes separado para cada dimensão. Por exemplo,

```
myArray[3][7]
```

C# e F# suportam matrizes retangulares e irregulares. Para matrizes retangulares, todas as expressões de índices em referências a elementos são colocadas em um único par de colchetes. Por exemplo,

```
myArray[3, 7]
```

6.5.7 Fatias

A **fatia** de uma matriz é alguma subestrutura dela. Por exemplo, se `A` é uma matriz, então a sua primeira linha é uma possível fatia, assim como sua última linha e sua primeira coluna. É importante notar que uma fatia não é um novo tipo de dados. Em vez disso, é um mecanismo para referenciar parte de uma matriz como uma unidade. Se as matrizes não podem ser manipuladas como unidade em uma linguagem, esta não tem uso para fatias.

Considere as seguintes declarações em Python:

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
```

Lembre-se de que o limite inferior padrão para matrizes em Python é 0. A sintaxe de referência a uma fatia em Python é um par de expressões numéricas separadas por dois pontos. A primeira é o primeiro índice da fatia; a segunda é o primeiro índice após o último índice na fatia. Logo, `vector[3:6]` é uma matriz de três elementos com os elementos do quarto ao sexto de `vector` (os elementos com os índices `3`, `4` e `5`). Uma linha de uma matriz é especificada ao darmos apenas um índice. Por exemplo, `mat[1]` se refere à segunda linha de `mat`; uma parte de uma linha pode ser especificada com a mesma sintaxe que a parte de uma matriz de uma dimensão. Por exemplo, `mat[0][0:2]` se refere ao primeiro e ao segundo elementos da primeira linha de `mat`, que é `[1,2]`.

Python também suporta fatias mais complexas de matrizes. Por exemplo, `vector[0:7:2]` referencia cada elemento de vector, até, mas não incluindo, o elemento com índice 7, iniciando com o índice `0`, que é `[2, 6, 10, 14]`.

Perl suporta fatias de duas formas, uma lista de subíndices específicos ou uma faixa de índices. Por exemplo,

```
@list[1..5] = @list2[3, 5, 7, 9, 13];
```

Note que referências a fatias usam nomes de matrizes, não nomes escalares, porque fatias são matrizes (não escalares).

Ruby suporta fatias com o método `slice` de seu objeto `Array`, o qual pode receber três formas de parâmetros. Um único parâmetro com uma expressão inteira é interpretado como um índice, o que faz `slice` retornar o elemento com o índice. Se forem passados dois parâmetros com expressões inteiras para `slice`, o primeiro será interpretado como o índice de início e o segundo como o número de elementos na fatia. Por exemplo, suponha que `list` seja definida como:

```
list = [2, 4, 6, 8, 10]
```

`list.slice(2, 2)` retorna `[6, 8]`. A terceira forma de parâmetros para `slice` é uma faixa, a qual tem a forma de uma expressão inteira, dois pontos e uma segunda expressão inteira. Com um parâmetro de faixa, `slice` retorna uma matriz do elemento com a faixa de índices. Por exemplo, `list.slice(1..3)` retorna `[4, 6, 8]`.

6.5.8 Avaliação

Matrizes são incluídas em praticamente todas as linguagens de programação. Os principais avanços desde sua introdução em Fortran I foram as fatias e as matrizes dinâmicas. Conforme discutido na Seção 6.6, os últimos avanços em matrizes são as matrizes associativas.

6.5.9 Implementação de matrizes

Implementar matrizes requer um esforço adicional considerável em tempo de compilação, em relação à implementação de tipos primitivos. O código para permitir o acesso

aos elementos de uma matriz deve ser gerado em tempo de compilação. Em tempo de execução, esse código deve ser executado para produzir endereços de elementos. Não existe uma maneira de computar previamente o endereço a ser acessado por uma referência, como em

```
list[k]
```

Uma matriz de uma dimensão é implementada como uma lista de células de memória adjacentes. Suponha que a matriz `list` tenha com um limite inferior para a faixa de índices de valor `0`. A função de acesso para `list` é geralmente construída da seguinte forma

endereço(`list[k]`) = endereço(`list[0]`) + `k` * tamanho_do_elemento

onde o primeiro operando da adição é a parte constante da função de acesso e o segundo é a parte variável.

Se o tipo do elemento é vinculado estaticamente e a matriz é estaticamente vinculada ao armazenamento, o valor da parte constante pode ser calculado antes da execução. Entretanto, as operações de adição e de multiplicação devem ser efetuadas em tempo de execução.

A generalização dessa função de acesso para um limite inferior arbitrário é

endereço(`list[k]`) = endereço(`list`[limite_inferior]) +
((`k` – limite_inferior) * tamanho_do_elemento)

O descritor em tempo de compilação para matrizes de uma dimensão pode ter a forma mostrada na Figura 6.4. O descritor inclui as informações necessárias para construir a função de acesso. Se a verificação em tempo de execução das faixas de índices não é feita e os atributos são todos estáticos, apenas a função de acesso é necessária durante a execução; nenhum descritor é necessário. Se a verificação em tempo de execução da faixa de índices é feita, essas faixas talvez precisem ser armazenadas em um descritor em tempo de execução. Se as faixas de índices de determinado tipo de matriz são estáticas, então elas podem ser incorporadas no código que faz a verificação, eliminando a necessidade do descritor em tempo de execução. Se qualquer uma das entradas do descritor

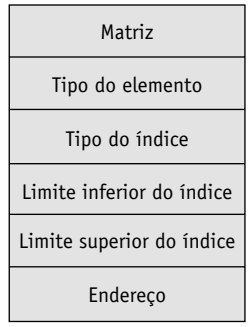


FIGURA 6.4
Descritor em tempo de compilação para matrizes de uma dimensão.

for vinculada dinamicamente, essas partes do descritor devem ser mantidas em tempo de execução.

Matrizes multidimensionais verdadeiras, ou seja, aquelas que não são matrizes de matrizes, são mais complexas de implementar do que as de uma dimensão, apesar de a extensão para mais dimensões ser direta. A memória em hardware é linear – apenas uma sequência simples de bytes. Então, valores de tipos de dados que têm duas ou mais dimensões devem ser mapeados para a memória de uma única dimensão. Existem duas maneiras pelas quais as matrizes multidimensionais podem ser mapeadas para uma dimensão: ordem principal de linha e ordem principal de coluna (não empregada em nenhuma linguagem comumente usada). Na **ordem principal de linha**, os elementos da matriz que têm em seu primeiro índice o valor do limite inferior daquele índice são armazenados primeiro, seguidos dos elementos do segundo valor do primeiro índice e assim por diante. A matriz é armazenada pelas linhas. Por exemplo, se a matriz tivesse os valores

3 4 7
6 2 5
1 3 8

ela seria armazenada por meio da ordem principal de linha, como

3, 4, 7, 6, 2, 5, 1, 3, 8

A função de acesso para uma matriz multidimensional é o mapeamento de seu endereço base e um conjunto de valores de índices para o endereço em memória do elemento especificado pelos valores de índices. A função de acesso para matrizes multidimensionais armazenadas em ordem principal de linha pode ser desenvolvida como segue. Em geral, o endereço de um elemento é o endereço base da estrutura, mais o tamanho do elemento multiplicado pelo número de elementos que o precedem na estrutura. Para uma matriz em ordem principal de linha, o número de elementos que precede um elemento

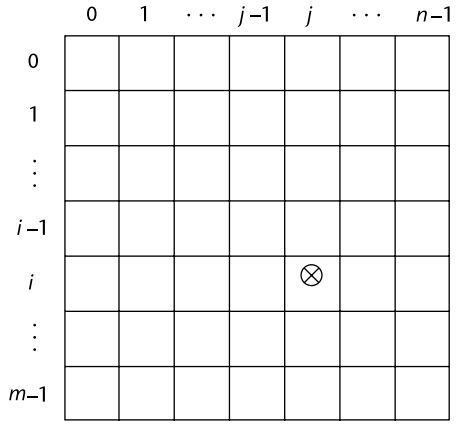


FIGURA 6.5
A localização do elemento `[i, j]` em uma matriz.

é o número de linhas acima dele multiplicado pelo tamanho de uma linha, somado ao número de elementos à esquerda dele em sua linha. Isso está ilustrado na Figura 6.5, na qual supomos que todos os limites inferiores de índices são zero.

Para obter um valor de endereço real, o número de elementos que precede o elemento desejado deve ser multiplicado pelo tamanho desse elemento. Agora, a função de acesso pode ser escrita como

localização(a[i, j]) = endereço de a[0, 0]
+ ((((número de linhas acima da i-ésima linha) * (tamanho de uma linha))
+ (número de elementos à esquerda da j-ésima coluna)) *
tamanho do elemento)

Como o número de linhas acima da i-ésima linha é i e o número de elementos à esquerda da j-ésima coluna é j, temos que

localização(a[i, j]) = endereço de a[0, 0] + ((((i * n) + j) *
tamanho_do_elemento)

onde n é o número de elementos por linha. O primeiro termo é a parte constante e o último é a parte variável.

A generalização para limites inferiores arbitrários resulta na seguinte função de acesso:

localização(a[i, j]) = endereço de a[li_linha, li_col]
= (((i – li_linha) * n) + (j – li_col)) * tamanho_do_elemento

onde li_linha é o limite inferior das linhas e li_col é o limite inferior das colunas. Isso pode ser reorganizado para a forma

localização(a[i, j]) = endereço de a[li_linha, li_col]
– (((li_linha * n) + li_col) * tamanho_do_elemento)
+ (((i * n) + j) * tamanho_do_elemento)

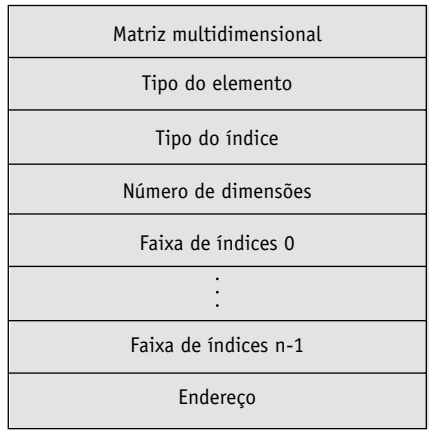


FIGURA 6.6
Um descritor em tempo de compilação para uma matriz multidimensional.

onde os dois primeiros termos são a parte constante e o último é a parte variável. Isso pode ser facilmente generalizado para um número arbitrário de dimensões.

Para cada dimensão de uma matriz, uma instrução de adição e uma de multiplicação são necessárias para a função de acesso. Logo, acessos a elementos de matrizes com diversos índices são dispendiosos. O descritor em tempo de compilação para uma matriz multidimensional é mostrado na Figura 6.6.

6.6 MATRIZES ASSOCIATIVAS

Uma **matriz associativa** é uma coleção não ordenada de elementos de dados, indexados por um número igual de valores chamados de **chaves**. No caso das matrizes não associativas, os índices nunca precisam ser armazenados (por causa de sua regularidade). As chaves definidas pelos usuários, entretanto, devem ser armazenadas na estrutura. Então, cada elemento de uma matriz associativa é, de fato, um par de entidades, uma chave e um valor. Usamos o projeto de matrizes associativas de Perl para ilustrar essa estrutura de dados. Matrizes associativas também são suportadas diretamente por Python, Ruby e Lua, assim como pelas bibliotecas de classe padrão de Java, C++, C# e F#.

A única questão de projeto específica para matrizes associativas é o formato das referências aos seus elementos.

6.6.1 Estrutura e operações

Em Perl, as matrizes associativas são chamadas de **dispersões** (*hashes*), porque na implementação seus elementos são armazenados e obtidos com funções de dispersão (*hash functions*). O espaço de nomes para dispersões em Perl é diferente: cada variável de dispersão deve começar com um sinal de percentual (`%`). Cada elemento de dispersão consiste em duas partes: uma chave, que é uma cadeia, e um valor, que é um escalar (número, cadeia ou referência). Dispersões podem ter valores literais atribuídos a elas com a sentença de atribuição, como em

```
%salaries = ("Gary" => 75000, "Perry" => 57000,
             "Mary" => 55750, "Cedric" => 47850);
```

Valores de elementos individuais são referenciados com uma notação similar à das matrizes em Perl. O valor da chave é colocado entre chaves, e o nome da dispersão é substituído por um nome de variável escalar, que é o mesmo, exceto pelo primeiro caractere. Apesar de dispersões não serem escalares, as partes que representam valores nos elementos de dispersão o são; então, referências a valores de elementos de dispersão usam nomes escalares. Lembre-se de que nomes de variáveis escalares começam com cifrão (`$`). Então, uma atribuição de `58850` ao elemento `%salaries` com a chave `"Perry"` apareceria como segue:

```
$salaries{"Perry"} = 58850;
```

Um novo elemento é adicionado por meio da mesma forma da sentença de atribuição. Um elemento pode ser removido da dispersão com o operador **`delete`**, como no seguinte caso:

Lua

ROBERTO IERUSALIMSCHY

Roberto Ierusalimschy é um dos criadores da linguagem de *scripting* Lua, muito usada em aplicações de desenvolvimento de jogos e sistemas embarcados. Ele é professor associado no Departamento de Ciência da Computação da Pontifícia Universidade Católica do Rio de Janeiro, no Brasil. (Para mais informações sobre Lua, visite www.lua.org.)

Como e quando você se envolveu com computação pela primeira vez Antes de entrar para a universidade, em 1978, não tinha ideia do que era computação. Lembro que tentei ler um livro sobre programação em Fortran, mas não passei do capítulo inicial sobre a definição de variáveis e constantes.

Em meu primeiro ano na universidade, cursei uma disciplina de Programação Básica em Fortran. Na época, executávamos nossos trabalhos de programação em um computador de grande porte IBM 370. Tínhamos de perfurar cartões com nossos códigos, envolver o conjunto de cartões com alguns cartões JCL fixos e entregá-los a um operador. Algum tempo depois (poucas horas) recebíamos uma listagem com os resultados, que normalmente eram apenas erros de compilação.

Logo após esse período, um amigo meu trouxe do exterior um microcomputador com uma CPU Z80, com 4Kbytes de memória. Começamos a fazer todos os tipos de programas para essa máquina, tudo em linguagem de montagem – ou, mais exatamente, em código de máquina, já que ele não tinha um montador. Escrevíamos nossos programas em linguagem de montagem e, depois, os traduzíamos manualmente para hexadecimal a fim de colocá-los em memória para executá-los.

Desde então, fui fisgado.

Nos últimos 25 anos, poucas linguagens de programação projetadas em ambientes acadêmicos foram bem-sucedidas. Apesar de você ser um acadêmico, Lua foi projetada para aplicações bastante práticas. Você a considera uma linguagem acadêmica ou industrial? Lua é certamente uma linguagem industrial, mas com um "sotaque" acadêmico. Lua foi criada para duas aplicações industriais, e tem sido usada em aplicações industriais desde seu início. Tentamos ser bastante pragmáticos em seu projeto. Entretanto, exceto por sua primeira versão, nunca estivemos sob a pressão típica de um ambiente industrial. Sempre tivemos o luxo de escolher quando lançar uma nova versão ou de escolher se aceitaríamos as exigências dos usuários. Isso nos deu alguma margem de manobra que outras linguagens não desfrutaram.

Mais recentemente, fizemos algumas pesquisas acadêmicas com Lua. Mas é um longo processo mesclar esses resultados acadêmicos com a distribuição oficial; esses resultados têm tido pouco impacto direto sobre a linguagem. Existem algumas boas exceções, no entanto, como a máquina virtual baseada em registradores e as "tabelas efêmeras" (que aparecerão em Lua 5.2).

Você disse que Lua evoluiu aos poucos, em vez de ter sido projetada. Você pode comentar sobre o que quis dizer e quais são as vantagens dessa estratégia? Quis dizer que as partes mais importantes da Lua não estavam presentes em sua primeira versão. A linguagem iniciou muito pequena e simples e obteve diversos de seus recursos relevantes à medida que evoluiu.

Antes de falar sobre as vantagens (e as desvantagens) dessa estratégia, quero esclarecer que não a escolhemos. Nunca pensamos, "vamos desenvolver uma nova linguagem". Apenas aconteceu.

Eu acho que uma das partes mais difíceis ao projetar uma linguagem é prever como diferentes mecanismos interagirão no dia a dia. Ao fazer uma linguagem crescer – isto é, criando-a peça por peça –, você pode evitar muitos desses problemas de interação, na medida em que é possível pensar sobre cada novo recurso apenas após o resto da linguagem já estar pronto e ter sido testado por usuários reais em aplicações reais.

É claro, essa estratégia apresenta uma grande desvantagem também: pode-se chegar a um ponto no qual um recurso extremamente necessário é incompatível com o que você já tem.

Lua foi modificada de diversas maneiras desde que foi lançada pela primeira vez, em 1994. Você disse que em alguns momentos se arrependeu de não ter incluído um tipo booleano na Lua. Por que você não adicionou um? Isso pode soar engraçado, mas do que realmente sentimos falta foi do valor "falso"; não tínhamos uso para um valor "verdadeiro".

Como em Lisp original, Lua tratou `nil` como o valor falso e todo o resto como verdadeiro. O problema é que nil também representa uma variável não inicializada.

Não existe uma maneira de distinguir uma variável não inicializada de uma variável falsa. Precisávamos de um valor falso para que essa distinção fosse possível, em termos acadêmicos. Mas o valor verdadeiro era inútil; 1 ou qualquer outra constante era boa o suficiente.

Esse é um exemplo típico de onde nosso pensamento "industrial" entrava em conflito com nossa visão "acadêmica". Uma mente pragmática adicionaria o tipo booleano sem pensar duas vezes. Mas, em termos acadêmicos, estávamos chateados com essa deselegância. No fim, o lado pragmático venceu, mas levou algum tempo.

Quais eram os recursos mais importantes de Lua, além do pré-processador, que posteriormente se tornaram reconhecidos como problemas e foram removidos da linguagem? Não me lembro de outros problemas significativos. Removemos diversos recursos de Lua, mas na maioria dos casos porque foram superados por um novo recurso, normalmente "melhor" em algum sentido. Isso aconteceu com métodos de etiquetagem – *tag methods* – (substituídos pelos metamétodos), referências fracas na API C (substituídas por tabelas fracas) e *upvalues* (substituídos pelo uso de escopo léxico apropriado).

Quando um novo recurso de Lua que romperia a compatibilidade com as versões anteriores da linguagem é considerado, como é tomada a decisão de implementá-lo ou não? Essas decisões são sempre difíceis. Primeiro, tentamos encontrar algum outro formato que possa evitar ou ao menos reduzir a incompatibilidade. Se isso não for possível, tentamos fornecer maneiras fáceis de contornar a incompatibilidade. (Por exemplo, se removemos uma função da biblioteca principal, podemos fornecer uma implementação separada que o programador pode incorporar em seu código.) Tentamos ainda medir qual a possibilidade de detectar e corrigir a incompatibilidade. O fato de um novo recurso criar erros de sintaxe (por exemplo, uma nova palavra reservada) não é tão ruim; podemos até mesmo fornecer uma ferramenta automática para corrigir o código antigo. Entretanto, se o novo recurso produzir erros sutis (por exemplo, uma função já existente retornar um resultado diferente), o consideramos inaceitável.

Os métodos de iteração, como aqueles de Ruby, foram considerados para Lua, em vez da sentença for que foi adicionada? Que considerações levaram à escolha? Eles não só foram considerados, como foram na verdade implementados! Desde a versão 3.1 (1998), Lua tem uma função "`foreach`", que aplica uma função para todos os pares em uma tabela. De maneira similar, com "`gsub`" é fácil aplicar uma dada função para cada caractere em uma cadeia.

Em vez de um mecanismo "de bloco" especial para o corpo do iterador, Lua tem usado funções de primeira classe para a tarefa. Veja o exemplo a seguir:

```
—'t' é uma tabela de nomes para valores
— o laço de repetição a seguir imprime
todas as chaves com valores maiores que
10
foreach(t, function(key, value)
  if value > 10 then print(key) end
end)
```

Entretanto, quando implementamos iteradores pela primeira vez, as funções em Lua não tinham escopo léxico completo. Além disso, a sintaxe é um pouco pesada (macros ajudariam) e as sentenças de saída (*break* e *return*) são sempre confusas quando usadas dentro de blocos de iteração. Assim, por fim, decidimos pela sentença `for`.

Mas "iteradores verdadeiros" ainda são um projeto útil em Lua, ainda mais agora que as funções apresentam escopo léxico apropriado. Em meu livro sobre Lua, termino o capítulo sobre a sentença `for` com uma discussão a respeito de iteradores verdadeiros.

Você pode descrever brevemente o que quer dizer quando descreve Lua como uma linguagem de extensão extensível? Ela é uma "linguagem extensível" porque é fácil registrar novas funções e tipos definidos em outras linguagens. Portanto, é fácil estender a linguagem. De um ponto de vista mais concreto, é fácil de chamar C a partir de Lua.

É uma "linguagem de extensão" porque é fácil usá-la para estender uma aplicação, para transformá-la em uma linguagem de macro para a aplicação. (Isso é *scripting* em seu mais puro significado.) De um ponto de vista mais concreto, é fácil chamar Lua a partir de C.

As estruturas de dados evoluíram de matrizes, registros e dispersões para combinações desses elementos. Você pode estimar o quão significativas são as tabelas de Lua na evolução das estruturas de dados nas linguagens de programação? Não acho que as tabelas de Lua tenham tido qualquer relevância na evolução de outras linguagens. Talvez isso mude no futuro, mas não tenho certeza. Em minha opinião, a principal vantagem oferecida pelas tabelas da Lua é sua simplicidade, uma solução "tudo-em-um". Mas essa simplicidade tem seus custos: por exemplo, a análise estática de programas em Lua é muito difícil, porque as tabelas são usadas de forma muito genérica e onipresente. Cada linguagem tem suas prioridades.

```
delete $salaries{"Gary"};
```

A dispersão inteira pode ser esvaziada por meio da atribuição do literal vazio a ela, como a seguir:

```
@salaries = ();
```

O tamanho de uma dispersão em Perl é dinâmico: ele aumenta quando um elemento é adicionado e encolhe quando um elemento é apagado (e também quando ela é esvaziada por meio da atribuição do literal vazio). O operador `exists` retorna verdadeiro ou falso, dependendo de sua chave operanda ser um elemento na dispersão. Por exemplo,

```
if (exists $salaries{"Shelly"}) ... . .
```

O operador `keys`, quando aplicado a uma dispersão, retorna uma matriz com as chaves da dispersão. O operador `values` faz o mesmo para os valores da dispersão. O operador `each` itera sobre os pares de elemento de uma dispersão.

As matrizes associativas de Python, chamadas de **dicionários**, são similares às de Perl, exceto que todos os valores são referências a objetos. As matrizes associativas suportadas por Ruby são semelhantes às de Python, exceto que as chaves podem ser qualquer objeto,[6] em vez de apenas cadeias. Existe uma progressão a partir das dispersões em Perl, nas quais as chaves devem ser cadeias, para matrizes em PHP, nas quais as chaves podem ser inteiros ou cadeias; e para dispersões em Ruby, nas quais qualquer objeto pode ser uma chave.

As matrizes de PHP são tanto normais como associativas. Elas podem ser tratadas como uma ou outra. A linguagem fornece funções que permitem tanto acessos indexados quanto dispersos aos elementos. Uma matriz pode ter elementos criados com índices numéricos simples e com chaves de dispersão na forma de cadeias.

Em Lua, o tipo tabela é a única estrutura de dados. Uma tabela Lua é uma matriz associativa na qual tanto as chaves quanto os valores podem ser de quaisquer tipos. Uma tabela pode ser usada como uma matriz tradicional, como uma matriz associativa ou como um registro (estrutura). Quando usada como uma matriz tradicional ou como uma matriz associativa, são colocados colchetes em torno das chaves. Quando usada como um registro, as chaves são os nomes dos campos e as referências aos campos podem empregar a notação com pontos (nome_registro.nome_campo).

O uso das matrizes associativas de Lua como registros é discutido na Seção 6.7.

C# e F# suportam matrizes associativas por meio de uma classe .NET.

Uma matriz associativa é muito melhor do que uma matriz caso sejam necessárias buscas a elementos, porque a operação de dispersão implícita usada para acessar os elementos é muito eficiente. Além disso, as matrizes associativas são ideais quando os dados a serem armazenados formam pares, como no caso de nomes de funcionários e seus salários. Por outro lado, se cada elemento de uma lista deve ser processado, é mais eficiente usar uma matriz.

[6] Objetos que mudam não são boas chaves, pois as mudanças podem modificar o valor da função de dispersão. Logo, matrizes e dispersões nunca são usadas como chaves.

6.6.2 Implementação de matrizes associativas

A implementação das matrizes associativas de Perl é otimizada para buscas rápidas, mas ela também fornece uma reorganização relativamente rápida quando o crescimento da matriz exige isso. Um valor de dispersão de 32 bits é computado para cada entrada e armazenado com a entrada, apesar de uma matriz associativa usar inicialmente apenas uma pequena parte do valor de dispersão. Quando uma matriz associativa precisa ser expandida para além de seu tamanho inicial, a função de dispersão não precisa ser mudada; em vez disso, são usados mais bits do valor de dispersão. Quando isso acontece, apenas metade das entradas precisa ser movida. Assim, apesar de a expansão das matrizes associativas não ser gratuita, ela não é tão dispendiosa quanto se poderia esperar.

Os elementos em matrizes PHP são colocados na memória por meio de uma função de dispersão. Entretanto, todos os elementos são ligados na ordem em que foram criados. Essas ligações são usadas para o suporte a acesso iterativo aos elementos por meio das funções `current` e `next`.

6.7 REGISTROS

Um **registro** é um agregado de elementos de dados no qual os elementos individuais são identificados por nomes e acessados por meio de deslocamentos a partir do início da estrutura.

Em geral, nos programas existe a necessidade de modelar uma coleção de dados na qual os elementos individuais não são do mesmo tipo ou tamanho. Por exemplo, informações sobre um estudante universitário podem incluir seu nome, seu número de matrícula, sua média de notas no histórico e assim por diante. Um tipo de dados para tal coleção pode usar uma cadeia de caracteres para o nome, um inteiro para o número de matrícula, um ponto flutuante para a média de notas no histórico e assim por diante. Registros são projetados para esse tipo de necessidade.

Pode parecer que registros e matrizes heterogêneas são a mesma coisa, mas esse não é o caso. Todos os elementos de matrizes heterogêneas são referências para objetos de dados que residem em posições espalhadas, geralmente no monte. Os elementos de um registro são de tamanhos potencialmente diferentes e residem em posições de memória adjacentes.

Os registros têm feito parte de todas as linguagens de programação mais populares – exceto pelas versões anteriores à Fortran 90 – desde os anos 1960, quando foram introduzidos por COBOL. Em algumas linguagens que têm suporte para a programação orientada a objetos, classes de dados servem como registros.

Em C, C++ e C#, os registros são suportados por meio do tipo de dados **`struct`**. Em C++, estruturas são uma pequena variação das classes. Em C#, as estruturas também são relacionadas às classes, mas são um tanto diferentes. As estruturas em C# são tipos de valores alocados na pilha, ao contrário dos objetos de classe, os quais são tipos de referências alocados no monte. Estruturas em C++ e C# em geral são usadas como estruturas de encapsulamento, em vez de como estruturas de dados. Elas são discutidas com mais detalhes em relação a essa capacidade no Capítulo 11. Estruturas também são incluídas em ML e F#.

Em Python e Ruby, registros podem ser implementados como dispersões, que podem elas próprias ser elementos de matrizes.

As seções a seguir descrevem como os registros são declarados ou definidos e como são feitas referências a campos dentro dos registros. Além disso, discutem as operações de registros comuns.

As questões de projeto a seguir são específicas aos registros:

- Qual é a forma sintática das referências a campos?
- Referências elípticas são permitidas?

6.7.1 Definições de registros

A diferença fundamental entre um registro e uma matriz é que os elementos, ou **campos**, do registro não são referenciados por índices. Em vez disso, são nomeados com identificadores, e são feitas referências para eles por meio desses identificadores. Outra diferença entre matrizes e registros é que, em algumas linguagens, os registros podem incluir uniões, as quais são discutidas na Seção 6.10.

O formato de COBOL de uma declaração de registro, que faz parte da divisão de dados de um programa COBOL, é ilustrado no exemplo a seguir:

```
01 EMPLOYEE-RECORD.
   02 EMPLOYEE-NAME.
      05 FIRST  PICTURE IS X(20).
      05 Middle PICTURE IS X(10).
      05 LAST  PICTURE IS X(20).
   02 HOURLY-RATE PICTURE IS 99V99.
```

O registro `EMPLOYEE-RECORD` consiste no registro `EMPLOYEE-NAME` e no campo `HOURLY-RATE`. Os numerais `01`, `02` e `05` que iniciam as linhas da declaração de registro são **números de nível**, os quais indicam, por seus valores relativos, a estrutura hierárquica do registro. Qualquer linha seguida por outra com um número de nível mais alto é, ela própria, um registro. A cláusula `PICTURE` mostra os formatos das posições de armazenamento de campo, com `X(20)` especificando 20 caracteres alfanuméricos e `99V99` especificando quatro dígitos decimais com o ponto decimal no meio.

Em Java e C#, os registros podem ser definidos como classes de dados, com registros aninhados como classes aninhadas. Membros de dados de tais classes servem como campos do registro.

Conforme mencionado, as matrizes associativas de Lua podem ser convenientemente usadas como registros. Por exemplo, considere a declaração:

```
employee.name = "Freddie"
employee.hourlyRate = 13.20
```

Essas sentenças de atribuição criam uma tabela (registro) chamada `employee` com dois elementos (campos), chamados `name` e `hourlyRate`, ambos inicializados.

6.7.2 Referências a campos de registros

Referências aos campos individuais dos registros são especificadas sintaticamente por métodos distintos, dois dos quais nomeiam o campo desejado e os registros que o envolvem. As referências a campos em COBOL têm a forma

nome_do_campo OF nome_do_registro _1 OF . . . OF nome_do_registro_n

onde o primeiro registro nomeado é o menor registro ou o mais interno que contém o campo. O próximo nome de registro na sequência é o do que contém o registro anterior e assim por diante. Por exemplo, o atributo `Middle` no registro de exemplo em COBOL, mostrado anteriormente, pode ser referenciado com

```
Middle OF EMPLOYEE-NAME OF EMPLOYEE-RECORD
```

A maioria das outras linguagens usa **notação por pontos** para referências a campos, onde os componentes da referência são conectados por pontos. Nomes em notação por pontos têm a ordem oposta das referências em COBOL: eles usam primeiro o nome do maior registro que envolve os outros e o nome do campo por último. Por exemplo, se `Middle` fosse um campo no registro `Employee_Name` incorporado ao registro `Employee_Record`, ele seria referenciado com o seguinte:

```
Employee_Record.Employee_Name.Middle
```

Referências a elementos em uma tabela Lua podem aparecer na sintaxe de referências a campos de registros, conforme visto nas sentenças de atribuição da Seção 6.7.1. Tais referências também podem ter a forma de elementos de tabela normais – por exemplo, `employee["name"]`.

Uma **referência completamente qualificada** a um campo de um registro é uma referência em que todos os nomes de registro intermediários, desde o que envolve todos os outros até o campo específico, são nomeados. No exemplo em COBOL acima, a referência ao campo está completamente qualificada. Como uma alternativa para as referências qualificadas, COBOL permite **referências elípticas** aos campos de registro. Nessas, o campo é nomeado, mas qualquer um ou todos os nomes de registros que o envolvem podem ser omitidos, desde que a referência resultante seja não ambígua no ambiente de referenciamento. Por exemplo, `FIRST`, `FIRST OF EMPLOYEE-NAME` e `FIRST OF EMPLOYEE-RECORD` são referências elípticas para o primeiro nome de funcionário no registro COBOL declarado acima. Apesar de as referências elípticas serem uma conveniência para o programador, elas exigem que o compilador tenha estruturas de dados e procedimentos elaborados para identificar corretamente o campo referenciado. De certa forma, elas também prejudicam a legibilidade.

6.7.3 Avaliação

Os registros são tipos de dados valiosos em linguagens de programação. O projeto de tipos registro é direto e seu uso é seguro.

Registros e matrizes estão intimamente relacionados às formas estruturais e é interessante compará-los. Matrizes são usadas quando todos os valores de dados têm o mesmo tipo e/ou são processados da mesma forma. Esse processamento é feito facilmente quando existe uma forma sistemática de sequenciamento ao longo da estrutura. Tal processamento é mais bem suportado usando-se índices dinâmicos como método de endereçamento.

Registros são usados quando a coleção de valores de dados é heterogênea e os campos diferentes não são processados da mesma maneira. Além disso, os campos de um

registro normalmente não precisam ser processados em uma ordem específica. Nomes de campos são como índices literais ou constantes. Como são estáticos, fornecem um acesso muito eficiente aos campos. Índices dinâmicos poderiam ser usados para acessar campos de registro, mas isso proibiria a verificação de tipos e seria mais lento.

Registros e matrizes representam métodos bem pensados e eficientes de satisfazer duas aplicações separadas, mas relacionadas, de estruturas de dados.

6.7.4 Implementação de registros

Os campos dos registros são armazenados em posições de memória adjacentes. Mas, como o tamanho dos campos não é necessariamente o mesmo, o método de acesso usado para matrizes não é usado para registros. Em vez disso, o endereço de deslocamento, relativo ao início do registro, é associado a cada campo. Todos os acessos a campos são manipulados com tais deslocamentos. O descritor em tempo de compilação para um registro tem a forma geral mostrada na Figura 6.7. Descritores em tempo de execução para registros são desnecessários.

6.8 TUPLAS

Uma tupla é um tipo de dados semelhante a um registro, exceto que os elementos não são nomeados.

Python inclui um tipo de tupla imutável. Se uma tupla precisa ser modificada, ela pode ser convertida em uma matriz com a função `list`. Após a mudança, ela pode ser convertida novamente em uma tupla com a função `tuple`. Um uso de tuplas ocorre quando uma matriz precisa ser protegida em relação à escrita, como quando ela é en-

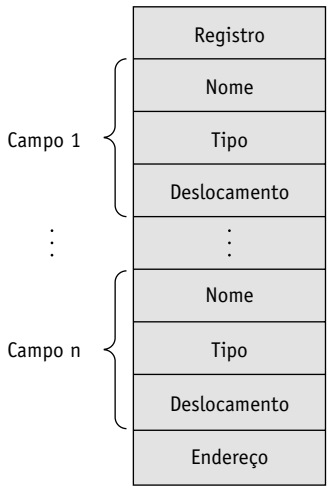


FIGURA 6.7
Um descritor em tempo de compilação para um registro.

viada como parâmetro para uma função externa e o usuário não quer que a função seja capaz de modificar o parâmetro.

As tuplas de Python são intimamente relacionadas às suas listas, exceto que as tuplas são imutáveis. Uma tupla é criada pela atribuição de uma literal de tupla, como no exemplo a seguir:

```
myTuple = (3, 5.8, 'apple')
```

Observe que os elementos de uma tupla não precisam ser do mesmo tipo.

Os elementos de uma tupla podem ser referenciados com indexação em colchetes, como no seguinte:

```
myTuple[1]
```

Isso referencia o primeiro elemento da tupla, pois os índices de tuplas começam em 1.

Tuplas podem ser concatenadas com o operador de adição (+) e podem ser apagadas com a sentença **del**. Existem ainda outros operadores e funções que atuam em tuplas.

ML contém um tipo de dados tupla. Uma tupla de ML deve ter pelo menos dois elementos, enquanto as tuplas de Python podem ser vazias ou conter um elemento. Assim como em Python, uma tupla de ML pode conter elementos de tipos mistos. A sentença a seguir cria uma tupla:

```
val myTuple = (3, 5.8, 'apple');
```

A sintaxe de um elemento de tupla é a seguinte:

```
#1(myTuple);
```

Isso referencia o primeiro elemento da tupla.

Um novo tipo de tupla pode ser definido na ML com uma declaração de tipo, como a seguinte:

```
type intReal = int * real;
```

Valores desse tipo consistem em um inteiro e um real.

F# também tem tuplas. Uma tupla é criada pela atribuição de um valor de tupla – o qual é uma lista de expressões separadas por vírgulas e delimitadas por parênteses – a um nome em uma sentença **let**. Se uma tupla tem dois elementos, eles podem ser referenciados com as funções `fst` e `snd`, respectivamente. Os elementos de uma tupla com mais de dois elementos geralmente são referenciados com um padrão de tupla no lado esquerdo de uma sentença **let**. Padrão de tupla é simplesmente uma sequência de nomes, um para cada elemento da tupla, com ou sem os parênteses delimitadores. Quando um padrão de tupla é o lado esquerdo de uma construção **let**, essa é uma atribuição múltipla. Por exemplo, considere as seguintes construções **let**:

```
let tup = (3, 5, 7);;
let a, b, c = tup;;
```

Isso atribui `3` a `a`, `5` a `b` e `7` a `c`.

Em Python, ML e F#, as tuplas são usadas para permitir que funções retornem vários valores.

6.9 LISTAS

Listas foram inicialmente suportadas na primeira linguagem de programação funcional, Lisp. Elas sempre fizeram parte das linguagens funcionais, mas nos últimos anos acabaram fazendo parte de algumas linguagens imperativas.

Em Scheme e Common Lisp, as listas são delimitadas por parênteses e os elementos não são separados por nenhuma pontuação. Por exemplo,

```
(A B C D)
```

Listas aninhadas têm a mesma forma, de modo que poderíamos ter

```
(A (B C) D)
```

Nessa lista, `(B C)` é uma lista aninhada dentro de outra lista, mais externa.

Dados e código têm a mesma forma sintática em Lisp e suas descendentes. Se a lista `(A B C)` é interpretada como código, essa é uma chamada para a função `A`, com parâmetros `B` e `C`.

Em Scheme, as operações de lista fundamentais são duas funções que separam listas e duas que constroem listas. A função `CAR` retorna o primeiro elemento de sua lista de parâmetros. Por exemplo, considere o seguinte exemplo:

```
(CAR '(A B C))
```

A aspa simples antes da lista de parâmetros impede que o interpretador considere a lista uma chamada para a função `A` com os parâmetros `B` e `C`, no caso em que ele a interpretaria. Essa chamada a `CAR` retorna `A`.

A função `CDR` retorna sua lista de parâmetros, menos o primeiro elemento. Por exemplo, considere o seguinte exemplo:

```
(CDR '(A B C))
```

Essa chamada de função retorna a lista `(B C)`.

Common Lisp também tem as funções `FIRST` (o mesmo que `CAR`), `SECOND`, . . ., `TENTH`, que retornam o elemento de sua lista de parâmetros especificados pelos seus nomes.

Em Scheme e Common Lisp, novas listas são construídas com as funções `CONS` e `LIST`. A função `CONS` recebe dois parâmetros e retorna uma nova lista, com seu primeiro parâmetro como primeiro elemento e seu segundo parâmetro como o restante dessa lista. Por exemplo, considere o seguinte:

```
(CONS 'A '(B C))
```

Essa chamada retorna a nova lista `(A B C)`.

A função LIST recebe qualquer número de parâmetros e retorna uma nova lista, com os parâmetros como seus elementos. Por exemplo, considere a seguinte chamada a LIST:

```
(LIST 'A 'B '(C D))
```

Essa chamada retorna a nova lista (A B (C D)).

ML possui listas e operações de listas, apesar de suas aparências não serem como as de Scheme. Listas são especificadas entre colchetes, com os elementos separados por vírgulas, como nesta lista de inteiros:

```
[5, 7, 9]
```

A lista vazia é [], que também pode ser especificada com nil.

A função CONS de Scheme é implementada como um operador infixo binário em ML, representada como ::. Por exemplo,

```
3 :: [5, 7, 9]
```

retorna a seguinte nova lista: [3, 5, 7, 9].

Os elementos de uma lista devem ser do mesmo tipo, então a lista a seguir seria inválida:

```
[5, 7.3, 9]
```

ML tem funções que correspondem a CAR e CDR de Scheme, chamadas de hd (*head* – cabeça) e tl (*tail* – cauda). Por exemplo,

```
hd [5, 7, 9] is 5
tl [5, 7, 9] is [7, 9]
```

Listas e operações em listas em Scheme e ML são discutidas mais detalhadamente no Capítulo 15.

Em F#, as listas estão relacionadas às de ML, com algumas diferenças notáveis. Os elementos de uma lista em F# são separados por pontos e vírgulas, e não por vírgulas, como ML. As operações hd e tl são as mesmas, mas são chamadas como métodos da classe List, como em List.hd [1; 3; 5; 7], que retorna 1. A operação CONS de F# é especificada como dois sinais de dois pontos, como em ML.

Python contém um tipo de lista que também serve como matriz para a linguagem. Ao contrário das listas de Scheme, Common Lisp, ML e F#, as listas de Python são mutáveis. Elas podem conter qualquer valor de dado ou objeto. Uma lista de Python é criada com uma atribuição de um valor de lista a um nome. Um valor de lista é uma sequência de expressões separadas por vírgulas e delimitadas com colchetes. Por exemplo, considere a sentença:

```
myList = [3, 5.8, "grape"]
```

Os elementos de uma lista são referenciados com índices em colchetes, como no exemplo a seguir:

```
x = myList[1]
```

Essa sentença atribui `5.8` a `x`. Os elementos de uma lista são indexados a partir de zero. Eles também podem ser atualizados por atribuição. Um elemento de uma lista pode ser apagado com `del`, como na sentença a seguir:

```
del myList[1]
```

Essa sentença remove o segundo elemento de `myList`.

Python inclui um poderoso mecanismo para criar matrizes, chamado **compreensões de lista**. A compreensão de lista é uma ideia derivada da notação de conjunto. Ela apareceu pela primeira vez na linguagem de programação funcional Haskell (consulte o Capítulo 15). A mecânica de uma compreensão de lista é uma função aplicada a cada um dos elementos de uma matriz e uma nova matriz construída a partir dos resultados. A sintaxe de uma compreensão de lista em Python é:

[expressão **for** var_iteração **in** matriz **if** condição]

Considere o seguinte exemplo:

```
[x * x for x in range(12) if x % 3 == 0]
```

A função **range** cria a matriz `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]`. A expressão condicional filtra todos os números da matriz não divisíveis por `3`. Então, a expressão eleva ao quadrado os números restantes. Os resultados são coletados em uma matriz, que é retornada. A compreensão de lista retorna a seguinte matriz:

```
[0, 9, 36, 81]
```

Fatias de listas também são suportadas em Python.

As compreensões de lista de Haskell têm a seguinte forma:

[corpo | qualificadores]

Por exemplo, considere a seguinte definição de lista:

```
[n * n | n <- [1..10]]
```

Ela define uma lista dos quadrados dos números de `1` a `10`.

F# inclui compreensões de lista, as quais, nessa linguagem, também podem ser usadas para criar matrizes. Por exemplo, considere a sentença:

```
let myArray = [|for i in 1 .. 5 -> (i * i) |];;
```

Essa sentença cria a matriz `[1; 4; 9; 16; 25]` e a chama de `myArray`.

Lembre-se, da Seção 6.5, que C# e Java suportam classes de coleção dinâmicas do monte genéricas, `List` e `ArrayList`, respectivamente. Na verdade, essas estruturas são listas.

6.10 UNIÕES

Uma **união** é um tipo cujas variáveis podem armazenar diferentes valores de tipos em vários momentos durante a execução de um programa. Como exemplo da necessidade de um

tipo união, considere uma tabela de constantes para um compilador, usada para armazenar as constantes encontradas em um programa que está sendo compilado. Um campo de cada entrada na tabela é para o valor da constante. Suponha que, para determinada linguagem sendo compilada, os tipos das constantes fossem: inteiro, ponto flutuante e booleano. Em termos de gerenciamento de tabela, seria conveniente se a mesma posição – um campo da tabela – pudesse armazenar um valor de qualquer um desses três tipos. Então, todos os valores constantes poderiam ser endereçados da mesma maneira. O tipo de tal posição é, em certo sentido, a união dos três tipos de valores que ela pode armazenar.

6.10.1 Questões de projeto

O problema da verificação de tipos para as uniões, discutido na Seção 6.12, é sua principal questão de projeto.

6.10.2 Uniões discriminadas versus uniões livres

C e C++ fornecem construções para representar uniões nas quais não existe um suporte da linguagem para a verificação de tipos. Em C e C++, a construção **`union`** é usada para especificar estruturas de união. As uniões nessas linguagens são chamadas de **uniões livres**, porque é permitido que os programadores tenham total liberdade em seu uso, em relação à verificação de tipos. Por exemplo, considere a seguinte união em C:

```
union flexType {
 int intEl;
 float floatEl;
};
union flexType el1;
float x ;
. . .
el1.intEl = 27;
x = el1.floatEl;
```

Essa última atribuição não é verificada em relação ao tipo, porque o sistema não pode determinar o tipo atual do valor de `el1`; então, ele atribui a representação em cadeia de bits de `27` para a variável **`float`** `x`, o que obviamente não faz sentido.

A verificação de tipos união requer que cada construção de união inclua um indicador de tipo. Tal indicador é chamado de **etiqueta** (*tag*) ou **discriminante**, e uma união com um discriminante é chamada de **união discriminada**. A primeira linguagem a fornecer uniões discriminadas foi ALGOL 68. Agora elas são suportadas por ML, Haskell e F#.

6.10.3 Uniões em F#

Em F#, uma união é declarada com uma sentença de tipo que usa operadores OU (`|`) para definir os componentes. Por exemplo, poderíamos ter o seguinte:

```
type intReal =
   | IntValue of int
   | RealValue of float;;
```

Nesse exemplo, `intReal` é a união. `IntValue` e `RealValue` são construtores. Valores de tipo `intReal` podem ser criados com os construtores como se fossem uma função, como nos exemplos a seguir:[7]

```
let ir1 = IntValue 17;;
let ir2 = RealValue 3.4;;
```

O acesso ao valor de uma união é feito com uma estrutura de casamento de padrões. Em F#, o casamento de padrões é especificado com a palavra reservada **`match`**. A forma geral da construção é a seguinte:

`match` padrão **`with`**
| lista_de_expressões$_1$ -> expressão$_1$
| . . .
| lista_de_expressões$_n$ -> expressão$_n$

O padrão pode ser qualquer tipo de dados. A lista de expressões pode incluir caracteres curingas (_) ou somente um caractere curinga. Por exemplo, considere a seguinte construção match:

```
let a = 7;;
let b = "grape";;
let x = match (a, b) with
   | 4, "apple" -> apple
   | _, "grape" -> grape
   | _ -> fruit;;
```

Para mostrar o tipo da união `intReal`, a seguinte função poderia ser usada:

```
let printType value =
    match value with
      | IntValue value -> printfn "It is an integer"
      | RealValue value -> printfn "It is a float";;
```

As linhas a seguir mostram chamadas a essa função e a saída:

```
printType ir1;;
It is an integer
printType ir2;;
It is a float
```

6.10.4 Avaliação

Uniões são construções potencialmente inseguras em algumas linguagens. Elas são uma das razões pelas quais C e C++ não são fortemente tipadas: essas linguagens não permitem que as referências para suas uniões sejam verificadas em relação aos seus tipos. Por

[7] A sentença **`let`** é usada para atribuir valores a nomes e para criar um escopo estático; os pontos e vírgulas duplos servem para finalizar sentenças quando o interpretador interativo de F# é usado.

outro lado, as uniões podem ser usadas seguramente, como em seu projeto nas linguagens ML, Haskell e F#.

Nem Java nem C# incluem uniões, o que pode ser um reflexo da crescente preocupação com a segurança em algumas linguagens de programação.

6.10.5 Implementação de uniões

Uniões são implementadas simplesmente por meio do uso do mesmo endereço para cada uma das variantes possíveis. É alocado um espaço de armazenamento suficiente para a maior variante.

6.11 PONTEIROS E REFERÊNCIAS

Um tipo **ponteiro** é aquele no qual as variáveis têm uma faixa de valores que consiste em endereços de memória e em um valor especial, **nil**. O valor nil não é um endereço válido e é utilizado para indicar que, em dado momento, um ponteiro não pode ser usado para referenciar uma célula de memória.

Ponteiros são projetados para dois tipos de uso. Primeiro, eles fornecem alguns dos recursos do endereçamento indireto, frequentemente usado em programação de linguagem de montagem. Segundo, fornecem uma maneira de gerenciar o armazenamento dinâmico. Um ponteiro pode ser usado para acessar uma posição em uma área onde o armazenamento é alocado dinamicamente, chamada de **monte** (*heap*).

As variáveis alocadas dinamicamente a partir do monte são chamadas de **variáveis dinâmicas do monte**. Em geral, elas não têm identificadores associados e, portanto, podem ser referenciadas apenas por variáveis dos tipos ponteiro ou referência. Variáveis sem nomes são chamadas de **variáveis anônimas**. É nessa última área de aplicação de ponteiros que surgem as questões de projeto mais importantes.

Diferentemente das matrizes e dos registros, os ponteiros não são tipos estruturados, apesar de serem definidos com um operador de tipo (* em C e C++). Além disso, são diferentes das variáveis escalares, porque são usados para referenciar alguma outra variável, em vez de para armazenar dados. Essas duas categorias de variáveis são chamadas de **tipos de referência** e **tipos de valor**, respectivamente.

Ambos os usos de ponteiros facilitam a escrita de programas em uma linguagem. Por exemplo, suponha que seja necessário implementar uma estrutura dinâmica como uma árvore binária em uma linguagem que não tem ponteiros nem armazenamento dinâmico. Isso exigiria que o programador fornecesse e mantivesse disponível uma coleção de nós de árvore, os quais provavelmente seriam implementados em matrizes paralelas. Além disso, o programador precisaria adivinhar o número máximo de nós necessários. Essa é uma maneira deselegante e passível de erros de trabalhar com árvores binárias.

Variáveis de referência, discutidas na Seção 6.11.6, são estritamente relacionadas aos ponteiros.

6.11.1 Questões de projeto

As principais questões de projeto particulares aos ponteiros são:

- Qual é o escopo e qual é o tempo de vida de uma variável do tipo ponteiro?
- Qual é o tempo de vida de uma variável dinâmica do monte (o valor referenciado por um ponteiro)?
- Os ponteiros são restritos em relação ao tipo de valores aos quais podem apontar?
- Os ponteiros são usados para gerenciamento de armazenamento dinâmico, endereçamento indireto ou ambos?
- A linguagem deveria suportar tipos ponteiro, tipos de referência ou ambos?

6.11.2 Operações de ponteiros

Linguagens que fornecem um tipo ponteiro normalmente incluem duas operações de ponteiros fundamentais: atribuição e desreferenciamento. A primeira modifica o valor de uma variável de ponteiro para algum endereço útil. Se as variáveis de ponteiro são usadas apenas para gerenciar armazenamento dinâmico, então o mecanismo de alocação, seja por operador ou por subprograma predefinido, serve para inicializá-las. Se os ponteiros são usados para endereçamento indireto a variáveis que não são dinâmicas do monte, então deve existir um operador explícito ou um subprograma predefinido para obter o endereço de uma variável, o qual pode ser então atribuído à variável de ponteiro.

Uma ocorrência de variável de ponteiro em uma expressão pode ser interpretada de duas maneiras. Primeiro, como uma referência ao conteúdo da célula de memória ao qual está vinculada, no caso de um ponteiro ser um endereço. Esse é exatamente o modo como uma variável que não é um ponteiro em uma expressão seria interpretada, apesar de que, nesse caso, seu valor provavelmente não fosse um endereço. Entretanto, o ponteiro também poderia ser interpretado como uma referência ao valor dentro da célula de memória apontado pela célula de memória à qual a variável de ponteiro está vinculada. Nesse caso, o ponteiro seria interpretado como uma referência indireta. O caso anterior é uma referência normal a um ponteiro; o último caso é o resultado de **desreferenciar** o ponteiro. Desreferenciar, o que leva uma referência a um nível de ação indireta, é a segunda operação fundamental dos ponteiros.

O desreferenciamento de ponteiros pode ser tanto explícito quanto implícito. Em muitas linguagens contemporâneas, isso ocorre somente quando explicitamente especificado. Em C++, ele é explicitamente especificado com o asterisco (`*`) como um operador unário prefixado. Considere o seguinte exemplo de desreferenciamento: se `ptr` é uma variável de ponteiro com o valor 7080 e a célula cujo endereço é 7080 tem o valor 206, então a atribuição

```
j = *ptr
```

configura `j` como 206. Esse processo é mostrado na Figura 6.8.

Quando os ponteiros apontam para registros, a sintaxe das referências para os campos desses registros varia entre as linguagens. Em C e C++, existem duas formas pelas quais um ponteiro para um registro pode ser usado a fim de referenciar um campo nesse registro. Se uma variável de ponteiro `p` aponta para um registro com um campo chamado `age`, `(*p).age` pode ser usado para referenciar esse campo. O operador `->`, quando usado entre um ponteiro para uma estrutura e um campo dessa estrutura, combina o

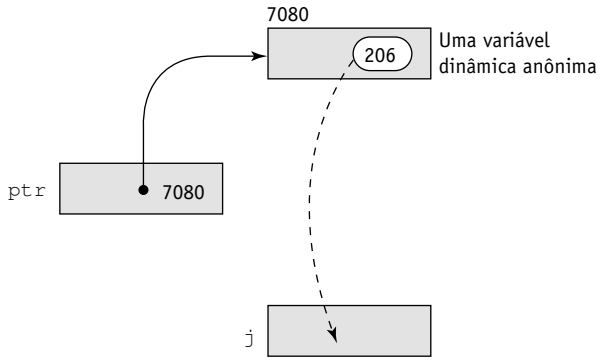


FIGURA 6.8
A operação de atribuição `j = *ptr`.

desreferenciamento e a referência ao campo. Por exemplo, a expressão `p->age` é equivalente a `(*p).age`.

Linguagens que fornecem ponteiros para o gerenciamento de um monte devem incluir uma operação de alocação explícita. A alocação é algumas vezes especificada com um subprograma, como `malloc` em C. Em linguagens que suportam programação orientada a objetos, a alocação de objetos no monte é normalmente especificada com o operador **`new`**. C++, que não permite liberação implícita, usa **`delete`** como operador de liberação.

6.11.3 Problemas com ponteiros

A primeira linguagem de programação de alto nível a incluir variáveis do tipo ponteiro foi PL/I, na qual os ponteiros podiam ser usados para referenciar tanto variáveis dinâmicas do monte quanto outras variáveis do programa. Os ponteiros de PL/I eram altamente flexíveis, mas seu uso podia levar a inúmeros tipos de erros de programação. Alguns dos problemas dos ponteiros em PL/I também estão presentes nos das linguagens subsequentes. Algumas linguagens recentes, como Java, substituíram completamente os ponteiros por tipos de referência que, com a liberação implícita, minimizam os principais problemas. Um tipo de referência é apenas um ponteiro com operações restritas.

6.11.3.1 Ponteiros soltos

Um **ponteiro solto***, ou **referência solta**, é um ponteiro que contém o endereço de uma variável dinâmica do monte já liberada. Ponteiros soltos são perigosos por diversas razões. Primeiro, a posição que é apontada pode ter sido realocada para alguma nova variável dinâmica do monte. Se a nova variável não for do mesmo tipo da antiga, as verificações de tipos dos usos do ponteiro solto serão inválidas. Ainda que a nova variável dinâmica seja do mesmo tipo, seu novo valor não terá relacionamento com o valor desreferenciado do ponteiro antigo. Além disso, se o ponteiro solto for usado para modificar a

* N. de T.: Tais ponteiros também são conhecidos como ponteiros pendurados, ponteiros pendentes ou ponteiros selvagens (*dangling pointers*).

variável dinâmica do monte, o valor da nova variável será destruído. Por fim, é possível que a posição seja temporariamente usada pelo sistema de gerenciamento de armazenamento, provavelmente como um ponteiro em uma cadeia de blocos de armazenamento disponíveis, permitindo que uma mudança na posição acarrete uma falha no gerente de armazenamento.

A seguinte sequência de operações cria um ponteiro solto em muitas linguagens:

1. Uma nova variável dinâmica do monte é criada e o ponteiro `p1` é configurado para apontar para ela.
2. O ponteiro `p2` é atribuído como o valor de `p1`.
3. A variável dinâmica do monte apontada por `p1` é explicitamente liberada (possivelmente configurando `p1` como `nil`), mas `p2` não é modificado pela operação. `p2` agora é um ponteiro solto. Se a operação de liberação não modificasse `p1`, tanto `p1` quanto `p2` seriam soltos. (Evidentemente, esse é um problema decorrente do uso de apelidos – `p1` e `p2` são apelidos.)

Por exemplo, em C++ teríamos:

```
int * arrayPtr1;
int * arrayPtr2 = new int[100];
arrayPtr1 = arrayPtr2;
delete [] arrayPtr2;
// Agora, arrayPtrl é solto, porque o
// armazenamento no monte para o qual ele
// apontava foi liberado.
```

Em C++, tanto `arrayPtr1` quanto `arrayPtr2` são agora ponteiros soltos, porque o operador C++ **`delete`** não tem efeito no valor de seu ponteiro passado como operando. Em C++, é comum (e seguro) seguir um operador **`delete`** com uma atribuição de zero, que representa null, para o ponteiro cujo valor apontado foi liberado.

Note que a liberação explícita de variáveis dinâmicas é a causa dos ponteiros soltos.

> **» *nota histórica***
>
> Pascal incluía um operador de liberação explícito: dispose. Em decorrência dos problemas de ponteiros soltos causados por dispose, algumas implementações da linguagem Pascal o ignoravam quando ele aparecia em um programa. Apesar de isso evitar os ponteiros soltos, também impedia o reúso do armazenamento no monte que era desnecessário para o programa. Lembre-se de que, inicialmente, a linguagem Pascal foi projetada para ser uma linguagem de ensino, em vez de uma ferramenta industrial.

6.11.3.2 Variáveis dinâmicas do monte perdidas

Uma **variável dinâmica do monte perdida** é uma alocada que não está mais acessível para os programas de usuário. Elas são frequentemente chamadas de **lixo**, pois não são úteis para seus propósitos originais e não podem ser realocadas para algum novo uso no programa. Variáveis dinâmicas do monte perdidas são, em geral, criadas pela seguinte sequência de operações:

1. O ponteiro `p1` é configurado para apontar para uma variável dinâmica do monte recém-criada.

2. p1 é posteriormente configurado para apontar para outra variável dinâmica do monte recém-criada.

A primeira variável dinâmica do monte está agora inacessível, ou perdida. Isso, às vezes, é chamado de **vazamento de memória**. O vazamento de memória é um problema, independentemente de a linguagem usar liberação implícita ou explícita. Nas seções a seguir, investigamos como os projetistas de linguagens lidaram com os problemas de ponteiros soltos e variáveis dinâmicas do monte perdidas.

6.11.4 Ponteiros em C e C++

Em C e C++, os ponteiros podem ser usados da mesma forma como os endereços são usados em linguagens de montagem. Isso significa que são extremamente flexíveis, mas devem ser usados com muito cuidado. Seu projeto não oferece soluções para os problemas relacionados aos ponteiros soltos ou às variáveis dinâmicas do monte perdidas. Entretanto, o fato de a aritmética de ponteiros ser possível em C e C++ torna seus ponteiros mais interessantes do que os de outras linguagens de programação.

Os ponteiros de C e C++ podem apontar para qualquer variável, independentemente de onde ela estiver alocada. Na verdade, eles podem apontar para qualquer lugar da memória, independentemente de lá existir uma variável ou não, um dos perigos de tais ponteiros.

Em C e C++, o asterisco (*) denota a operação de desreferenciamento e o e comercial (&) denota o operador para produzir o endereço de uma variável. Por exemplo, considere o código:

```
int *ptr;
int count, init;
. . .
ptr = &init;
count = *ptr;
```

A atribuição para a variável ptr a configura com o endereço de init. A atribuição a count desreferencia ptr para produzir o valor em init, então atribuído a count. O efeito das duas sentenças de atribuição é atribuir o valor de init a count. Note que a declaração de um ponteiro especifica seu tipo de domínio.

Aos ponteiros pode ser atribuído o valor de endereço de qualquer variável do tipo de domínio correto ou a constante zero, usada para nil.

A aritmética de ponteiros também é possível de algumas formas restritas. Por exemplo, se ptr é uma variável de ponteiro declarada para apontar para uma variável de algum tipo de dados, então

```
ptr + index
```

é uma expressão válida. A semântica de tal expressão é a seguinte: em vez de o valor de index ser simplesmente adicionado a ptr, ele é primeiro escalado pelo tamanho da célula de memória (em unidades de memória) para a qual ptr está apontando (seu tipo base). Por exemplo, se ptr aponta para uma célula de memória de um tipo com um tamanho de quatro unidades de memória, então index é multiplicado por 4 e o resultado

é adicionado a `ptr`. O propósito primário para esse tipo de aritmética de endereços é a manipulação de matrizes. A discussão a seguir é relacionada apenas a matrizes de uma dimensão.

Em C e C++, todas as matrizes usam zero como limite inferior de suas faixas de índices, e nomes de matrizes sem índices sempre se referem ao endereço do primeiro elemento. Considere as declarações:

```
int list [10];
int *ptr;
```

Agora, considere a atribuição

```
ptr = list;
```

Ela atribui o endereço de `list[0]` a `ptr`. Dada essa atribuição, as seguintes afirmações são verdadeiras:

- `*(ptr + 1)` é equivalente a `list[1]`.
- `*(ptr + index)` é equivalente a `list[index]`.
- `ptr[index]` é equivalente a `list[index]`.

Está claro, a partir dessas sentenças, que as operações de ponteiros incluem a mesma escala usada em operações de indexação. Além disso, ponteiros para matrizes podem ser indexados como se fossem nomes de matrizes.

Ponteiros em C e C++ podem apontar para funções. Esse recurso é usado para passar funções como parâmetros para outras funções. Ponteiros também são usados para passagem de parâmetros, conforme discutido no Capítulo 9.

C e C++ incluem ponteiros do tipo **`void *`**, os quais podem apontar para valores de quaisquer tipos. Na verdade, eles são ponteiros genéricos. Entretanto, a verificação de tipos não é um problema com ponteiros **`void *`**, porque essas linguagens não permitem desreferenciá-los. Os ponteiros **`void *`** são comumente usados como os tipos de parâmetros de funções que operam em memória. Por exemplo, suponha que quiséssemos uma função para mover uma sequência de bytes de dados de um lugar para outro na memória. Ela seria mais geral se pudessem ser passados ponteiros de qualquer tipo. Isso seria permitido se os parâmetros formais correspondentes na função fossem do tipo **`void *`**. Então, a função os converteria no tipo **`char *`** e efetuaria a operação, independentemente de quais tipos de ponteiros fossem passados como parâmetros reais.

6.11.5 Tipos de referência

Uma variável de **tipo de referência** é similar a um ponteiro, com uma diferença fundamental: um ponteiro se refere a um endereço em memória, enquanto uma referência, a um objeto ou a um valor em memória. Consequentemente, apesar de ser natural realizar aritmética em endereços, não faz sentido fazê-lo em referências.

C++ inclui uma forma especial de tipo de referência usada primariamente para os parâmetros formais em definições de funções. Uma variável de tipo de referência em C++ é um ponteiro constante sempre desreferenciado implicitamente. Como uma variável de

tipo de referência em C++ é uma constante, ela deve ser inicializada com o endereço de alguma variável em sua definição e, após a inicialização, uma variável de tipo de referência nunca pode ser modificada para referenciar qualquer outra variável. A desreferência implícita impede a atribuição ao valor de endereço de uma variável de referência.

Variáveis de tipo de referência são especificadas em definições ao se preceder seus nomes com o sinal de e comercial (`&`). Por exemplo,

```
int result = 0;
int &ref_result = result;
. . .
ref_result = 100;
```

Nesse segmento de código, `result` e `ref_result` são apelidos.

Quando usados como parâmetros formais em definições de funções, os tipos de referência em C++ fornecem uma comunicação bidirecional entre as funções chamadora e chamada. Isso não é possível com os tipos de parâmetros primitivos que não são ponteiros, porque os parâmetros em C++ são passados por valor. Passar um ponteiro como um parâmetro realiza a mesma comunicação bidirecional, mas parâmetros formais como ponteiros exigem desreferenciamento explícito, tornando o código menos legível e menos seguro. Parâmetros de referência são referenciados na função chamada exatamente como os outros parâmetros. A função chamadora não precisa especificar que um parâmetro, cujo parâmetro formal correspondente é um tipo de referência, é algo não usual. O compilador passa endereços, em vez de valores, para parâmetros de referência.

Em sua busca por maior segurança em relação à C++, os projetistas de Java removeram completamente os ponteiros estilo C++. Diferentemente das variáveis de referência de C++, as de Java podem ser atribuídas a diferentes instâncias de classes; elas não são constantes. Todas as instâncias de classes em Java são referenciadas por variáveis de referência. Na verdade, esse é o único uso para variáveis de referência em Java. Tais questões são discutidas em mais detalhes no Capítulo 12.

No código a seguir, `String` é uma classe padrão de Java:

```
String str1;
. . .
str1 = "This is a Java literal string";
```

Nesse código, `str1` é definida como uma referência a uma instância (ou objeto) da classe `String`. Ela é inicialmente configurada como null. A atribuição subsequente configura `str1` para referenciar o objeto `String`, `"This is a Java literal string"`.

Como as instâncias de classe em Java são liberadas implicitamente (não existe um operador de liberação explícito), não podem existir referências soltas nessa linguagem.

C# inclui tanto as referências de Java quanto os ponteiros de C++. Entretanto, o uso de ponteiros é veementemente desencorajado. Na verdade, quaisquer programas que usem ponteiros devem incluir o modificador **`unsafe`**. Note que, apesar de os objetos apontados por referências serem liberados implicitamente, isso não é verdade para objetos apontados por ponteiros. Os ponteiros foram incluídos em C# principalmente para permitir que os programas nessa linguagem interoperassem com código C e C++.

Todas as variáveis são referências nas linguagens orientadas a objetos Smalltalk, Python, Ruby e Lua. Elas são sempre desreferenciadas implicitamente. Além disso, os valores diretos dessas variáveis não podem ser acessados.

6.11.6 Avaliação

Os problemas de ponteiros soltos e lixo já foram discutidos em profundidade. Os problemas de gerenciamento do monte são discutidos na Seção 6.11.7.3.

Ponteiros têm sido comparados à instrução goto, que aumenta a faixa de sentenças que podem ser executadas em seguida. Variáveis de ponteiros aumentam a faixa de células de memórias que podem ser referenciadas por uma variável. Talvez a frase mais contundente a respeito dos ponteiros tinha sido escrita por Hoare (1973): "Sua introdução nas linguagens de alto nível foi um passo para trás, do qual talvez nunca nos recuperemos."

Por outro lado, os ponteiros são essenciais em alguns tipos de aplicações de programação. Por exemplo, eles são necessários para escrever *drivers* de dispositivos, nos quais os endereços absolutos específicos precisam ser acessados.

As referências de Java e C# fornecem algo da flexibilidade e das capacidades dos ponteiros, sem seus problemas associados. Resta-nos ver se os programadores estarão dispostos a trocar a eficácia dos ponteiros em C e C++ pela maior segurança das referências. A extensão do uso de ponteiros nos programas C# será uma medida disso.

6.11.7 Implementação de ponteiros e de tipos de referência

Na maioria das linguagens, os ponteiros são usados no gerenciamento do monte. O mesmo vale para referências em Java e em C#, assim como para as variáveis em Smalltalk e Ruby; então, não podemos tratar ponteiros e referências separadamente. Primeiro, descreveremos brevemente como ponteiros e referências são representados internamente. Então, discutiremos duas soluções possíveis para o problema dos ponteiros soltos. Por fim, descreveremos os principais problemas das técnicas de gerenciamento do monte.

6.11.7.1 Representação de ponteiros e de tipos de referência

Na maioria dos computadores de grande e médio porte, os ponteiros e as referências são valores únicos armazenados em células de memória. Entretanto, nos primeiros microcomputadores baseados em microprocessadores Intel, os endereços têm duas partes: um segmento e um deslocamento. Então, os ponteiros e as referências são implementados nesses sistemas como pares de células de 16 bits, um para cada uma das duas partes de um endereço.

6.11.7.2 Solução para o problema dos ponteiros soltos

Existem diversas soluções propostas para o problema dos ponteiros soltos. Entre elas, estão as **lápides** (*tombstones*) (Lomet, 1975), nas quais cada variável dinâmica do monte inclui uma célula especial, chamada de lápide, que é um ponteiro para a variável dinâmica do monte. A variável de ponteiro real aponta apenas para lápides e nunca para variáveis dinâmicas do monte. Quando uma variável dinâmica do monte é liberada, a lápide continua a existir, mas é atribuído a ela o valor `nil`, indicando que a variável

dinâmica do monte não existe mais. Essa abordagem evita que um ponteiro aponte para uma variável liberada. Qualquer referência para qualquer ponteiro que aponte para uma lápide nula pode ser detectada como um erro.

Lápides são dispendiosas, tanto em termos de tempo quanto de espaço. Como elas nunca são liberadas, seu armazenamento nunca é solicitado de volta. Cada acesso a uma variável dinâmica do monte por meio de uma lápide requer mais um nível de ação indireta, o que, na maioria dos computadores, exige um ciclo adicional de máquina. Aparentemente, nenhum dos projetistas das linguagens mais populares achou que a segurança adicional valia o custo, já que nenhuma linguagem amplamente usada emprega lápides.

Uma alternativa às lápides é a **abordagem fechaduras e chaves** (*locksandkeys*) usada na implementação do UW-Pascal (Fischer e LeBlanc, 1977, 1980). Nesse compilador, os valores de ponteiros são representados como pares ordenados (chave, endereço), onde a chave é um valor inteiro. Variáveis dinâmicas do monte são representadas como o armazenamento da variável, mais uma célula de cabeçalho que armazena um valor de fechadura inteiro. Quando uma variável dinâmica do monte é alocada, um valor de fechadura é criado e colocado tanto na célula de fechadura na variável dinâmica do monte quanto na célula chave do ponteiro que é especificado na chamada a **`new`**. Cada acesso ao ponteiro desreferenciado compara o valor da chave do ponteiro com o valor da fechadura na variável dinâmica do monte. Se eles casam, o acesso é válido; caso contrário, é tratado como um erro em tempo de execução. Quaisquer cópias do valor do ponteiro para outros ponteiros devem copiar o valor da chave. Logo, qualquer número de ponteiros pode referenciar uma variável dinâmica do monte. Quando uma variável dinâmica do monte é liberada com **`dispose`**, seu valor de fechadura é substituído por um valor de fechadura inválido. Então, se um ponteiro que não for aquele especificado em **`dispose`** for desreferenciado, seu valor de endereço ainda assim estará intacto, mas seu valor de chave não casará mais com a fechadura, então o acesso não será permitido.

É claro que a melhor solução para o problema dos ponteiros soltos é tirar das mãos dos programadores a responsabilidade pela liberação de variáveis dinâmicas do monte. Se os programas não puderem liberar variáveis dinâmicas do monte explicitamente, não existirão ponteiros soltos. Para fazer isso, o sistema em tempo de execução deve liberar implicitamente as variáveis dinâmicas do monte, quando elas não forem mais úteis. Os sistemas Lisp já faziam isso. Tanto Java quanto C# usam essa estratégia para suas variáveis de referência. Lembre-se de que os ponteiros em C# não incluem liberação implícita.

6.11.7.3 Gerenciamento do monte

O gerenciamento do monte pode ser um processo em tempo de execução bastante complexo. Examinaremos o processo em duas situações: uma na qual todo o armazenamento do monte é alocado e liberado em unidades de tamanho único, e uma na qual são alocados e liberados segmentos de tamanho variável. Note que, para a liberação, discutiremos apenas as estratégias implícitas. Nossa discussão será breve e elementar, visto que uma análise cuidadosa desses processos e seus problemas associados não é tanto uma questão de projeto de linguagem, e sim uma questão de implementação.

Células de tamanho único A situação mais simples é aquela em que toda a alocação e a liberação ocorre em células de tamanho fixo. Ela é ainda mais simplificada quando

cada célula já contém um ponteiro. Esse é o cenário de muitas implementações de Lisp, em que os problemas de alocação de armazenamento dinâmico foram encontrados em larga escala pela primeira vez. Todos os programas Lisp e a maioria dos dados na linguagem consistem em células de listas encadeadas.

Em um monte de alocação de tamanho único, todas as células disponíveis são encadeadas usando os ponteiros nas células e formando uma lista de espaços disponíveis. A alocação consiste simplesmente em pegar o número de células necessárias dessa lista quando elas forem requisitadas. A liberação é um processo muito mais complexo. Uma variável dinâmica do monte pode ser apontada por mais de um ponteiro, tornando difícil determinar quando a variável não é mais útil para o programa. O fato de um ponteiro estar desconectado de uma célula obviamente não a torna lixo; podem existir diversos outros ponteiros que ainda estão apontando para a célula.

Em Lisp, diversas das operações mais frequentes em programas criam coleções de células que não ficam mais acessíveis ao programa e, portanto, deveriam ser liberadas (colocadas na lista de espaço disponível). Um dos objetivos fundamentais do projeto de Lisp era garantir que a recuperação de células não usadas não fosse tarefa do programador, mas do sistema de tempo de execução. Tal objetivo deixou os implementadores de Lisp com a seguinte questão fundamental de projeto: quando a liberação deve ser realizada?

Existem diversas abordagens diferentes para a coleta de lixo. As duas técnicas tradicionais mais comuns são, de certa forma, processos opostos. Elas são chamadas de **contadores de referências**, nos quais a recuperação de memória é incremental e feita quando células inacessíveis são criadas, e de **marcar e varrer**, na qual a recuperação ocorre apenas quando a lista de espaços disponíveis se torna vazia. Esses dois métodos são, algumas vezes, chamados de **abordagem ansiosa** (*eager*) e **abordagem preguiçosa** (*lazy*), respectivamente. Muitas variações dessas duas abordagens foram desenvolvidas. Nesta seção, entretanto, discutimos apenas os processos básicos.

O método de contagem de referências para a recuperação de armazenamento atinge seu objetivo mantendo um contador em cada célula que armazena o número de ponteiros que atualmente apontam para a célula. Embutida na operação de decremento para contadores de referência, que ocorre quando um ponteiro é desconectado da célula, está uma verificação para um valor igual a zero. Se a contagem de referências chega a zero, isso significa que nenhum ponteiro no programa está apontando para a célula; então, ela se tornou lixo e pode ser retornada para a lista de espaço disponível.

Existem três problemas no método de contagem de referências. Primeiro, se as células de armazenamento são relativamente pequenas, o espaço necessário para os contadores é significativo. Segundo, algum tempo de execução é obviamente necessário para manter os valores de contagem. Cada vez que um valor de ponteiro é modificado, a célula para a qual ele estava apontando deve ter seu contador decrementado, e a célula para a qual ele está apontando agora deve ter seu contador incrementado. Em uma linguagem como Lisp, na qual praticamente toda ação envolve a modificação de ponteiros, isso pode ocupar uma porção significativa do tempo de execução total de um programa. É claro que, se as mudanças nos ponteiros não forem muito frequentes, isso não será um problema. Parte da ineficiência dos contadores de referência pode ser eliminada pela abordagem chamada de **contagem de referências desreferenciadas**, que evita a contagem de referências para alguns ponteiros. O terceiro problema é que surgem com-

plicações quando uma coleção de células é conectada circularmente. O problema aqui é que cada célula na lista circular tem um valor de contagem de referências de no mínimo 1, o que a impede de ser coletada e colocada de volta na lista de espaço disponível. Uma solução para esse problema pode ser encontrada em Friedman e Wise (1979).

A vantagem da abordagem de contagem de referências é que ela é intrinsecamente incremental. Suas ações são intercaladas com as da aplicação; portanto, ela nunca causa demoras significativas na execução da aplicação.

O processo original da coleta de lixo marcar e varrer funciona como descrito a seguir: o sistema de tempo de execução aloca células de armazenamento conforme solicitado e desconecta ponteiros de células conforme a necessidade, sem se preocupar com a recuperação de armazenamento (permitindo que o lixo se acumule), até que tenha alocado todas as células disponíveis. Nesse ponto, é iniciado um processo de marcar e varrer para recolher todo o lixo que foi espalhado em torno do monte. Para facilitar esse processo, cada célula do monte tem um bit ou campo indicador extra, usado pelo algoritmo de coleta.

O processo marcar e varrer consiste em três fases distintas. Primeiro, todas as células no monte têm seus indicadores configurados para indicar que eles são lixo. Essa é, obviamente, uma condição correta para apenas algumas das células. A segunda parte, chamada de fase marcar, é a mais difícil. Cada ponteiro no programa é rastreado no monte e todas as células alcançáveis são marcadas como não sendo lixo. Após isso, a terceira fase, chamada de fase varrer, é executada: todas as células no monte que não foram especificamente classificadas como em uso são retornadas para a lista de espaço disponível.

Para ilustrar o tipo dos algoritmos utilizados para marcar as células que são usadas atualmente, fornecemos a seguinte versão simples de um algoritmo de marcação. Assumimos que todas as variáveis dinâmicas do monte, ou células do monte, consistem em uma parte de informação; uma parte para o marcador, chamada `marker`; e dois ponteiros chamados `llink` e `rlink`. Essas células são usadas para construir grafos dirigidos com no máximo duas arestas partindo de qualquer nó. O algoritmo de marcação percorre todas as árvores de extensão do grafo, marcando todas as células encontradas. Como outros percursos de grafos, o algoritmo de marcação usa recursão.

```
for every pointer r do
    mark(r)

void mark(void * ptr) {
    if (ptr != 0)
      if (*ptr.marker is not marked) {
        set *ptr.marker
        mark(*ptr.llink)
        mark(*ptr.rlink)
      }
}
```

A Figura 6.9 mostra um exemplo das ações desse procedimento em um grafo dado. Esse algoritmo de marcação simples requer uma boa dose de armazenamento (de espaço de pilha para suportar recursão). Um processo de marcação que não requer espaço de pilha adicional foi desenvolvido por Schorr e Waite (1967). Seu método inverte os

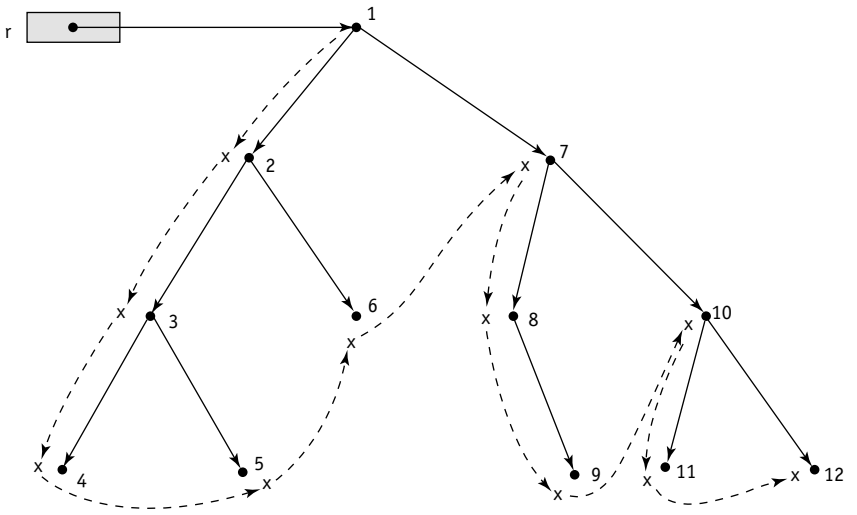


FIGURA 6.9
Um exemplo das ações do algoritmo de marcação.

ponteiros à medida que rastreia estruturas ligadas. Então, quando o final de uma lista é alcançado, o processo pode seguir os ponteiros retroativamente para fora da estrutura.

O problema mais sério da versão original da abordagem marcar e varrer era o fato de ela ser realizada raramente – apenas quando um programa usasse todo ou praticamente todo o armazenamento do monte. Marcar e varrer nessa situação leva um bom tempo, porque a maioria das células precisa ser rastreada e marcada como em uso atualmente. Isso causa uma espera significativa no andamento da aplicação. Além disso, o processo pode levar a apenas um pequeno número de células que podem ser colocadas na lista de espaço disponível. Esse problema foi resolvido por meio de uma variedade de melhorias. Por exemplo, a coleta de lixo **marcar e varrer incremental** ocorre com mais frequência, bem antes de a memória estar esgotada, tornando o processo mais eficiente em termos da quantidade de armazenamento recuperada. Além disso, o tempo necessário para cada execução do processo é obviamente menor, reduzindo a espera na execução das aplicações. Outra possibilidade é realizar o processo marcar e varrer em partes, em vez de em toda a memória associada à aplicação, em diferentes momentos. Isso fornece os mesmos tipos de melhorias que as proporcionadas pelo marcar e varrer incremental.

Tanto os algoritmos de marcação para o método marcar e varrer como os processos necessários para o método de contagem de referências podem se tornar mais eficientes pelo uso de rotação de ponteiros e operações de deslocamento descrito por Suzuki (1982).

Células de tamanho variável Gerenciar um monte a partir do qual células de tamanho variável[8] são alocadas envolve todas as dificuldades de gerenciar um monte para célu-

[8]As células têm tamanhos variáveis porque são abstratas e armazenam os valores das variáveis independentemente de seus tipos. Além disso, uma variável poderia ser um tipo estruturado.

las de tamanho único, além de problemas adicionais. Infelizmente, células de tamanho variável são exigidas pela maioria das linguagens de programação. Os problemas adicionais advindos do gerenciamento de células de tamanho variável dependem do método usado. Se for usado o marcar e varrer, os seguintes problemas adicionais ocorrem:

- A configuração inicial dos indicadores para todas as células no monte, para indicar que elas são lixo, é difícil. Como as células são de tamanhos diferentes, percorrê-las é um problema. Uma solução é exigir que cada célula tenha seu tamanho especificado como seu primeiro campo. Então, o percurso pode ser feito, apesar de exigir um pouco mais de espaço e uma quantidade maior de tempo que seu correspondente para células de tamanho fixo.
- O processo de marcação não é trivial. Como um encadeamento pode ser percorrido a partir de um ponteiro se não existe uma posição predefinida para o ponteiro nas células apontadas? As células que não contêm ponteiros também são problemáticas. Adicionar um ponteiro interno para cada célula, mantido em segundo plano pelo sistema em tempo de execução, funcionará. Entretanto, esse processamento de manutenção em segundo plano adiciona sobrecargas ao custo de executar o programa, tanto em relação ao espaço quanto em relação ao tempo de execução.
- Manter a lista de espaços disponíveis é outra fonte de sobrecarga. A lista pode começar com uma única célula formada por todo o espaço disponível. Requisições de segmentos simplesmente reduzem o tamanho desse bloco. Células liberadas são adicionadas à lista. O problema é que, cedo ou tarde, a lista se torna longa, com segmentos, ou blocos, de tamanhos variados. Isso torna a alocação mais lenta, porque as requisições fazem a lista ser percorrida em busca de blocos grandes o suficiente. Por fim, a lista pode consistir em um grande número de blocos muito pequenos, que não são grandes o suficiente para a maioria das requisições. Nesse ponto, blocos adjacentes talvez precisem ser unidos em blocos maiores. Alternativas ao uso do primeiro bloco suficientemente grande na lista podem diminuir a busca, mas podem exigir que a lista seja ordenada pelo tamanho do bloco. Em ambos os casos, manter a lista é uma sobrecarga adicional.

Se forem usados contadores de referências, os dois primeiros problemas são evitados, mas a questão do espaço disponível na lista de manutenção permanece.

Para um estudo abrangente sobre problemas de gerenciamento de memória, consulte Wilson (2005).

6.12 VERIFICAÇÃO DE TIPOS

Para a discussão sobre a verificação de tipos, o conceito de operandos e operadores é generalizado a fim de incluir subprogramas e sentenças de atribuição. Subprogramas serão pensados como operadores cujos operandos são seus parâmetros. O símbolo de atribuição será considerado um operador binário, com sua variável alvo e sua expressão sendo os operandos.

A **verificação de tipos** é a atividade que garante que os operandos de um operador sejam de tipos compatíveis. Um tipo **compatível** ou é válido para o operador, ou pode, dentro das regras da linguagem, ser implicitamente convertido pelo código gerado pelo

compilador (ou pelo interpretador) em um tipo válido. Essa conversão automática é chamada de **coerção**. Por exemplo, se uma variável `int` e uma variável `float` são adicionadas em Java, o valor da variável `int` sofre uma coerção para `float` e uma adição de ponto flutuante é efetuada.

Um **erro de tipo** é a aplicação de um operador a um operando de um tipo não apropriado. Por exemplo, na versão original de C, se um valor `int` fosse passado para uma função que esperava um valor `float`, ocorria um erro de tipo (porque os compiladores dessa linguagem não verificavam os tipos dos parâmetros).

Se todas as vinculações de variáveis a tipos são estáticas na linguagem, a verificação de tipos pode ser feita praticamente sempre de maneira estática. A vinculação dinâmica de tipos requer a verificação de tipos em tempo de execução, chamada de **verificação de tipos dinâmica**.

Algumas linguagens, como JavaScript e PHP, devido à sua vinculação de tipos dinâmica, permitem apenas a verificação de tipos dinâmica. É melhor detectar erros em tempo de compilação do que em tempo de execução, porque a correção feita mais cedo é geralmente menos dispendiosa. A penalidade para a verificação estática é uma flexibilidade reduzida para o programador. Menos atalhos e truques são permitidos. Contudo, tais técnicas agora são normalmente consideradas propensas a erros e prejudiciais à legibilidade.

A verificação de tipos é complicada quando uma linguagem permite que uma célula de memória armazene valores de tipos diferentes em momentos diferentes da execução. Tais células de memória podem ser criadas com uniões de C e C++ e com as uniões discriminadas de ML, Haskell e F#. Nesses casos, a verificação de tipos, se for feita, deve ser dinâmica e requer que o sistema em tempo de execução mantenha o tipo do valor atual de tais células de memória. Então, apesar de todas as variáveis serem estaticamente vinculadas a tipos em linguagens como C++, nem todos os erros de tipos podem ser detectados por verificação estática de tipos.

6.13 TIPAGEM FORTE

Uma das ideias sobre projeto de linguagem que se tornou proeminente na chamada revolução da programação estruturada da década de 1970 foi a **tipagem forte**. Ela é amplamente reconhecida como uma característica de linguagem muito valiosa. Infelizmente, muitas vezes é definida de maneira pouco rígida, ou mesmo usada na literatura em computação sem ser definida.

Uma linguagem de programação é **fortemente tipada** se erros de tipos são sempre detectados. Isso requer que os tipos de todos os operandos possam ser determinados em tempo de compilação ou em tempo de execução. A importância da tipagem forte está na habilidade de detectar usos incorretos de variáveis que resultam em erros de tipo. Uma linguagem fortemente tipada também permite a detecção, em tempo de execução, de usos de valores de tipo incorretos em variáveis que podem armazenar valores de mais de um tipo.

C e C++ não são linguagens fortemente tipadas, porque ambas incluem tipos união que não são verificados em relação a tipos.

ML é fortemente tipada, apesar de tipos de alguns parâmetros de funções poderem não ser conhecidos em tempo de compilação. F# também é fortemente tipada.

Java e C#, embora sejam baseadas em C++, são quase fortemente tipadas. Os tipos podem ser convertidos explicitamente, resultando em um erro de tipos. Entretanto, não existem maneiras implícitas pelas quais os erros de tipos possam passar despercebidos.

As regras de coerção de uma linguagem têm efeito importante no valor da verificação de tipos. Por exemplo, as expressões são fortemente tipadas em Java. Entretanto, um operador aritmético com um operando de ponto flutuante e um operador inteiro é válido. O valor do operando inteiro sofre uma coerção para ponto flutuante e uma operação de ponto flutuante é efetuada. Isso é o que o programador normalmente pretende. Entretanto, a coerção também resulta em uma perda de um dos benefícios da tipagem forte – a detecção de erros. Por exemplo, suponha que um programa tem as variáveis inteiras (**`int`**) `a` e `b` e a variável de ponto flutuante (**`float`**) chamada `d`. Agora, se um programador quisesse digitar `a + b`, mas equivocadamente digitasse `a + d`, o erro não seria detectado pelo compilador. O valor de `a` simplesmente sofreria uma coerção para **`float`**. Então, o valor da tipagem forte é enfraquecido pela coerção. Linguagens com muitas coerções, como C e C++, são menos confiáveis do que linguagens com poucas coerções, como ML e F#. Java e C# têm cerca de metade das coerções de tipo em atribuições que C++; portanto, sua detecção de erros é melhor que a de C++, mas ainda assim não é tão eficiente quanto as de ML e F#. A questão da coerção é examinada em detalhes no Capítulo 7.

6.14 EQUIVALÊNCIA DE TIPOS

A ideia da compatibilidade de tipos foi definida quando ocorreu a introdução da questão da verificação de tipos. As regras de compatibilidade ditam os tipos de operandos aceitáveis para cada um dos operadores e especificam os possíveis tipos de erros da linguagem.[9] As regras são chamadas "de compatibilidade" porque, em alguns casos, o tipo de um operando pode ser convertido implicitamente pelo compilador ou pelo sistema de tempo de execução para tornar-se aceitável pelo operador.

As regras de compatibilidade de tipos são simples e rígidas para os escalares predefinidos. Entretanto, nos casos dos tipos estruturados, como matrizes e registros e alguns tipos definidos pelo usuário, as regras são mais complexas. A coerção desses tipos é rara; então, a questão não é a compatibilidade, mas a equivalência. Ou seja, dois tipos são **equivalentes** em uma expressão se um operando de um tipo é substituído por um de outro, sem coerção. A equivalência de tipos é uma forma estrita da compatibilidade – compatibilidade sem coerção. A questão central aqui é como essa equivalência é definida.

O projeto das regras de equivalência de tipos de uma linguagem é importante, porque influencia o projeto dos tipos de dados e das operações fornecidas para seus valores. Com os tipos discutidos aqui, existem poucas operações predefinidas. Talvez o resultado mais importante de duas variáveis serem de tipos equivalentes é que uma pode ter seu valor atribuído para a outra.

[9] A compatibilidade de tipos também é uma questão no relacionamento entre os parâmetros reais em uma chamada a subprograma e os parâmetros formais na definição do subprograma. Essa questão é discutida no Capítulo 9.

Existem duas abordagens para definir equivalência de tipos: por nome e por estrutura. A **equivalência de tipos por nome** significa que duas variáveis são equivalentes se são definidas na mesma declaração ou em declarações que usam o mesmo nome de tipo. A **equivalência de tipos por estrutura** significa que duas variáveis têm tipos equivalentes se seus tipos têm estruturas idênticas. Existem algumas variações dessas abordagens, e muitas linguagens usam combinações de ambas.

A equivalência de tipos por nome é fácil de implementar, mas é mais restritiva. Em uma interpretação estrita, uma variável cujo tipo é uma subfaixa dos inteiros não seria equivalente a uma variável do tipo inteiro. Por exemplo, suponha que Ada usasse equivalência estrita de tipos por nome e considere o seguinte código:

```
type Indextype is 1..100;
count : Integer;
index : Indextype;
```

Os tipos das variáveis `count` e `index` não seriam equivalentes; `count` não poderia ser atribuída a `index` ou vice-versa.

Outro problema da equivalência de tipos por nome surge quando um tipo estruturado ou um definido pelo usuário é passado entre subprogramas por meio de parâmetros. Tal tipo deve ser definido apenas uma vez, globalmente. Um subprograma não pode informar o tipo de tais parâmetros formais em termos locais. Esse era o caso na versão original do Pascal.

Note que, para usar a equivalência de tipos por nome, todos eles devem ter nomes. A maioria das linguagens permite aos usuários definirem tipos anônimos – sem nomes. Para que uma linguagem use equivalência de tipos por nome, o compilador deve nomeá-los (com nomes internos) implicitamente.

A equivalência de tipos por estrutura é mais flexível que a equivalência por nome, mas é mais difícil de ser implementada. Sob a equivalência por nome, apenas os nomes dos dois tipos precisam ser comparados para determiná-la. Sob a equivalência de tipos por estrutura, entretanto, as estruturas inteiras dos dois tipos devem ser comparadas. Essa comparação nem sempre é simples. (Considere uma estrutura de dados que se refere ao seu próprio tipo, como uma lista encadeada.) Outras questões também podem surgir. Por exemplo, dois tipos de registro (ou **`struct`**) são equivalentes se têm a mesma estrutura, mas nomes de campos diferentes? Dois tipos de matrizes de uma dimensão, em uma linguagem que permite que limites inferiores de faixas de índice sejam definidos em suas declarações, são equivalentes se têm o mesmo tipo de elemento, mas faixas de índice `0..10` e `1..11`? Dois tipos de enumeração são equivalentes se têm o mesmo número de componentes, mas literais com nomes diferentes?

Outra dificuldade na equivalência de tipos por estrutura é que ela não permite diferenciar tipos com a mesma estrutura. Por exemplo, considere as seguintes declarações em Ada:

```
type Celsius = Float;
     Fahrenheit = Float;
```

Os tipos das variáveis desses dois tipos são considerados equivalentes sob a equivalência de tipos por estrutura, permitindo que sejam misturados em expressões, o que é claramente indesejável nesse caso, considerando a diferença indicada pelos nomes dos tipos.

De modo geral, tipos com nomes diferentes provavelmente são abstrações de diferentes categorias de valores de problemas e não devem ser considerados equivalentes.

Ada usa uma forma restritiva de equivalência de tipos por nome, mas fornece duas construções de tipos – subtipos e tipos derivados – que evitam os problemas associados à equivalência por nome. Um **tipo derivado** é um novo tipo baseado em algum previamente definido ao qual ele não é equivalente, apesar de terem estrutura idêntica. Os tipos derivados herdam todas as propriedades de seus ancestrais. Considere o seguinte exemplo:

```
type Celsius is new Float;
type Fahrenheit is new Float;
```

Os tipos de variáveis desses dois tipos derivados não são equivalentes, apesar de suas estruturas serem idênticas. Além disso, variáveis de nenhum desses tipos são equivalentes a nenhum outro tipo de ponto flutuante. Literais são uma exceção a essa regra. Um literal como `3.0` tem o tipo real e é equivalente a qualquer tipo de ponto flutuante. Tipos derivados podem incluir também restrições de faixa dos tipos ancestrais, entretanto ainda herdam todas as operações dos ancestrais.

Um **subtipo** em Ada é uma versão possivelmente reduzida em relação à faixa de um tipo existente. Um subtipo tem equivalência com seu ancestral. Por exemplo, considere a declaração:

```
subtype Small_type is Integer range 0..99;
```

O tipo `Small_type` é equivalente ao tipo `Integer`.

Note que, em Ada, os tipos derivados são muito diferentes dos de subfaixa. Por exemplo, considere as seguintes declarações de tipo:

```
type Derived_Small_Int is new Integer range 1..100;
subtype Subrange_Small_Int is Integer range 1..100;
```

As variáveis de ambos os tipos, `Derived_Small_Int` e `Subrange_Small_Int`, têm a mesma faixa de valores válidos e ambas herdam as operações de `Integer`. Entretanto, as variáveis do tipo `Derived_Small_Int` não são compatíveis com nenhum `Integer`. Por outro lado, as variáveis do tipo `Subrange_Small_Int` são compatíveis com variáveis e constantes do tipo `Integer` e de qualquer subtipo de `Integer`.

Para variáveis de um tipo de matriz sem restrições em Ada, é usada a equivalência de tipos por estrutura. Por exemplo, considere a seguinte declaração de tipo e duas declarações de objetos:

```
type Vector is array (Integer range <>) of Integer;
Vector_1: Vector (1..10);
Vector_2: Vector (11..20);
```

Os tipos desses dois objetos são equivalentes, apesar de terem nomes e faixas de índices diferentes. Mas, para objetos de tipos matriz sem restrições, é usada a equivalência de tipos por estrutura, em vez de a equivalência por nome. Como ambos os tipos têm 10 elementos e os elementos de ambos são do tipo `Integer`, eles são equivalentes em relação ao tipo.

Para tipos anônimos restritos, Ada usa uma forma altamente restritiva de equivalência de tipos por nome. Considere as seguintes declarações Ada de tipos anônimos restritos:

```
A : array (1..10) of Integer;
```

Nesse caso, A tem um tipo anônimo, mas único, atribuído pelo compilador e não disponível para o programa. Se também tivéssemos

```
B : array (1..10) of Integer;
```

A e B seriam anônimos, mas de tipos distintos e não equivalentes, apesar de serem estruturalmente idênticos. A declaração múltipla

```
C, D : array (1..10) of Integer;
```

cria dois tipos anônimos, um para C e outro para D, que não são equivalentes. Essa declaração é, na verdade, tratada como se fossem duas:

```
C : array (1..10) of Integer;
D : array (1..10) of Integer;
```

Note que a forma de equivalência de tipos por nome de Ada é mais restritiva que a equivalência por nome definida no início desta seção. Se, em vez disso, tivéssemos escrito

```
type List_10 is array (1..10) of Integer;
C, D : List_10;
```

os tipos de C e D seriam equivalentes.

A equivalência de tipos por nome funciona bem para Ada, em parte porque todos eles, exceto as matrizes anônimas, precisam ter nomes de tipos (e o compilador dá nomes internos aos tipos anônimos).

As regras de equivalência de tipos para Ada são mais rígidas que aquelas para linguagens que têm muitas coerções entre tipos. Por exemplo, os dois operandos de um operador de adição em Java podem ser praticamente qualquer combinação de tipos numéricos da linguagem. Um dos operandos simplesmente sofrerá uma coerção para o tipo do outro. Mas em Ada não existem coerções dos operandos de uma operação aritmética.

C usa tanto a equivalência de tipos por nome quanto por estrutura. Cada declaração de **`struct`**, **`enum`** e **`union`** cria um novo tipo que não é equivalente a nenhum outro. Então, a equivalência de tipos por nome é usada para os tipos que representam estruturas, enumerações e uniões. Outros tipos não escalares usam equivalência de tipos por estrutura. Os tipos de matrizes são equivalentes se tiverem componentes do mesmo tipo. Além disso, se um tipo matriz tiver um tamanho constante, ele será equivalente a outras matrizes com o mesmo tamanho constante ou àquelas sem um tamanho constante. Note que **`typedef`** em C e C++ não introduz um novo tipo; a instrução apenas define um novo nome para um tipo existente. Então, qualquer tipo definido com **`typedef`** tem equivalência com seu tipo ancestral. Uma exceção ao C usar equivalência de tipos por nome para estruturas, enumerações e uniões ocorre se duas estruturas, enumerações ou uniões forem definidas em diferentes arquivos, o que faz com que a equivalência de

tipos por estrutura seja usada. Essa é uma brecha na regra da equivalência de tipos por nomes que permite a equivalência de estruturas, enumerações e uniões definidas em diferentes arquivos.

C++ é como C, exceto que nela não existem exceções para estruturas, enumerações e uniões definidas em diferentes arquivos.

Em linguagens que não permitem que os usuários definam e nomeiem tipos, como Fortran e COBOL, a equivalência de nomes obviamente não pode ser usada.

Linguagens orientadas a objetos, como Java e C++, trazem com elas outra questão acerca da compatibilidade de tipos: a compatibilidade de objetos e seu relacionamento com a hierarquia de herança, discutida no Capítulo 12.

A compatibilidade em expressões é discutida no Capítulo 7; a compatibilidade de tipos para parâmetros de subprogramas é discutida no Capítulo 9.

6.15 TEORIA E TIPOS DE DADOS

A teoria de tipos é uma ampla área de estudo em matemática, lógica, ciência da computação e filosofia. Ela começou na matemática, no início de 1900, e mais tarde se tornou uma ferramenta padrão na lógica. Qualquer discussão geral sobre a teoria de tipos é necessariamente complexa, longa e altamente abstrata. Mesmo quando restrita à ciência da computação, a teoria de tipos inclui assuntos tão diversos e complexos quanto cálculo lambda tipado, combinadores, metateoria de quantificação restrita (*bounded quantification*), tipos existenciais e polimorfismo de ordem mais alta (*higher-order*). Todos esses tópicos estão muito além dos objetivos deste livro.

Em ciência da computação, existem dois ramos de teoria de tipos: prático e abstrato. O prático se preocupa com tipos de dados em linguagens de programação comerciais; o abstrato se concentra principalmente em cálculo lambda tipado, uma área de pesquisa muito explorada por parte dos cientistas da computação teóricos nos últimos 50 anos. Esta seção se restringe a uma breve discussão sobre alguns dos formalismos matemáticos que são a base para os tipos de dados nas linguagens de programação.

Um tipo de dado define um conjunto de valores e uma coleção de operações sobre esses valores. Um **sistema de tipos** é um conjunto de tipos e as regras que governam seu uso em programas. Obviamente, cada linguagem de programação tipada define um sistema de tipos. O modelo formal de um sistema de tipos de uma linguagem de programação consiste em um conjunto de tipos e em uma coleção de funções que definem as regras de tipos da linguagem usadas para determinar o tipo de qualquer expressão. Um sistema formal que descreve as regras de um sistema de tipos, as chamadas gramáticas de atributos, é apresentado no Capítulo 3.

Um modelo alternativo às gramáticas de atributos usa um mapa de tipos e uma coleção de funções, não associadas às regras gramaticais, que especificam as regras de tipos. Um mapa de tipos é similar ao estado de um programa usado em semântica denotacional; consiste em um conjunto de pares ordenados em que o primeiro elemento de cada par é um nome de variável e o segundo elemento é seu tipo. Um mapa de tipos é construído por meio das declarações de tipo no programa. Em uma linguagem estaticamente tipada, o mapa precisa ser mantido apenas durante a compilação, apesar de mudar à medida que o programa é analisado pelo compilador. Se alguma verificação de tipos é feita dinamicamente, o mapa deve ser mantido durante a execução. A versão concreta

de um mapa de tipos em um sistema de compilação é a tabela de símbolos, construída principalmente pelos analisadores léxico e sintático. Os tipos dinâmicos algumas vezes são mantidos com etiquetas anexadas a valores ou objetos.

Conforme mencionado, um tipo de dados é um conjunto de valores, apesar de, em um tipo de dados, os elementos serem normalmente ordenados. Por exemplo, os elementos em todos os tipos de enumeração são ordenados. Contudo, em um conjunto matemático, os elementos não são ordenados. Apesar dessa diferença, operações de conjuntos podem ser usadas em tipos de dados para descrever novos tipos de dados. Os tipos de dados estruturados em linguagens de programação são definidos por operadores de tipo, ou construções que correspondem às operações de conjuntos. Essas operações de conjunto/construtores de tipos são brevemente apresentadas nos parágrafos seguintes.

Um mapeamento finito é uma função de um conjunto finito de valores, o conjunto domínio, para valores no conjunto de faixa. Mapeamentos finitos modelam duas categorias de tipos em linguagens de programação – funções e matrizes –, apesar de, em algumas linguagens, as funções não serem tipos. Todas as linguagens incluem matrizes, definidas em termos de uma função de mapeamento que mapeia índices para elementos na matriz. Para matrizes tradicionais, o mapeamento é simples – valores inteiros são mapeados para endereços dos elementos da matriz; para as associativas, o mapeamento é definido por uma função que descreve uma operação de resumo. A função resumo mapeia as chaves das matrizes associativas, normalmente cadeias de caracteres,[10] para endereços dos elementos da matriz.

Um produto cartesiano de n conjuntos, S_1, S_2, . . ., S_n, é um conjunto denotado por $S_1 \times S_2 \times \ldots \times S_n$. Cada elemento do conjunto do produto cartesiano tem um elemento de cada um dos conjuntos constituintes. Assim, $S_1 \times S_2 = \{(x, y) \mid x \text{ está em } S_1 \text{ e } y \text{ está em } S_2\}$. Por exemplo, se $S_1 = \{1, 2\}$ e $S_2 = \{a, b\}$, $S_1 \times S_2 = \{(1, a), (1, b), (2, a), (2, b)\}$. Um produto cartesiano define tuplas matemáticas, que aparecem em Python, ML e F# como um tipo de dados (consulte a Seção 6.5). Os produtos cartesianos também modelam registros ou estruturas, apesar de não precisamente. Produtos cartesianos não têm nomes de elementos, mas os registros os exigem. Por exemplo, considere a seguinte estrutura em C:

```
struct intFloat {
  int myInt;
  float myFloat;
};
```

Essa estrutura define o produto cartesiano cujo tipo é **`int × float`**. Os nomes dos elementos são `myInt` e `myFloat`.

A união dos dois conjuntos, S_1 e S_2, é definida como $S_1 \cup S_2 = \{x \mid x \text{ está em } S_1 \text{ ou } x \text{ está em } S_2\}$. A união de conjuntos modela os tipos de dados união, conforme descrito na Seção 6.10.

Subconjuntos matemáticos são definidos por meio do fornecimento de uma regra que os elementos devem seguir. Os conjuntos modelam os subtipos de Ada, apesar de não precisamente, dado que os subtipos devem consistir em elementos adjacentes de

[10] Em Ruby e Lua, as chaves das matrizes associativas não precisam ser cadeias de caracteres – elas podem ser de qualquer tipo.

seus conjuntos pais. Elementos de conjuntos matemáticos não são ordenados, então o modelo não é perfeito.

Note que ponteiros, definidos com operadores de tipos, como | ast | em C, não são definidos em termos de uma operação de conjunto.

Isso conclui nossa discussão sobre formalismos em tipos de dados, assim como nossa discussão completa sobre tipos de dados.

RESUMO

Os tipos de dados de uma linguagem influenciam muito a determinação de seu estilo e sua utilidade. Com as estruturas de controle, eles formam a essência de uma linguagem.

Os tipos de dados primitivos da maioria das linguagens imperativas incluem os tipos numéricos, de caracteres e booleanos. Os tipos numéricos, em geral, são diretamente suportados pelo hardware.

Os tipos de enumeração e de subfaixa definidos pelo usuário são convenientes e melhoram a legibilidade e a confiabilidade dos programas.

Matrizes fazem parte da maioria das linguagens de programação. O relacionamento entre uma referência a um elemento de matriz e o endereço desse elemento é dado em uma função de acesso, que é uma implementação de um mapeamento. As matrizes podem ser estáticas, como as de C++, cuja definição inclui o especificador **`static`**; dinâmicas da pilha fixas, como em funções de C (sem o especificador **`static`**); dinâmicas do monte fixas, como em objetos Java; ou dinâmicas do monte, como nas matrizes em Perl. A maioria das linguagens permite poucas operações em matrizes completas.

Atualmente, os registros são incluídos na maioria das linguagens. Campos de registros são especificados de diversas maneiras. No caso de COBOL, podem ser referenciados sem a nomeação de todos os registros que os envolvem, apesar de isso ser confuso de se implementar e de prejudicar a legibilidade. Em diversas linguagens que suportam programação orientada a objetos, os registros são suportados com objetos.

Tuplas são semelhantes aos registros, mas não têm nomes para suas partes constituintes. Elas fazem parte de Python, ML e F#.

Listas são fundamentais nas linguagens de programação funcionais, mas atualmente também são incluídas em Python e C#.

Uniões são estruturas que podem armazenar valores de tipos diferentes em momentos diferentes. Uniões discriminadas incluem uma etiqueta para gravar o valor de tipo atual. Uma união livre é uma união que não tem essa etiqueta. A maioria das linguagens com uniões não tem projetos seguros para elas, com exceção das linguagens ML e F#.

Ponteiros são usados para lidar com a flexibilidade e para controlar o gerenciamento de armazenamento dinâmico. Os ponteiros apresentam alguns perigos inerentes: ponteiros soltos são difíceis de ser evitados, e podem ocorrer vazamentos de memória.

Tipos de referência, como os de Java e C#, fornecem gerenciamento do monte sem os perigos dos ponteiros.

Tipos de enumeração e de registro são relativamente fáceis de implementar. Matrizes também são diretas, apesar de o acesso a seus elementos ser um processo dispendioso quando elas têm diversos índices. A função de acesso requer uma adição e uma multiplicação adicionais para cada índice.

Os ponteiros são relativamente fáceis de implementar se o gerenciamento do monte não for considerado. Esse gerenciamento é simples se todas as células forem do mesmo tamanho, mas é complicado para a alocação e a liberação de células de tamanho variável.

Tipagem forte é o conceito de exigir que todos os erros de tipos sejam detectados. O valor da tipagem forte é um aumento na confiabilidade.

As regras de equivalência de tipos determinam quais operações são válidas entre os tipos estruturados de uma linguagem. As equivalências de tipos por nome e por estrutura são as duas abordagens fundamentais para definir equivalência de tipos.

Teorias de tipos têm sido desenvolvidas em muitas áreas. Na ciência da computação, o ramo prático da teoria de tipos define os tipos e as regras dos tipos das linguagens de programação. A teoria de conjuntos pode ser usada para modelar a maioria dos tipos de dados estruturados nas linguagens de programação.

NOTAS BIBLIOGRÁFICAS

Existe uma literatura abundante que trata do projeto, do uso e da implementação de tipos de dados. Hoare fornece uma das primeiras definições sistemáticas de tipos estruturados em Dahl et al. (1972). Uma discussão geral sobre uma ampla variedade de tipos de dados aparece em Cleaveland (1986).

A implementação de verificações em tempo de execução nas possíveis inseguranças dos tipos de dados em Pascal é discutida em Fischer e LeBlanc (1980). A maioria dos livros de projeto de compiladores, como Fischer e LeBlanc (1991) e Aho et al. (1986), descreve métodos de implementação para tipos de dados, assim como fazem outros textos sobre linguagens de programação, como Pratt e Zelkowitz (2001) e Scott (2009). Uma discussão detalhada sobre os problemas de gerenciamento do monte pode ser encontrada em Tenenbaum et al. (1990). Métodos de coleta de lixo são desenvolvidos por Schorr e Waite (1967) e por Deutsch e Bobrow (1976). Uma lista abrangente de algoritmos de coleta de lixo pode ser encontrada em Cohen (1981) e Wilson (2005).

QUESTÕES DE REVISÃO

1. O que é um descritor?
2. Quais são as vantagens e as desvantagens dos tipos de dados decimais?
3. Quais são as questões de projeto para tipos de cadeias de caracteres?
4. Descreva as três opções para tamanhos de cadeias.
5. Defina *tipos ordinais*, *de enumeração* e *de subfaixa*.
6. Quais são as vantagens dos tipos de enumeração definidos pelo usuário?
7. De que maneira os tipos de enumeração definidos pelo usuário de C# são mais confiáveis que os de C++?
8. Quais são as questões de projeto para matrizes?

9. Defina *matrizes estáticas*, *dinâmicas da pilha fixas*, *dinâmicas do monte fixas* e *dinâmicas do monte*. Quais são as vantagens de cada uma delas?
10. O que acontece quando um elemento não existente de uma matriz é referenciado em Perl?
11. Como JavaScript suporta matrizes esparsas?
12. Que linguagens suportam índices negativos?
13. Que linguagens suportam fatias de matrizes com tamanhos de passos (*stepsizes*)?
14. O que é uma constante agregada?
15. Defina *ordem principal de linha* e *ordem principal de coluna*.
16. O que é uma função de acesso para uma matriz?
17. Quais são as entradas exigidas em um descritor de matriz em Java e quando elas devem ser armazenadas (em tempo de compilação ou em tempo de execução)?
18. Qual é a estrutura de uma matriz associativa?
19. Qual é o propósito dos números de níveis em registros em COBOL?
20. Defina *referências completamente qualificadas* e *elípticas* para campos em registros.
21. Qual é a principal diferença entre um registro e uma tupla?
22. As tuplas de Python são mutáveis?
23. Qual é a finalidade de um padrão de tuplas de F#?
24. Em qual linguagem imperativa as listas servem como matrizes?
25. Qual é a ação da função `CAR` de Scheme?
26. Qual é a ação da função `tl` de F#?
27. De que maneira a função `CDR` de Scheme modifica seu parâmetro?
28. No que se baseiam as compreensões de lista de Python?
29. Defina *união*, *união livre* e *união discriminada*.
30. As uniões de F# são discriminadas?
31. Quais são as questões de projeto para os tipos ponteiro?
32. Quais são os dois problemas mais comuns dos ponteiros?
33. Por que os ponteiros da maioria das linguagens são restritos de forma a apontarem para uma única variável de tipo?
34. O que é um tipo de referência em C++ e para que ele é comumente usado?
35. Por que as variáveis de referência em C++ são melhores que os ponteiros para parâmetros formais?
36. Quais vantagens as variáveis de tipo de referência em Java e C# têm em relação aos ponteiros de outras linguagens?

37. Descreva a abordagem preguiçosa e a ansiosa para recuperar lixo.
38. Por que a aritmética de referências em Java e C# não faz sentido?
39. O que é um tipo compatível?
40. Defina erro de tipo.
41. Defina uma linguagem fortemente tipada.
42. Por que Java não é fortemente tipada?
43. O que é uma conversão implícita sem conversão?
44. Quais linguagens não têm coerções de tipo?
45. Por que C e C++ não são fortemente tipadas?
46. O que é a equivalência de tipos por nome?
47. O que é a equivalência de tipos por estrutura?
48. Qual é a principal vantagem da equivalência de tipos por nome?
49. Qual é a principal desvantagem da equivalência de tipos por estrutura?
50. Para quais tipos C usa a equivalência de tipos por estrutura?
51. Que operação de conjunto modela o tipo de dados **`struct`** de C?

PROBLEMAS

1. Quais são os argumentos a favor e contra a representação de valores booleanos como bits únicos em memória?
2. Como um valor decimal perde espaço em memória?
3. Os minicomputadores VAX usam um formato para números de ponto flutuante que não é o mesmo do padrão IEEE. Qual é o formato e por que ele foi escolhido pelos projetistas dos computadores VAX? Uma referência para representações de ponto flutuante em VAX é Sebesta (1991).
4. Compare os métodos de lápides e fechaduras e chaves para evitar ponteiros soltos, do ponto de vista da segurança e do custo de implementação.
5. Que desvantagens existem no desreferenciamento implícito de ponteiros, mas apenas em certos contextos?
6. Explique todas as diferenças entre os subtipos e os tipos derivados em Ada.
7. Que justificativa significativa existe para o operador −> em C e C++?
8. Quais são as diferenças entre os tipos de enumeração de C++ e os de Java?
9. Matrizes multidimensionais podem ser armazenadas em ordem principal de linha, como em C++, ou em ordem principal de coluna, como em Fortran. Desenvolva a função de acesso para ambas as disposições para matrizes tridimensionais.

10. Na linguagem Burroughs Extended ALGOL, as matrizes são armazenadas como uma matriz de uma dimensão de ponteiros para suas linhas, as quais são tratadas como matrizes de uma dimensão de valores. Quais são as vantagens e desvantagens de tal esquema?
11. Analise e escreva uma comparação das funções `malloc` e `free` de C com os operadores **`new`** e **`delete`** de C++. Use a segurança como principal consideração na comparação.
12. Analise e escreva uma comparação do uso de ponteiros C++ com o de variáveis de referência em Java para referenciar variáveis dinâmicas do monte fixas. Use a segurança como principal consideração na comparação.
13. Redija um texto curto sobre o que foi perdido e o que foi ganho na decisão dos projetistas de Java de não incluírem os ponteiros de C++.
14. Quais são os argumentos a favor e contra a recuperação de armazenamento do monte implícita de Java, quando comparada com a recuperação de armazenamento do monte explícita exigida em C++? Considere sistemas de tempo real.
15. Quais são os argumentos para a inclusão dos tipos de enumeração em C#, apesar de não estarem disponíveis nas primeiras versões de Java?
16. Qual é sua expectativa em relação ao nível de uso de ponteiros em C#? Com que frequência eles serão usados quando não forem absolutamente necessários?
17. Crie duas listas de aplicações de matrizes, uma para aquelas que exigem matrizes irregulares e outra para aquelas que exigem matrizes retangulares. Agora, discuta se apenas matrizes irregulares, apenas matrizes regulares ou ambas devem ser incluídas em uma linguagem de programação.
18. Compare as capacidades de manipulação de cadeias das bibliotecas de classes de C++, Java e C#.
19. Pesquise a definição de *fortemente tipada*, conforme dada por Gehani (1983), e compare-a com a definição apresentada neste capítulo. De que forma elas são diferentes?
20. De que forma a verificação de tipos estática é melhor que a verificação de tipos dinâmica?
21. Explique como as regras de coerção podem enfraquecer o efeito benéfico da tipagem forte.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Projete um conjunto simples de programas de teste para determinar as regras de compatibilidade de tipos de um compilador C ao qual você tenha acesso. Escreva um relatório sobre suas descobertas.

2. Determine se algum compilador C ao qual você tenha acesso implementa a função `free`.
3. Escreva um programa que efetue a multiplicação de matrizes em alguma linguagem que faça verificação de faixas de índices e para a qual você possa obter uma versão em linguagem de montagem ou em linguagem de máquina a partir do compilador. Determine o número de instruções necessárias para a verificação de faixa de índices e compare com o número total de instruções para o processo de multiplicação de matrizes.
4. Se você tem acesso a um compilador no qual o usuário pode especificar se a verificação de faixas de índices é desejada, escreva um programa que faça um grande número de acessos a matrizes e cronometre sua execução. Execute o programa com a verificação de faixa de índices e sem ela e compare os tempos.
5. Escreva um programa simples em C++ para investigar a segurança de seus tipos de enumeração. Inclua ao menos 10 operações diferentes em tipos de enumeração para determinar se ocorrências incorretas ou incoerentes são válidas. Agora, escreva um programa em C# que faça as mesmas coisas e execute-o para determinar quantas ocorrências incorretas ou incoerentes são válidas. Compare seus resultados.
6. Escreva um programa em C++ ou C# que inclua dois tipos de enumeração diferentes e que tenha um número significativo de operações usando os tipos de enumeração. A seguir, escreva o mesmo programa utilizando apenas variáveis inteiras. Compare a legibilidade e faça uma previsão das diferenças em torno da confiabilidade entre os dois programas.
7. Escreva um programa em C que faça um grande número de referências para elementos de matrizes de duas dimensões usando apenas índices. Escreva um segundo programa que efetue as mesmas operações, mas que use ponteiros e aritmética de ponteiros para a função de mapeamento de armazenamento, a fim de fazer as referências aos elementos da matriz. Compare a eficiência em termos de tempo dos dois programas. Qual dos dois é mais confiável? Por quê?
8. Escreva um programa em Perl que use uma dispersão (*hash*) e um grande número de operações nessa dispersão. Por exemplo, a dispersão poderia armazenar o nome e a idade de pessoas. Um gerador de números aleatórios poderia ser usado para criar nomes de três caracteres e idades, que poderiam ser adicionados à dispersão. Quando um nome duplicado é gerado, causa um acesso à dispersão, mas não adiciona um novo elemento. Reescreva o mesmo programa sem usar dispersões. Compare a eficiência de execução dos dois. Compare a facilidade de programar e a legibilidade de ambos.
9. Escreva um programa na linguagem de sua escolha, que se comporte diferentemente se a linguagem usar equivalência de nomes ou se usar equivalência estrutural.
10. Para que tipos de A e B a sentença de atribuição simples A = B é válida em C++, mas não em Java?

7

Expressões e sentenças de atribuição

7.1 Introdução

7.2 Expressões aritméticas

7.3 Operadores sobrecarregados

7.4 Conversões de tipos

7.5 Expressões relacionais e booleanas

7.6 Avaliação em curto-circuito

7.7 Sentenças de atribuição

7.8 Atribuição de modo misto

Como o título indica, este capítulo trata de sentenças de atribuição e de expressão. Primeiro, são discutidas as regras semânticas que determinam a ordem de avaliação dos operadores em expressões. A seguir, é feita uma explanação sobre um problema em potencial na ordem de avaliação dos operandos, que ocorre quando as funções têm efeitos colaterais. Operadores sobrecarregados, tanto predefinidos quanto definidos pelo usuário, são então discutidos, assim como seus efeitos nas expressões em programas. A seguir, expressões de modo misto são descritas e avaliadas. Isso leva à definição e à avaliação de conversões de tipo de alargamento e de estreitamento, tanto implícitas quanto explícitas. Expressões relacionais e booleanas também são discutidas, incluindo o processo de avaliação em curto-circuito. Por fim, são abordadas as sentenças de atribuição, desde sua forma mais simples até todas suas variações, incluindo atribuições como expressões e atribuições de modo misto.

Expressões de casamento de padrões de cadeias de caracteres foram abordadas quando discorremos sobre cadeias de caracteres, no Capítulo 6; então, não são mencionadas neste capítulo.

7.1 INTRODUÇÃO

Expressões são os meios fundamentais de especificar computações em uma linguagem de programação. É crucial para um programador entender tanto a sintaxe quanto a semântica de expressões da linguagem que utiliza. Um mecanismo formal (BNF) para descrever a sintaxe de expressões foi apresentado no Capítulo 3. Neste capítulo, são discutidas as semânticas de expressões.

Para entender a avaliação de expressões, é necessário conhecer as ordens de avaliação de operadores e operandos. A ordem de avaliação de operadores de expressões é ditada pelas regras de associatividade e de precedência da linguagem. Apesar de o valor de uma expressão algumas vezes depender disso, a ordem de avaliação dos operandos em expressões muitas vezes não é mencionada pelos projetistas de linguagens. Isso permite que os implementadores escolham a ordem, o que leva à possibilidade de os programas produzirem resultados diferentes em implementações diferentes. Outras questões relacionadas à semântica de expressões são diferenças de tipos, coerções e avaliação em curto-circuito.

A essência das linguagens de programação imperativas é o papel dominante das sentenças de atribuição. A finalidade dessas sentenças é causar o efeito colateral de alterar os valores de variáveis, ou o estado, do programa. Então, uma parte integrante de todas as linguagens imperativas é o conceito de variáveis cujos valores mudam durante a execução de um programa.

Linguagens funcionais usam variáveis de um tipo diferente, como os parâmetros de funções. Essas linguagens também têm sentenças de declaração que vinculam valores a nomes. Essas declarações são semelhantes às sentenças de atribuição, mas não têm efeitos colaterais.

7.2 EXPRESSÕES ARITMÉTICAS

A avaliação automática de expressões aritméticas similares àquelas encontradas na matemática, na ciência e nas engenharias foi um dos principais objetivos das primeiras

linguagens de programação de alto nível. A maioria das características das expressões aritméticas em linguagens de programação foi herdada de convenções que evoluíram na matemática. Em linguagens de programação, as expressões aritméticas consistem em operadores, operandos, parênteses e chamadas a funções. Um operador pode ser **unário**, por possuir um único operando, **binário**, por ter dois operandos, ou **ternário**, por ter três operandos.

Na maioria das linguagens de programação, os operadores binários usam a notação convencional (**infix**), ou seja, eles aparecem entre seus operandos. Uma exceção é Perl, que possui alguns operadores **prefixados*** (prefix) precedendo seus operandos.

O propósito de uma expressão aritmética é especificar uma computação aritmética. Uma implementação de tal computação deve realizar duas ações: obter os operandos, normalmente a partir da memória, e executar neles as operações aritméticas. Nas próximas seções, são investigados os detalhes comuns de projeto de expressões aritméticas.

A seguir, são listadas as principais questões de projeto para as expressões aritméticas discutidas nesta seção:

- Quais são as regras de precedência de operadores?
- Quais são as regras de associatividade de operadores?
- Qual é a ordem de avaliação dos operandos?
- Existem restrições acerca de efeitos colaterais na avaliação de operandos?
- A linguagem permite a sobrecarga de operadores definida pelo usuário?
- Que tipo de mistura de tipos é permitida nas expressões?

7.2.1 Ordem de avaliação de operadores

As regras de precedência e de associatividade para operadores de uma linguagem determinam a ordem de avaliação de seus operadores.

7.2.1.1 Precedência

O valor de uma expressão depende, ao menos parcialmente, da ordem de avaliação dos operadores na expressão. Considere a expressão:

```
a + b * c
```

Suponha que as variáveis `a`, `b` e `c` tenham os valores `3`, `4` e `5`, respectivamente. Se a expressão for avaliada da esquerda para a direita (a adição primeiro e depois a multiplicação), o resultado é `35`. Se for avaliada da direita para a esquerda, o resultado é `23`.

Em vez de simplesmente avaliar os operadores em uma expressão da esquerda para a direita ou da direita para a esquerda, os matemáticos desenvolveram o conceito de hierarquizar os operadores por prioridades de avaliação e basear a avaliação da ordem das expressões parcialmente nessa hierarquia. Por exemplo, na matemática, a multiplicação tem prioridade maior que a adição, talvez por seu nível mais alto de complexidade. Se essa convenção fosse aplicada à expressão do exemplo anterior, como seria o caso na maioria das linguagens de programação, a multiplicação seria efetuada primeiro.

*N. de T.: Também conhecida como notação polonesa.

As **regras de precedência de operadores** para avaliação de expressões definem parcialmente a ordem pela qual os operadores de diferentes níveis de precedência são avaliados. As regras de precedência de operadores para expressões são baseadas na hierarquia de prioridades dos operadores, conforme vista pelo projetista da linguagem. As regras de precedência de operadores das linguagens imperativas comuns são praticamente todas iguais, porque são baseadas nas da matemática. Nessas linguagens, a exponenciação tem a precedência mais alta (quando fornecida pela linguagem), seguida pela multiplicação e pela divisão no mesmo nível, depois pela adição e pela subtração binária no mesmo nível.

Muitas linguagens também incluem versões unárias da adição e da subtração. A adição unária é chamada de **operador identidade**, porque normalmente não tem uma operação associada e não faz efeito em seu operando. Ellis e Stroustrup (1990, p. 56), falando sobre C++, chamaram tal operador de acidente histórico e acertaram ao classificá-lo como inútil. A subtração unária, é claro, modifica o sinal de seu operando. Em Java e C#, ela causa também a conversão implícita de operandos dos tipos **`short`** e **`byte`** no tipo **`int`**.

Em todas as linguagens imperativas comuns, o operador unário de subtração pode aparecer no início de uma expressão ou em qualquer lugar dentro dela, desde que seja entre parênteses para evitar que fique ao lado de outro operador. Por exemplo,

```
a + (- b) * c
```

é válido, mas

```
a + - b * c
```

normalmente não é.

A seguir, considere as expressões:

```
- a / b
- a * b
- a ** b
```

Nos dois primeiros casos, a precedência relativa do operador unário de subtração e do operador binário é irrelevante – a ordem de avaliação dos dois operadores não tem efeito no valor da expressão. No último caso, entretanto, a ordem importa.

Entre as linguagens de programação mais usadas, apenas Fortran, Ruby, Visual Basic e Ada têm o operador de exponenciação. Nas quatro, a exponenciação tem precedência mais alta que a subtração unária, então

```
- A ** B
```

é equivalente a

```
-(A ** B)
```

As precedências dos operadores aritméticos de Ruby e das linguagens baseadas em C são:

	Ruby	*Linguagens baseadas em C*
Mais alta	`**`	++ e −− posfixados
	+ e − unários	++ e −− prefixados, + e − unários
	`*`, `/`, `%`	`*`, `/`, `%`
Mais baixa	+ e − binários	+ e − binários

O operador `**` é a exponenciação. O operador `%` recebe dois operandos inteiros e produz o resto do primeiro, após a divisão pelo segundo.[1] Os operadores ++ e −− das linguagens baseadas em C são descritos na Seção 7.7.4.

APL difere das outras linguagens, porque tem só um nível de precedência, como ilustrado na seção a seguir.

A precedência diz respeito a apenas algumas das regras para a ordem de avaliação de operadores; as regras de associatividade também a afetam.

7.2.1.2 Associatividade

Considere a expressão:

```
a - b + c - d
```

Se os operadores de adição e subtração têm o mesmo nível de precedência, como nas linguagens de programação, as regras de precedência nada dizem a respeito da ordem de avaliação dos operadores nessa expressão.

Quando uma expressão contém duas ocorrências adjacentes[2] de operadores com o mesmo nível de precedência, a questão sobre qual operador é avaliado primeiro é respondida pelas regras de **associatividade** da linguagem. Um operador pode ter associatividade à direita ou à esquerda, o que significa que, quando existem dois operadores adjacentes com a mesma precedência, o da esquerda é avaliado primeiro ou o da direita é avaliado primeiro, respectivamente.

A associatividade nas linguagens mais usadas é da esquerda para a direita, exceto quando o operador de exponenciação (quando fornecido) associa da direita para a esquerda. Na expressão em Java

```
a - b + c
```

o operador esquerdo é avaliado primeiro.

A exponenciação em Fortran e Ruby é associativa à direita; então, na expressão

```
A ** B ** C
```

o operador da direita é avaliado primeiro.

Em Visual Basic, o operador de exponenciação, `^`, é associativo à esquerda.

[1] Em versões de C antes de C99, o operador `%` era dependente da implementação em algumas situações, porque a divisão também era dependente dela.

[2] Dizemos que dois operadores são "adjacentes" se eles são separados por um único operando.

As regras de associatividade para algumas linguagens bastante utilizadas são:

Linguagem	*Regra de associatividade*
Ruby	Esquerda: `*`, `/`, `+`, `-`
	Direita: `**`
Linguagens baseadas em C	Esquerda: `*`, `/`, `%`, + binário, − binário
	Direita: `++`,`--`, − unário, + unário

Conforme mencionado na Seção 7.2.1.1, em APL todos os operadores têm o mesmo nível de precedência. Logo, a ordem de avaliação dos operadores em expressões APL é determinada inteiramente pela regra de associatividade, da direita para a esquerda para todos os operadores. Por exemplo, na expressão

```
A × B + C
```

o operador de adição é avaliado primeiro, seguido pelo operador de multiplicação (`×` é o operador de multiplicação em APL). Se `A` fosse `3`, `B` fosse `4` e `C` fosse `5`, o valor dessa expressão em APL seria `27`.

Muitos compiladores para as linguagens mais empregadas fazem uso do fato de que alguns operadores aritméticos são matematicamente associativos, ou seja, as regras de associatividade não têm impacto no valor de uma expressão que contém apenas esses operadores. Por exemplo, a adição é matematicamente associativa, então na matemática o valor da expressão

```
A + B + C
```

não depende da ordem de avaliação dos operadores. Se operações de ponto flutuante para operações matematicamente associativas também fossem associativas, o compilador poderia usar esse fato para realizar algumas otimizações simples. Especificamente, se é permitido ao compilador reordenar a avaliação de operadores, ele pode produzir um código um pouco mais rápido para a avaliação de expressões. Os compiladores comumente realizam tais tipos de otimizações.

Infelizmente, em um computador, tanto as representações em ponto flutuante quanto as operações aritméticas de ponto flutuante são apenas aproximações de seus correspondentes matemáticos (devido às limitações de tamanho). O fato de um operador matemático ser associativo não implica necessariamente que uma operação de ponto flutuante correspondente seja associativa. Na verdade, apenas se todos os operandos e os resultados intermediários puderem ser representados exatamente em notação de ponto flutuante o processo será precisamente associativo. Por exemplo, existem situações patológicas nas quais a adição de inteiros em um computador *não* é associativa. Suponha que um programa precise avaliar a expressão

```
A + B + C + D
```

e que `A` e `C` sejam números positivos muito grandes, e `B` e `D` sejam números negativos com valores absolutos muito altos. Nessa situação, somar `B` a `A` não causa uma exceção de transbordamento (overflow), mas somar `C` a `A`, sim. De maneira similar, somar `C` a `B` não causa transbordamento, mas somar `D` a `B`, sim. Devido às limitações

da aritmética computacional, a adição é catastroficamente não associativa nesse caso. Logo, se o compilador reordenar essas operações de adição, isso afetará o valor da expressão. Esse problema, é claro, pode ser evitado pelo programador, supondo que os valores aproximados das variáveis sejam conhecidos. O programador pode especificar a expressão em duas partes (em duas sentenças de atribuição), garantindo que o transbordamento seja evitado. Entretanto, essa situação pode surgir de maneiras muito mais sutis, nas quais é bem menos provável que o programador note a dependência em relação à ordem.

7.2.1.3 Parênteses

Os programadores podem alterar as regras de precedência e de associatividade colocando parênteses em expressões. Uma parte com parênteses de uma expressão tem precedência em relação às suas partes adjacentes sem parênteses. Por exemplo, apesar de a multiplicação ter precedência em relação à adição, na expressão

```
(A + B) * C
```

a adição será avaliada primeiro. Matematicamente, isso é perfeitamente natural. Nessa expressão, o primeiro operando do operador de multiplicação não estará disponível até que a adição da subexpressão que contém parênteses seja avaliada. Além disso, a expressão da Seção 7.2.1.2 poderia ser especificada como

```
(A + B) + (C + D)
```

para evitar transbordamento.

Linguagens que permitem parênteses em expressões aritméticas podem dispensar todas as regras de precedência e simplesmente associar todos os operadores da esquerda para a direita ou da direita para a esquerda. O programador especificaria a ordem desejada da avaliação com parênteses. Essa abordagem seria simples porque nem o autor nem os leitores dos programas precisariam se lembrar de regras de precedência ou de associatividade. A desvantagem desse esquema é que ele torna a escrita de expressões mais tediosa e compromete seriamente a legibilidade do código. Mesmo assim, essa foi a escolha feita por Ken Iverson, o projetista de APL.

7.2.1.4 Expressões em Ruby

Lembre-se de que Ruby é uma linguagem orientada a objetos pura, ou seja, todos os valores de dados, incluindo literais, são objetos. Ela oferece suporte para a coleção de operações aritméticas e lógicas incluída nas linguagens baseadas em C. O que diferencia Ruby das linguagens baseadas em C na área de expressões é que todos os operadores aritméticos, relacionais e de atribuição, assim como os índices de matrizes, deslocamentos e operadores lógicos bit a bit, são implementados como métodos. Por exemplo, a expressão `a + b` é uma chamada ao método + do objeto referenciado por `a`, passando o objeto referenciado por `b` como parâmetro.

Um resultado interessante da implementação de operadores como métodos é que eles podem ser sobrescritos por programas de aplicação. Portanto, esses operadores podem ser redefinidos. Embora não seja sempre útil redefini-los para tipos predefinidos, conforme veremos na Seção 7.3, é útil definir operadores predefinidos para tipos defi-

nidos pelo usuário, o que pode ser feito com a sobrecarga de operadores, em algumas linguagens.

7.2.1.5 Expressões em Lisp

Assim como acontece em Ruby, todas as operações lógicas e aritméticas em Lisp são efetuadas por subprogramas. Mas, em Lisp, os subprogramas devem ser chamados explicitamente. Por exemplo, para especificar a expressão C `a + b * c` em Lisp, deve-se escrever a seguinte expressão:[3]

```
(+ a (* b c))
```

Nessa expressão, + e * são os nomes de funções.

7.2.1.6 Expressões condicionais

Sentenças **if–then–else** podem ser usadas para realizar uma atribuição baseada em expressão condicional. Por exemplo, considere

```
if (count == 0)
  average = 0;
else
  average = sum / count;
```

Nas linguagens baseadas em C, esse código poderia ser especificado mais convenientemente em uma sentença de atribuição por meio de uma expressão condicional, com a seguinte forma:

expressão_1 ? expressão_2 : expressão_3

onde expressão_1 é interpretada como uma expressão booleana. Se a expressão_1 for avaliada como verdadeira, o valor da expressão inteira será o valor da expressão_2; caso contrário, será o valor da expressão_3. Por exemplo, o efeito do exemplo **if–then–else** pode ser atingido com a seguinte sentença de atribuição, por meio de uma expressão condicional:

```
average = (count == 0) ? 0 : sum / count;
```

Na prática, a interrogação denota o início da cláusula **then** e os dois pontos marcam o início da cláusula **else**. Ambas as cláusulas são obrigatórias. Note que o sinal **?** é usado em expressões condicionais como um operador ternário.

Expressões condicionais podem ser usadas em qualquer lugar de um programa (em uma linguagem baseada em C) no qual qualquer outra expressão possa ser usada. Além de nas linguagens baseadas em C, expressões condicionais são fornecidas em Perl, JavaScript e Ruby.

[3]Quando uma lista é interpretada como código em Lisp, o primeiro elemento é o nome da função e os outros são parâmetros para a função.

7.2.2 Ordem de avaliação de operandos

Uma característica de projeto de expressões menos discutida é a ordem de avaliação dos operandos. As variáveis em expressões são avaliadas por meio da obtenção de seus valores a partir da memória. As constantes são algumas vezes avaliadas da mesma maneira. Em outros casos, uma constante pode fazer parte da instrução de linguagem de máquina e não exigir uma busca em memória. Se um operando é uma expressão entre parênteses, todos os operadores que ela contém devem ser avaliados antes que seu valor possa ser usado como operando.

Se nenhum dos operandos de um operador tiver efeitos colaterais, a ordem de avaliação dos operandos é irrelevante. Logo, o único caso interessante surge quando a avaliação de um operando tem efeitos colaterais.

7.2.2.1 Efeitos colaterais

Um **efeito colateral** de uma função, chamado de efeito colateral funcional, ocorre quando a função modifica um de seus parâmetros ou uma variável global. (Uma variável global é declarada fora da função, mas é acessível na função.)

Considere a expressão:

```
a + fun(a)
```

Se `fun` não tem o efeito colateral de modificar `a`, a ordem de avaliação dos dois operandos, `a` e `fun(a)`, não tem efeito no valor da expressão. No entanto, se `fun` modifica `a`, existe um efeito. Considere a seguinte situação: `fun` retorna `10` e modifica o valor de seu parâmetro para `20`. Suponha que tivéssemos o seguinte:

```
a = 10;
b = a + fun(a);
```

Então, se o valor de `a` for obtido primeiro (no processo de avaliação da expressão), seu valor será `10` e o valor da expressão será `20`. Mas, se o segundo operando for avaliado primeiro, o valor do primeiro operando será `20` e o valor da expressão será `30`.

O seguinte programa em C ilustra o mesmo problema quando uma função modifica uma variável global que aparece em uma expressão:

```
int a = 5;
int fun1() {
   a = 17;
   return 3;
} /* fim de fun1 */
 void main() {
   a = a + fun1();
} /* fim de main */
```

O valor calculado para `a` em `main` depende da ordem de avaliação dos operandos na expressão `a + fun1()`. O valor de `a` será `8` (se `a` for avaliado primeiro) ou `20` (se a chamada à função for avaliada primeiro).

Note que, na matemática, funções não têm efeitos colaterais, porque nela não existe a noção de variáveis. O mesmo é válido para linguagens de programação funcionais. Em

ambas, as funções são muito mais fáceis de serem analisadas e entendidas do que as funções das linguagens imperativas, porque seu contexto é irrelevante para seu significado.

Existem duas soluções possíveis para o problema da ordem de avaliação de operandos e efeitos colaterais. Primeiro, o projetista da linguagem poderia impedir que a avaliação afetasse o valor das expressões simplesmente pela proibição de efeitos colaterais funcionais. Segundo, a definição da linguagem poderia determinar que os operandos em expressões devem ser avaliados em uma ordem específica e exigir que os implementadores garantam essa ordem.

Impedir efeitos colaterais funcionais nas linguagens imperativas é difícil e diminui a flexibilidade para o programador. Considere o caso de C e C++, as quais têm apenas funções; isso significa que todos os subprogramas retornam um valor. Para eliminar os efeitos colaterais de parâmetros bidirecionais e ainda assim fornecer subprogramas que retornem mais de um valor, os valores precisariam ser colocados em uma estrutura e ela deveria ser retornada. O acesso a variáveis globais em funções também precisaria ser proibido. Entretanto, quando a eficiência é importante, usar o acesso a variáveis globais para evitar a passagem de parâmetros é um método importante para aumentar a velocidade de execução. Em compiladores, por exemplo, o acesso global a dados como a tabela de símbolos é comum e frequente.

O problema de ter uma ordem de avaliação estrita é que algumas técnicas de otimização de código usadas pelos compiladores envolvem a reordenação da avaliação de operandos. Uma ordem garantida não permite esses métodos de otimização quando chamadas a funções estão envolvidas. Logo, não existe uma solução perfeita, como podemos ver no projeto das linguagens atuais.

A definição da linguagem Java garante que os operandos sejam avaliados da esquerda para a direita, eliminando o problema discutido nesta seção.

7.2.2.2 Transparência referencial e efeitos colaterais

O conceito de transparência referencial está relacionado e é afetado pelos efeitos colaterais funcionais. Um programa tem a propriedade de **transparência referencial** se quaisquer duas expressões que têm o mesmo valor puderem ser substituídas uma pela outra em qualquer lugar no programa, sem afetar a ação deste. O valor de uma função transparente referencialmente depende de seus parâmetros.[4] A conexão da transparência referencial e dos efeitos colaterais funcionais está ilustrada no seguinte exemplo:

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

Se a função `fun` não tiver efeitos colaterais, `result1` e `result2` serão iguais, porque as expressões atribuídas a elas são equivalentes. Entretanto, suponha que `fun` tem o efeito colateral de somar `1` a `b` ou a `c`. Então `result1` não seria igual a `result2`. Esse efeito colateral viola a transparência referencial do programa em que o código aparece.

Existem diversas vantagens de se ter programas transparentes referencialmente. A mais importante é que sua semântica é muito mais fácil de entender do que a dos não

[4] Além disso, o valor da função não pode depender da ordem em que os seus parâmetros são avaliados.

transparentes referencialmente. Ser transparente referencialmente torna uma função equivalente a uma função matemática, em termos de facilidade de entendimento.

Como não têm variáveis, os programas escritos em linguagens funcionais puras são transparentes referencialmente. Funções em uma linguagem funcional pura não têm estados, os quais seriam armazenados em variáveis locais. Se uma função assim usa um valor de fora dela, esse valor deve ser uma constante, visto que não existem variáveis. Logo, o valor da função depende dos valores de seus parâmetros.

A transparência referencial será discutida em mais detalhes no Capítulo 15.

7.3 OPERADORES SOBRECARREGADOS

Operadores aritméticos são usados para mais de um propósito. Por exemplo, o sinal + é usado para especificar a adição tanto de inteiros quanto de valores de ponto flutuante. Algumas linguagens – Java, por exemplo – também o usam para concatenação de cadeias. Esse uso múltiplo de um operador é chamado de **sobrecarga de operadores**, uma prática considerada aceitável, desde que nem legibilidade nem confiabilidade sejam comprometidas.

Como um exemplo dos possíveis perigos da sobrecarga, considere o uso do e comercial (`&`) em C++. Como um operador binário, ele especifica uma operação lógica E (AND) bit a bit. Como um operador unário, entretanto, seu significado é totalmente diferente. Como um operador unário com uma variável como seu operando, o valor da expressão é o endereço dessa variável. Nesse caso, o e comercial é chamado de operador endereço-de. Por exemplo, a execução de

```
x = &y;
```

faz com que o endereço de `y` seja colocado em `x`. Existem dois problemas nesse uso múltiplo do e comercial. Primeiro, usar o mesmo símbolo para duas operações completamente não relacionadas é prejudicial para a legibilidade. Segundo, o simples erro de deixar de fora o primeiro operando de um E bit a bit pode passar despercebido pelo compilador, porque o e comercial é então interpretado como um operador endereço-de. Esse erro pode ser de difícil diagnóstico.

Praticamente todas as linguagens de programação têm um problema menos sério, mas similar, em geral relacionado à sobrecarga do operador de subtração. O problema é que o compilador não consegue dizer se o operador é binário ou unário.[5] Então, mais uma vez, a não inclusão do primeiro operando quando o operador deveria ser binário não pode ser detectada como erro pelo compilador. Entretanto, os significados das duas operações, unária e binária, são, no mínimo, intimamente relacionados, não prejudicando seriamente a legibilidade.

Algumas linguagens capazes de suportar tipos abstratos de dados (veja o Capítulo 11), como C++, C# e F#, permitem ao programador sobrecarregar símbolos de operadores. Por exemplo, suponha que um usuário quer definir o operador `*` entre um inteiro escalar e uma matriz de inteiros, de forma a significar que cada elemento da matriz seja multiplicado por esse escalar. Tal operador poderia ser definido escrevendo-se um sub-

[5]ML reduz esse problema usando símbolos diferentes para operadores de subtração unário e binário: til (~) para unário e travessão (–) para binário.

programa chamado `*`, responsável por essa nova operação. O compilador escolherá o significado correto quando um operador sobrecarregado for especificado com base nos tipos dos operandos, assim como ocorre com os operadores sobrecarregados definidos pela linguagem. Por exemplo, se essa nova definição para `*` for feita em um programa C#, um compilador C# usará a nova definição de `*` sempre que o operador aparecer com um inteiro simples como operando da esquerda e uma matriz de inteiros como operador da direita.

Quando usada com bom-senso, a sobrecarga de operadores definida pelo usuário pode melhorar a legibilidade. Por exemplo, se + e `*` fossem sobrecarregados para um tipo de dados abstrato matriz e `A`, `B`, `C` e `D` fossem variáveis desse tipo, então

```
A * B + C * D
```

poderia ser usada em vez de

```
MatrixAdd(MatrixMult(A, B), MatrixMult(C, D))
```

Por outro lado, a sobrecarga definida pelo usuário pode ser prejudicial à legibilidade. Por exemplo, nada impede que um usuário defina + como multiplicação. Além disso, ao ver um operador `*` em um programa, o leitor deve encontrar ambos os tipos de operandos e a definição do operador para determinar seus significados. Alguma ou todas essas definições podem estar em arquivos diferentes.

Outra consideração é o processo de construção de um sistema de software a partir de módulos criados por grupos diferentes. Se grupos diferentes sobrecarregaram o mesmo operador de maneiras distintas, essas diferenças obviamente precisarão ser eliminadas antes de juntar as peças do sistema como um todo.

C++ tem alguns operadores que não podem ser sobrecarregados. Entre eles estão o operador de membro de classe ou de estrutura (.) e o operador de resolução de escopo (::). Curiosamente, a sobrecarga de operadores foi um dos recursos de C++ não reproduzido em Java. Entretanto, ela reapareceu em C#.

A implementação de sobrecarga de operadores definida pelo usuário é discutida no Capítulo 9.

7.4 CONVERSÕES DE TIPOS

As conversões de tipos são de estreitamento ou alargamento. Uma **conversão de estreitamento** converte um valor em um tipo que não pode armazenar aproximações equivalentes a todos os valores do tipo original. Por exemplo, converter um **`double`** em um **`float`** em Java é uma conversão de estreitamento, porque a faixa do tipo **`double`** é muito maior que a de **`float`**. Uma **conversão de alargamento** converte um valor em um tipo que pode incluir ao menos aproximações de todos os valores do tipo original. Por exemplo, converter um **`int`** em um **`float`** em Java é uma conversão de alargamento. Conversões de alargamento são quase sempre seguras, mantendo a magnitude aproximada do valor convertido. Conversões de estreitamento nem sempre são seguras – algumas vezes, a magnitude do valor convertido é modificada no processo. Por exemplo, se o valor de ponto flutuante 1.3E25 for convertido em um inteiro em um programa Java, o resultado não será de nenhum modo relacionado ao valor original.

Apesar de as conversões de alargamento serem normalmente seguras, elas podem resultar em uma precisão reduzida. Em muitas implementações de linguagens, apesar de as conversões de inteiro em ponto flutuante serem de alargamento, alguma precisão pode ser perdida. Por exemplo, em muitos casos, inteiros são armazenados em 32 bits, o que permite ao menos nove dígitos decimais de precisão. Mas os valores de ponto flutuante também são armazenados em 32 bits, com apenas cerca de sete dígitos de precisão (devido ao espaço usado para o expoente). Então, alargamentos de inteiro para ponto flutuante podem resultar na perda de dois dígitos de precisão.

Coerções de tipos não primitivos são, é claro, mais complexas. No Capítulo 5, foram discutidas as complicações da compatibilidade de atribuição de matrizes e registros. Existe também a questão acerca de que tipos de parâmetros e de retorno permitem a sobrescrita de um método de uma superclasse – apenas quando os tipos forem os mesmos, ou também em outras situações. Essa questão, assim como o conceito de subclasses como subtipos, é discutida no Capítulo 12.

Conversões de tipo podem ser explícitas ou implícitas. As duas subseções seguintes discutem essas formas de conversões de tipo.

7.4.1 Coerção em expressões

Uma das decisões de projeto relacionadas às expressões aritméticas é se um operador pode ter operandos de tipos diferentes. Linguagens que permitem tais expressões, chamadas de **expressões de modo misto**, devem definir convenções para conversões de tipo em operandos implícitos, porque os computadores não têm operações binárias que recebam operandos de tipos diferentes. Lembre-se de que, no Capítulo 5, a coerção foi definida como uma conversão de tipo implícita iniciada pelo compilador ou pelo sistema de tempo de execução. Conversões de tipo explícitas solicitadas pelo programador são referidas como conversões explícitas (*casts*), não como coerções.

Apesar de alguns símbolos de operações poderem ser sobrecarregados, assumimos que um sistema de computação, seja em hardware ou em algum nível de simulação de software, tem uma operação para cada tipo de operando e operador definido na linguagem.[6] Para operadores sobrecarregados em uma linguagem que usa vinculação de tipos estática, o compilador escolhe o tipo correto de operação com base nos tipos dos operandos. Quando os dois operandos de um operador não são do mesmo tipo e isso é válido na linguagem, o compilador deve escolher um deles para sofrer coerção e gerar o código para essa coerção. Na discussão a seguir são examinadas as escolhas de projeto relativas à coerção de diversas linguagens comuns.

Os projetistas de linguagens não entraram em um acordo a respeito da questão das coerções em expressões aritméticas. Aqueles que são contra uma ampla faixa de coerções estão preocupados com problemas de confiabilidade que podem resultar de tais coerções, porque elas reduzem os benefícios da verificação de tipos. Aqueles que preferem incluir uma ampla faixa de coerções estão mais preocupados com a perda de flexibilidade que resulta de tais restrições. A questão é se os programadores devem se preocupar com essa categoria de erros ou se o compilador deve detectá-los.

[6] Essa suposição não é verdade para muitas linguagens. Um exemplo é dado mais adiante nesta seção.

Como uma simples ilustração do problema, considere o código Java a seguir:

```
int a;
float b, c, d;
. . .
d = b * a;
```

Suponha que o segundo operando de um operador de multiplicação devesse ser `c`, mas, por engano, foi digitado um `a`. Como expressões de modo misto são válidas em Java, o compilador não pode detectar isso como um erro. Ele simplesmente inseriria código para realizar uma coerção do valor do operando **`int`**, `a`, para um **`float`**. Se as expressões de modo misto não fossem válidas em Java, esse erro de digitação seria detectado pelo compilador como um erro de tipo.

Como a detecção de erros é reduzida quando são permitidas expressões de modo misto, F# e ML não as permitem. Por exemplo, essas linguagens não permitem a mistura de operandos inteiros e de ponto flutuante em expressões.

Na maioria das outras linguagens comuns, não existem restrições em expressões aritméticas de modo misto.

As linguagens baseadas em C têm tipos inteiros menores que **`int`**. Em Java, são os tipos **`byte`** e **`short`**. Operandos de todos esses tipos sofrem coerção para **`int`** sempre que praticamente qualquer operador é aplicado a eles. Então, embora os dados possam ser armazenados em variáveis desses tipos, eles não podem ser manipulados antes da conversão em um tipo maior. Por exemplo, considere o seguinte código Java:

```
byte a, b, c;
. . .
a = b + c;
```

Os valores de `b` e `c` sofrem coerção para **`int`** e uma adição de inteiros é realizada. Então, a soma é convertida no tipo **byte** e colocada em `a`. Dado o maior tamanho das memórias dos computadores atuais, existe pouco incentivo para usar **`byte`** e **`short`**, a menos que um grande número deles deva ser armazenado.

7.4.2 Conversão de tipo explícita

A maioria das linguagens fornece alguma capacidade para a realização de conversões explícitas, tanto de alargamento quanto de estreitamento. Em alguns casos, mensagens de aviso são produzidas quando uma conversão de estreitamento explícita resulta em uma mudança significativa no valor do objeto convertido.

Nas linguagens baseadas em C, as conversões de tipo explícitas são chamadas de ***casts***. Para especificar um *cast*, o tipo desejado é colocado entre parênteses imediatamente antes da expressão a ser convertida, como em

```
(int) angle
```

Uma das razões para os parênteses em torno do nome do tipo nessas conversões é que a primeira dessas linguagens, C, tinha diversos nomes de tipos de duas palavras, como **`long int`**.

» *nota histórica*

Como um exemplo mais extremo dos perigos e custos de muitas coerções, considere os esforços de PL/I para ter flexibilidade em expressões. Em PL/I, uma variável do tipo cadeia de caracteres pode ser o operando de um operador aritmético, com um inteiro como o outro operando. Em tempo de execução, a cadeia é varrida em busca de um valor numérico. Se o valor contiver um ponto decimal, assume-se que o valor é um tipo de ponto flutuante, que o outro operando sofre uma coerção para ponto flutuante e que a operação resultante é de ponto flutuante. Essa política de coerção é muito dispendiosa, porque tanto a verificação de tipos quanto a conversão devem ser feitas em tempo de execução. Ela também elimina a possibilidade de detectar erros de programação em expressões, já que um operador binário pode combinar um operando de qualquer tipo com um operando de praticamente qualquer outro tipo.

Em ML e F#, os *casts* têm a sintaxe de chamadas a funções. Por exemplo, em F# teríamos:

```
float(sum)
```

7.4.3 Erros em expressões

Diversos erros podem ocorrer durante a avaliação de expressões. Se a linguagem requer verificação de tipos, seja ela estática ou dinâmica, não podem ocorrer erros de tipo dos operandos. Os possíveis erros devidos a coerções de operandos em expressões já foram discutidos. Os outros tipos de erros ocorrem em função das limitações da aritmética computacional e das limitações inerentes da aritmética. Os erros mais comuns ocorrem quando o resultado de uma operação não pode ser representado na célula de memória para a qual ele foi alocado. Isso é chamado de **transbordamento** (*overflow*) ou de **transbordamento negativo** (*underflow*), dependendo se o resultado foi muito grande ou muito pequeno. Uma limitação da aritmética é que a divisão por zero não é permitida. Evidentemente, o fato de não ser matematicamente viável não impede um programa de tentar efetuar tal operação.

O transbordamento, o transbordamento negativo de ponto flutuante e a divisão por zero são exemplos de erros em tempo de execução, algumas vezes chamados de **exceções**. Recursos de linguagens que permitem aos programas detectar e tratar exceções são discutidos no Capítulo 14.

7.5 EXPRESSÕES RELACIONAIS E BOOLEANAS

Além das expressões aritméticas, as linguagens de programação oferecem suporte para expressões relacionais e booleanas.

7.5.1 Expressões relacionais

Um **operador relacional** é aquele que compara os valores de seus dois operandos. Uma expressão relacional tem dois operandos e um operador relacional. O valor de uma expressão relacional é booleano, exceto quando valores booleanos não têm um tipo específico incluído na linguagem. Os operadores relacionais são comumente sobrecarregados para uma variedade de tipos. A operação que determina a verdade ou a falsidade de uma expressão relacional depende dos tipos dos operandos. Ela pode ser simples, como para operandos inteiros, ou complexa, como para operandos do tipo cadeia de caracteres. Os tipos dos operandos que podem ser usados para operadores relacionais são tipos numéricos, cadeias e tipos de enumeração.

A sintaxe dos operadores relacionais para igualdade e desigualdade difere entre algumas das linguagens de programação. Por exemplo, para desigualdade, as linguagens baseadas em C usam `!=`, Lua usa `~=`, Fortran 95+ usa `.NE.` ou `<>` e ML e F# usam `<>`.

JavaScript e PHP têm dois operadores relacionais adicionais, `===` e `!==`. Eles são semelhantes aos seus equivalentes, `==` e `!=`, mas impedem a coerção de seus operandos. Por exemplo, a expressão

```
"7" == 7
```

é verdadeira em JavaScript porque, quando uma cadeia e um número são os operandos de um operador relacional, a cadeia sofre uma coerção para um número. Entretanto,

```
"7" === 7
```

é falsa, porque nenhuma coerção é feita nos operandos desse operador.

» *nota histórica*

Os projetistas de Fortran I usaram abreviações em inglês para os operadores relacionais porque os símbolos > e < não estavam nas perfuradoras de cartões na época do projeto da linguagem (meados dos anos 50).

Ruby usa == para o operador de igualdade relacional que usa coerções e `eql?` para igualdade sem coerções. Ruby usa === apenas na cláusula **`when`** de sua sentença **`case`**, conforme discutido no Capítulo 8.

Os operadores relacionais sempre têm precedência mais baixa que os operadores aritméticos, de forma que, em expressões como

```
a + 1 > 2 * b
```

as expressões aritméticas são avaliadas primeiro.

7.5.2 Expressões booleanas

Expressões booleanas consistem em variáveis booleanas, constantes booleanas, expressões relacionais e operadores booleanos. Os operadores normalmente incluem aqueles para as operações E, OU e NÃO (AND, OR e NOT) e, algumas vezes, para OU exclusivo (XOR) e equivalência. Operadores booleanos normalmente recebem apenas operandos booleanos (variáveis booleanas, literais booleanos ou expressões relacionais) e produzem valores booleanos.

Na matemática de álgebras booleanas, os operadores OU e E devem ter precedência igual. Entretanto, as linguagens baseadas em C atribuem uma precedência maior para o E do que para o OU. Talvez isso tenha derivado da correlação, sem fundamento, da multiplicação com o E e da adição com o OU, o que naturalmente resultaria em uma atribuição de precedência maior para o E.

Como expressões aritméticas podem ser os operandos de expressões relacionais e expressões relacionais podem ser os operandos de expressões booleanas, as três categorias de operadores devem ser colocadas em diferentes níveis de precedência, relativas umas às outras.

A precedência dos operadores aritméticos, relacionais e booleanos nas linguagens baseadas em C é a seguinte:

Mais alta	++ e − posfixados
	+ unário, − unário, ++, −− e ! prefixados
	*, /, %
	+ binário, − binário
	<, >, <=, >=
	=, !=
	&&
Mais baixa	||

Versões de C anteriores à C99 são exceção entre as linguagens imperativas mais populares, visto que não têm um tipo booleano e, dessa forma, não têm valores booleanos. Em lugar deles, valores numéricos são usados para representar valores booleanos. Em vez de operandos booleanos, são usadas variáveis escalares (numéricas ou caracteres) e constantes, com o zero considerado falso e todos os valores diferentes de zero, verdadeiros. O resultado da avaliação de tal expressão é um inteiro, com o valor 0 se for falsa e 1 se for verdadeira. A aritmética de expressões também pode ser usada para expressões booleanas em C99 e C++.

Um resultado incomum do projeto de C para expressões relacionais é que a seguinte expressão é válida:

```
a > b > c
```

O operador relacional mais à esquerda é avaliado primeiro porque os operadores relacionais em C são associativos à esquerda, produzindo 0 ou 1. Então, esse resultado é comparado com a variável `c`. Nunca existe uma comparação entre `b` e `c` nessa expressão.

Algumas linguagens, incluindo Perl e Ruby, fornecem dois conjuntos de operadores lógicos binários, `&&` e **`and`** para o E e `||` e **`or`** para o OU. Uma diferença entre `&&` e **`and`** (e entre `||` e **`or`**) é que as versões por extenso têm precedência menor. Além disso, **`and`** e **`or`** têm precedência igual, mas `&&` tem precedência maior que `||`.

Quando os operadores não aritméticos das linguagens baseadas em C são incluídos, existem mais de 40 operadores e ao menos 14 níveis diferentes de precedência. Essa é uma evidência clara da diversidade das coleções de operadores e da complexidade de expressões possíveis nessas linguagens.

A legibilidade determina que uma linguagem deve incluir um tipo booleano, conforme foi mencionado no Capítulo 6, em vez de simplesmente usar tipos numéricos em expressões booleanas. Alguma detecção de erros é perdida no uso de tipos numéricos para operandos booleanos, porque qualquer expressão numérica, seja pretendida ou não, é um operando válido para um operador booleano. Nas outras linguagens imperativas, qualquer expressão não booleana usada como operando de um operador booleano é detectada como um erro.

7.6 AVALIAÇÃO EM CURTO-CIRCUITO

Uma **avaliação em curto-circuito** de uma expressão é uma avaliação na qual o resultado é determinado sem se avaliar todos os operandos e/ou operadores. Por exemplo, o valor da expressão aritmética

```
(13 * a) * (b / 13 - 1)
```

é independente do valor de `(b / 13 - 1)` se `a` for igual a `0`, pois `0 * x = 0` para qualquer `x`. Então, quando `a` é `0`, não há a necessidade de avaliar `(b / 13 - 1)` ou de efetuar a segunda multiplicação. Entretanto, em expressões aritméticas, esse atalho não é facilmente detectado durante a execução; então, ele nunca é utilizado.

O valor da expressão booleana

```
(a >= 0) && (b < 10)
```

é independente da segunda expressão relacional se `a < 0`, porque a expressão `(FALSE && (b < 10))` é falsa para todos os valores de `b`. Então, quando `a` é menor que `0`, não há a necessidade de avaliar `b`, a constante `10`, a segunda expressão relacional ou a operação `&&`. Diferentemente do caso das expressões aritméticas, esse atalho pode ser facilmente descoberto durante a execução.

Para ilustrar um problema em potencial na avaliação que não é em curto-circuito de expressões booleanas, suponha que Java não usasse avaliação em curto-circuito. Um laço de repetição para busca em uma tabela poderia ser escrito com a sentença **`while`**. Uma versão simples de código Java para tal busca, supondo que `list`, que tem `listlen` elementos, é a matriz a ser buscada e `key` é o valor a ser buscado, é

```
index = 0;
while ((index < listlen) && (list[index] != key))
  index = index + 1;
```

Se a avaliação não for em curto-circuito, ambas as expressões relacionais na expressão booleana da sentença **`while`** serão avaliadas, independentemente do valor da primeira. Então, se `key` não estiver em `list`, o programa terminará com uma exceção de índice fora de faixa. A mesma iteração que tem `index == listlen` referenciará `list[listlen]`, o que causa o erro de indexação, porque `list` é declarada com o valor `listlen-1` como limite superior de índice.

Se uma linguagem fornece avaliação em curto-circuito de expressões booleanas e ela é usada, isso não é um problema. No exemplo anterior, um esquema de avaliação em curto-circuito avaliaria o primeiro operando e o operador E, mas pularia o segundo operando se o primeiro fosse falso.

Uma linguagem que fornece avaliação em curto-circuito de expressões booleanas e que também tem efeitos colaterais em expressões permite que erros sutis ocorram. Suponha que a avaliação em curto-circuito seja usada em uma expressão e que parte da expressão que contém um efeito colateral não seja avaliada; então, o efeito colateral ocorrerá apenas nas avaliações completas da expressão como um todo. Se a corretude do programa depende do efeito colateral, a avaliação em curto-circuito pode resultar em um erro sério. Por exemplo, considere a expressão Java

```
(a > b) || ((b++) / 3)
```

Nessa expressão, `b` é modificado (na segunda expressão aritmética) apenas quando `a <= b`. Se o programador supusesse que `b` seria modificado toda vez que essa expressão fosse avaliada durante a execução (e a corretude do programa dependesse disso), o programa falharia.

Nas linguagens baseadas em C, os operadores E e OU usuais, `&&` e `||`, respectivamente, são avaliados em curto-circuito. Entretanto, essas linguagens também têm operadores E e OU bit a bit, `&` e `|`, respectivamente, que podem ser usados em operandos com valores booleanos e que não são avaliados em curto-circuito. É claro que os operadores bit a bit só serão equivalentes aos operadores booleanos usuais se todos os operandos puderem ser apenas `0` (para falso) ou `1` (para verdadeiro).

Todos os operadores lógicos de Ruby, Perl, ML, F# e Python são avaliados em curto-circuito.

7.7 SENTENÇAS DE ATRIBUIÇÃO

Conforme já mencionamos, a sentença de atribuição é uma das construções centrais nas linguagens imperativas. Ela fornece o mecanismo por meio do qual o usuário pode mudar dinamicamente as vinculações de valores a variáveis. Na seção seguinte é discutida a forma mais simples de atribuição. As seções subsequentes descrevem uma variedade de alternativas.

7.7.1 Atribuições simples

Praticamente todas as linguagens de programação usadas atualmente utilizam o sinal de igualdade para o operador de atribuição. Todas elas devem usar algo diferente de um sinal de igualdade para o operador de igualdade relacional, para evitar confusão com seu operador de atribuição.

ALGOL 60 foi pioneira no uso de `:=` como operador de atribuição, o que evita a confusão da atribuição com a igualdade. Ada também usa esse operador de atribuição.

As escolhas de projeto relativas a como as atribuições são usadas em uma linguagem variam bastante. Em algumas linguagens, como Fortran e Ada, uma atribuição pode aparecer apenas como uma sentença autônoma, e o destino é restringido a uma única variável. Existem, entretanto, muitas alternativas.

7.7.2 Alvos condicionais

Perl permite alvos condicionais em sentenças de atribuição. Por exemplo, considere

```
($flag ? $count1 : $count2) = 0;
```

que é equivalente a

```
if ($flag) {
  $count1 = 0;
} else {
```

```
  $count2 = 0;
}
```

7.7.3 Operadores de atribuição compostos

Um **operador de atribuição composto** é um método de atalho usado para especificar uma forma de atribuição comumente necessária. A forma de atribuição que pode ser abreviada com essa técnica tem a variável de destino aparecendo também como o primeiro operando na expressão no lado direito, como em

```
a = a + b
```

Operadores de atribuição compostos foram introduzidos por ALGOL 68, posteriormente foram adotados de uma forma um pouco diferente por C e fazem parte de outras linguagens baseadas em C, como Perl, JavaScript, Python e Ruby. A sintaxe desses operadores de atribuição é a concatenação do operador binário com o operador =. Por exemplo,

```
sum += value;
```

é equivalente a

```
sum = sum + value;
```

As linguagens que oferecem suporte a operadores de atribuição compostos têm versões para a maioria de seus operadores binários.

7.7.4 Operadores de atribuição unários

As linguagens baseadas em C, Perl e JavaScript incluem dois operadores aritméticos unários especiais que são atribuições abreviadas. Eles combinam operações de incremento e decremento com atribuição. Os operadores ++, para incremento, e −−, para decremento, podem ser usados tanto em expressões quanto para formar sentenças de atribuição autônomas de um único operador. Eles podem aparecer como operadores pré-fixados, que precedem os operandos, ou como operadores pós-fixados, que vêm após os operandos. Na sentença de atribuição

```
sum = ++ count;
```

o valor de `count` é decrementado em 1 unidade e então atribuído a `sum`. Essa operação também poderia ser declarada como

```
count = count + 1;
sum = count;
```

Se o mesmo operador for usado como um operador pós-fixado, como em

```
sum = count ++;
```

a atribuição do valor de `count` a `sum` ocorrerá primeiro; então, `count` será incrementado. O efeito é o mesmo das duas sentenças a seguir

```
sum = count;
count = count + 1;
```

Um exemplo do uso do operador unário de incremento para formar uma sentença de atribuição completa é

```
count ++;
```

que simplesmente incrementa `count`. Ele não se parece com uma atribuição, mas certamente é. Tal código é equivalente à sentença

```
count = count + 1;
```

Quando dois operadores unários são aplicados ao mesmo operando, a associação é da direita para a esquerda. Por exemplo, em

```
- count ++
count
```

`count` é primeiro incrementado e então negado. Então, ele é equivalente a

```
- (count ++)
```

em vez de a

```
(- count) ++
```

7.7.5 Atribuição como uma expressão

Nas linguagens baseadas em C, Perl e JavaScript, a sentença de atribuição produz um resultado, o qual é o mesmo que o valor atribuído ao alvo. Ela pode, dessa forma, ser usada como uma expressão e como um operando em outras expressões. Esse projeto trata o operador de atribuição de forma bastante similar a qualquer outro operador binário, exceto por ter o efeito colateral de modificar seu operando da esquerda. Por exemplo, em C, é comum escrever sentenças como

```
while ((ch = getchar()) != EOF) { ... }
```

Nessa sentença, o próximo caractere do arquivo da entrada padrão, normalmente o teclado, é obtido com `getchar` e atribuído à variável `ch`. O resultado, ou valor atribuído, é então comparado com a constante `EOF`. Se `ch` não for igual a `EOF`, a sentença composta `{...}` será executada. Note que a atribuição precisa estar entre parênteses – nas linguagens que oferecem suporte à atribuição como uma expressão, a precedência do operador de atribuição é menor que a dos operadores relacionais. Sem os parênteses, o novo caractere seria comparado primeiro com `EOF`. Então, o resultado dessa comparação, seja `0` ou `1`, seria atribuído a `ch`.

A desvantagem de permitir sentenças de atribuição como operandos em expressões é que isso causa ainda outro tipo de efeito colateral nas expressões. Esse tipo de efeito colateral pode levar a expressões difíceis de serem lidas e entendidas. Uma expressão com qualquer tipo de efeito colateral tem essa desvantagem. Ela não pode ser lida como

uma expressão, que na matemática é a denotação de um valor, mas apenas como uma lista de instruções com uma ordem de execução estranha. Por exemplo, a expressão

```
a = b + (c = d / b) - 1
```

denota as instruções

```
Atribuir d / b para c
Atribuir b + c para temp
Atribuir temp - 1 para a
```

Note que o tratamento de um operador de atribuição como qualquer outro operador binário permite o efeito de atribuições de múltiplos alvos, como

```
sum = count = 0;
```

no qual `count` primeiro recebe o valor zero e, então, o valor de `count` é atribuído a `sum`. Essa forma de atribuição com múltiplos alvos também é válida em Python.

Existe uma perda da detecção de erros no projeto do operador de atribuição em C que com frequência leva a erros de programas. Em particular, se digitarmos

```
if (x = y) ...
```

em vez de

```
if (x == y) ...
```

um equívoco que se comete facilmente, isso não será detectado como um erro pelo compilador. Em vez de testar uma expressão relacional, o valor atribuído a `x` é testado (nesse caso, é o valor de `y` que chega a essa sentença). Isso é resultado de duas decisões de projeto: permitir que atribuições se comportem como um operador binário comum e usar dois operadores bastante semelhantes, = e ==, com significados completamente diferentes. Esse é outro exemplo das deficiências de segurança dos programas C e C++. Note que Java e C# permitem apenas expressões **booleanas** em suas sentenças **`if`**, o que impede que esse problema ocorra*.

7.7.6 Atribuições múltiplas

Diversas linguagens de programação recentes, incluindo Perl, Ruby e Lua, fornecem sentenças de atribuição de múltiplos alvos e fontes. Por exemplo, em Perl é possível escrever

```
($first, $second, $third) = (20, 40, 60);
```

Isso significa que `20` é atribuído para `$first`, `40` é atribuído para `$second` e `60` é atribuído para `$third`. Se os valores das duas variáveis precisam ser trocados, isso pode ser feito com uma atribuição, como em

```
($first, $second) = ($second, $first);
```

*N. de R. T.: Na verdade, esse problema poderia ocorrer em Java caso as variáveis `x` e `y` do exemplo fossem do tipo primitivo booleano ou do tipo wrapper Boolean.

que troca os valores de `$first` e `$second` corretamente, sem o uso de uma variável temporária (ao menos sem o uso de uma gerenciada pelo programador).

A sintaxe da forma mais simples de atribuição múltipla em Ruby é semelhante à de Perl, exceto pelo fato de que os lados esquerdo e o direito não ficam entre parênteses. Além disso, Ruby inclui algumas outras versões mais elaboradas de atribuições múltiplas, que não são discutidas aqui.

> **» *nota histórica***
>
> O computador PDP-11, no qual a linguagem C foi implementada pela primeira vez, tem modos de endereçamento de autoincremento e autodecremento, que são versões em hardware dos operadores de incremento e decremento de C quando usados como índices de matrizes. Alguém poderia supor, a partir disso, que o projeto desses operadores em C foi baseado na arquitetura do PDP-11. Entretanto, essa suposição estaria errada, porque os operadores de C foram herdados da linguagem B, projetada antes do primeiro PDP-11.

7.7.7 Atribuições em linguagens de programação funcionais

Todos os identificadores usados em linguagens funcionais puras e alguns utilizados em outras linguagens funcionais são apenas nomes de valores. Como tal, seus valores nunca mudam. Por exemplo, em ML, nomes são vinculados a valores com a declaração **`val`**, cuja forma está exemplificada a seguir:

```
val cost = quantity * price;
```

Se `cost` aparece no lado esquerdo de uma declaração **`val`** subsequente, essa declaração cria uma nova versão do nome `cost`, a qual não tem nenhuma relação com a versão anterior, que, então, está oculta.

F# tem uma declaração muito parecida, que usa a palavra reservada **`let`**. A diferença entre **`let`** de F# e **`val`** de ML é que **`let`** cria um novo escopo, enquanto **`val`** não. Na verdade, em ML, declarações **`val`** são frequentemente aninhadas em construções **`let`** . **`let`** e **`val`** são discutidas mais detalhadamente no Capítulo 15.

7.8 ATRIBUIÇÃO DE MODO MISTO

Expressões de modo misto foram discutidas na Seção 7.4.1. Com frequência, as sentenças de atribuição também são de modo misto. A questão de projeto é: o tipo da expressão precisa ter o mesmo tipo da variável que recebe a atribuição, ou a coerção pode ser usada em alguns casos em que os tipos não casam?

C, C++ e Perl usam regras de coerção para atribuição de modo misto similares àquelas que tais linguagens usam para expressões de modo misto; ou seja, muitas das misturas possíveis são válidas, com coerções aplicadas livremente.[7]

Num claro distanciamento de C++, Java e C# permitem a atribuição de modo misto apenas se a coerção exigida é de alargamento.[8] Então, um valor **`int`** pode ser atribuído a

[7]Note que, em Python e Ruby, os tipos são associados aos objetos, não às variáveis; portanto, não existe algo como atribuições de modo misto nessas linguagens.

[8]Não é exatamente verdade: se um literal inteiro – para o qual o compilador atribui o tipo **`int`** por padrão – é atribuído para uma variável **`char`**, **`byte`** ou **`short`** e o literal está na faixa do tipo da variável, o valor **`int`** sofre coerção para o tipo da variável em uma conversão de estreitamento. Essa conversão de estreitamento não pode resultar em um erro.

uma variável **`float`**, mas não o contrário. Não permitir metade das possíveis atribuições de modo misto é uma maneira simples, mas efetiva, de aumentar a confiabilidade de Java e C# em relação à C e C++.

Evidentemente, nas linguagens funcionais, em que as atribuições são usadas apenas para dar nomes a valores, não existe nenhuma atribuição de modo misto.

RESUMO

Expressões são compostas de constantes, variáveis, parênteses, chamadas a funções e operadores. Sentenças de atribuição incluem variáveis-alvo, operadores de atribuição e expressões.

A semântica de uma expressão é determinada, em grande parte, pela ordem de avaliação dos operadores. As regras de associatividade e de precedência para operadores em expressões de uma linguagem determinam a ordem de avaliação dos operadores nessas expressões. A ordem de avaliação dos operandos é importante se efeitos colaterais funcionais são possíveis. Conversões de tipo podem ser de alargamento ou de estreitamento. Algumas conversões de estreitamento produzem valores errôneos. Conversões de tipo implícitas, ou coerções, em expressões são comuns. Apesar disso, eliminam alguns dos benefícios da verificação de tipos, diminuindo a confiabilidade.

Sentenças de atribuição têm aparecido em uma ampla variedade de formas, incluindo alvos condicionais, operadores de atribuição e atribuições de listas.

QUESTÕES DE REVISÃO

1. Defina *precedência de operador* e *associatividade de operador*.
2. O que é um operador ternário?
3. O que é um operador pré-fixado?
4. Que operador normalmente tem associatividade à direita?
5. O que é um operador não associativo?
6. Que regras de associatividade são usadas por APL?
7. Qual é a diferença entre a maneira pela qual os operadores são implementados em C++ e Ruby?
8. Defina *efeito colateral funcional*.
9. O que é uma coerção?
10. O que é uma expressão condicional?
11. O que é um operador sobrecarregado?
12. Defina *conversões de alargamento* e *de estreitamento*.
13. Em JavaScript, qual é a diferença entre == e ===?

14. O que é uma expressão de modo misto?
15. O que é transparência referencial?
16. Quais são as vantagens da transparência referencial?
17. Como a ordem de avaliação dos operandos interage com os efeitos colaterais funcionais?
18. O que é a avaliação em curto-circuito?
19. Cite uma linguagem que sempre faz avaliação em curto-circuito de expressões booleanas. Cite uma que nunca faz isso.
20. Como C oferece suporte para expressões relacionais e booleanas?
21. Qual é o propósito de um operador de atribuição composto?
22. Qual é a associatividade dos operadores aritméticos unários de C?
23. Qual é uma possível desvantagem de tratar o operador de atribuição como se ele fosse um operador aritmético?
24. Que duas linguagens incluem atribuições múltiplas?
25. Que atribuições de modo misto são permitidas em Java?
26. Que atribuições de modo misto são permitidas em ML?
27. O que é um *cast*?

PROBLEMAS

1. Quando você poderia desejar que o compilador ignorasse diferenças de tipos em uma expressão?
2. Argumente a favor e contra permitir expressões aritméticas de modo misto.
3. Você acha que a eliminação de operadores sobrecarregados em sua linguagem favorita seria benéfica? Justifique sua resposta.
4. Seria uma boa ideia eliminar todas as regras de precedência de operadores e exigir parênteses para mostrar a precedência desejada em expressões? Justifique sua resposta.
5. As operações de atribuição de C (por exemplo, +=) deveriam ser incluídas em outras linguagens (que ainda não as têm)? Justifique sua resposta.
6. As formas de atribuição de um único operando de C (por exemplo, `++count`) deveriam ser incluídas em outras linguagens (que ainda não as têm)? Justifique sua resposta.
7. Descreva uma situação na qual o operador de adição em uma linguagem de programação não seria comutativo.

8. Descreva uma situação na qual o operador de adição em uma linguagem de programação não seria associativo.

9. Suponha as seguintes regras de associatividade e de precedência para expressões:

Precedência	*Mais alta*	`*`, `/`, **not**
		`+`, `-`, `&`, **mod**
		- (unário)
		`=`, `/=`, `<`, `<=`, `>=`, `>`
		and
	Mais baixa	**or**, **xor**
Associatividade	*Esquerda para a direita*	

Mostre a ordem de avaliação das seguintes expressões por meio do uso de parênteses em todas as subexpressões e da colocação de um expoente no parêntese direito para indicar a ordem. Por exemplo, para a expressão

```
a + b * c + d
```

a ordem de avaliação seria representada como

```
((a + (b * c)¹)² + d)³
```

a. `a * b - 1 + c`
b. `a * (b - 1) / c` **mod** `d`
c. `(a - b) / c & (d * e / a - 3)`
d. `-a` **or** `c = d` **and** `e`
e. `a > b` **xor** `c` **or** `d <= 17`
f. `-a + b`

10. Mostre a ordem de avaliação das expressões do Problema 9, supondo que não existem regras de precedência e que todos os operadores são associativos da direita para a esquerda.

11. Escreva uma descrição em BNF para as regras de precedência e de associatividade definidas para as expressões no Problema 9. Suponha que os únicos operandos são os nomes `a`, `b`, `c`, `d` e `e`.

12. Usando a gramática do Problema 11, desenhe árvores de análise sintática para as expressões do Problema 9.

13. Seja a função `fun` definida como

```
int fun(int*k) {
 *k += 4;
 return 3 * (*k) - 1;
 }
```

Suponha que `fun` seja usada em um programa, como:

```
void main() {
  int i = 10, j = 10, sum1, sum2;
  sum1 = (i / 2) + fun(&i);
  sum2 = fun(&j) + (j / 2);
}
```

Quais são os valores de `sum1` e `sum2`

a. se os operandos na expressão forem avaliados da esquerda para a direita?
b. se os operandos na expressão forem avaliados da direita para a esquerda?

14. Qual seu principal argumento contra as (ou a favor das) regras de precedência de operadores de APL?
15. Explique por que é difícil eliminar efeitos colaterais funcionais em C.
16. Para alguma linguagem de sua escolha, crie uma lista de símbolos de operadores que poderiam ser usados para eliminar todas as sobrecargas de operadores.
17. Determine se as conversões de tipo explícitas de estreitamento em duas linguagens que você conhece fornecem mensagens de erro quando um valor convertido perde sua utilidade.
18. Deveria ser permitido a um compilador com otimização para C e C++ modificar a ordem das subexpressões em uma expressão booleana? Justifique sua resposta.
19. Considere o seguinte programa em C:

```
int fun(int *i) {
  *i += 5;
  return 4;
 }
void main() {
  int x = 3;
  x = x + fun(&x);
  }
```

Qual é o valor de `x` após a sentença de atribuição em `main`, supondo que

a. os operandos são avaliados da esquerda para a direita.

b. os operandos são avaliados da direita para a esquerda.

20. Por que Java especifica que operandos em expressões são todos avaliados da esquerda para a direita?
21. Explique como as regras de coerção de uma linguagem afetam sua detecção de erros.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Execute o código dado no Problema 13 (no Conjunto de problemas) em algum sistema que suporte C para determinar os valores de `sum1` e `sum2`. Explique os resultados.
2. Reescreva o programa do exercício anterior em C++, Java e C#, execute-os e compare os resultados.
3. Escreva um programa de teste, em sua linguagem favorita, que determine e mostre a precedência e a associatividade de seus operadores aritméticos e booleanos.
4. Escreva um programa em Java que exponha a regra da linguagem para a ordem de avaliação de operandos quando um deles é uma chamada a método.
5. Repita o Exercício de programação 4 com C++.
6. Repita o Exercício de programação 4 com C#.
7. Escreva um programa em C++, Java ou C# que ilustre a ordem de avaliação de expressões usadas como parâmetros reais de um método.
8. Escreva um programa em C que tenha as seguintes sentenças:

```
int a, b;
a = 10;
b = a + fun();
printf("With the function call on the right, ");
printf(" b is: %d\n", b);
a = 10;
b = fun() + a;
printf("With the function call on the left, ");
printf(" b is: %d\n", b);
```

   e defina `fun` para somar 10 a `a`. Explique os resultados.

9. Escreva um programa em Java, C++ ou C# que efetue um grande número de operações de ponto flutuante e um número igual de operações em inteiros. Compare o tempo exigido.

8

Estruturas de controle no nível de sentença

8.1 Introdução

8.2 Sentenças de seleção

8.3 Sentenças de iteração

8.4 Desvio incondicional

8.5 Comandos protegidos

8.6 Conclusões

O fluxo de controle, ou sequência de execução, em um programa pode ser examinado em diversos níveis. O Capítulo 7 discutiu o fluxo de controle dentro de expressões, governado pelas regras de associatividade e de precedência. No nível mais alto está o fluxo de controle entre unidades de programas, discutido nos Capítulos 9 e 13. Entre esses dois extremos está a importante questão do fluxo de controle entre sentenças, assunto deste capítulo.

Começamos com um panorama da evolução das sentenças de controle. Esse tópico é seguido de um exame cuidadoso das sentenças de seleção, tanto as de dois caminhos quanto as de seleção múltipla. Em seguida, discutimos a variedade de sentenças para repetição que vêm sendo desenvolvidas e usadas em linguagens de programação. Depois, verificamos os problemas associados às sentenças de desvios incondicionais. Por fim, descrevemos as sentenças de controle de comandos protegidos.

8.1 INTRODUÇÃO

Computações em programas escritos em linguagens imperativas são realizadas por meio da avaliação de expressões e da atribuição dos valores resultantes a variáveis. Entretanto, existem poucos programas úteis compostos inteiramente por sentenças de atribuição. Pelo menos dois mecanismos linguísticos adicionais são necessários para tornar as computações nos programas flexíveis e eficientes: um para escolher entre caminhos alternativos de fluxo de controle (de execução de sentenças) e um para gerar a execução repetida de sentenças ou de sequências de sentenças. Sentenças que fornecem essas capacidades são chamadas de **sentenças de controle**.

Computações em linguagens de programação funcionais são realizadas pela avaliação de expressões e pela aplicação de funções a parâmetros dados. Além disso, o fluxo de execução entre as expressões e funções é controlado por outras expressões e funções, embora algumas sejam semelhantes às sentenças de controle das linguagens imperativas.

As sentenças de controle da primeira linguagem de programação bem-sucedida, Fortran, foram, na verdade, projetadas pelos arquitetos do IBM 704. Tudo era diretamente relacionado às instruções em linguagem de máquina, de forma que seus recursos eram mais o resultado do projeto de instruções que do projeto de linguagem. Na época, pouco se sabia acerca da dificuldade da programação, e se pensava que as sentenças de controle de Fortran, em meados dos anos 1950, eram completamente adequadas. Pelos padrões de hoje, entretanto, elas seriam consideradas muito deficientes.

Boa parte da pesquisa e da discussão foi dedicada às sentenças de controle nos 10 anos entre meados dos anos 1960 e meados dos anos 1970. Uma das principais conclusões desses esforços foi que, apesar de uma única seleção de controle (um goto selecionável) ser minimamente suficiente, uma linguagem projetada para *não* incluir um goto precisa apenas de um pequeno número de sentenças de controle diferentes. Na verdade, foi provado que todos os algoritmos que podem ser expressos por diagramas de fluxo podem ser codificados em uma linguagem de programação com apenas duas sentenças de controle: uma para escolher entre dois caminhos de fluxo de controle e uma para iterações logicamente controladas (Böhm e Jacopini, 1966). Um resultado importante disso é que a sentença de desvio incondicional é supérflua – potencialmente útil, mas não essencial. Esse fato, combinado com os problemas práticos do uso de desvios in-

condicionais, ou gotos, levou a um grande debate acerca do goto, conforme discutido na Seção 8.4.

Os programadores se preocupam menos com os resultados de pesquisas teóricas sobre sentenças de controle do que com a questão da facilidade de escrita e de legibilidade. Todas as linguagens bastante usadas incluem mais sentenças de controle do que as duas minimamente exigidas, porque a facilidade de escrita é melhorada com um número maior e uma variedade mais ampla de sentenças. Por exemplo, em vez de exigir o uso de uma sentença com um laço controlado logicamente para todos os laços, é mais fácil escrever programas quando uma sentença de laço controlado por contador pode ser usada para construir laços naturalmente controlados por um contador. O principal fator que restringe o número de sentenças de controle em uma linguagem é a legibilidade, porque a presença de diversas formas de sentenças exige que os leitores dos programas conheçam uma linguagem maior. Lembre-se de que poucas pessoas aprendem todas as sentenças de uma linguagem relativamente extensa; em vez disso, elas aprendem o subconjunto que escolhem usar, em geral um diferente daquele usado pelo programador que escreveu o programa que estão tentando ler. Por outro lado, poucas sentenças de controle podem exigir o uso de sentenças de nível mais baixo, como o goto, que também tornam os programas menos legíveis.

A questão acerca da melhor coleção de sentenças de controle para fornecer os recursos necessários e a facilidade de escrita desejada tem sido bastante debatida. É essencialmente uma questão de quanto uma linguagem deveria ser expandida para aumentar sua facilidade de escrita à custa de sua simplicidade, seu tamanho e sua legibilidade.

Uma **estrutura de controle** é uma sentença de controle e a coleção de sentenças cuja execução ela controla.

Existe apenas uma questão de projeto relevante para todas as sentenças de controle de seleção e de iteração: a estrutura de controle deve ter múltiplas entradas? Todas as sentenças de seleção e de iteração controlam a execução de segmentos de código, e a questão é se a execução desses segmentos de código sempre começa com a primeira sentença no segmento. Atualmente, acredita-se que múltiplas entradas acrescentam pouco à flexibilidade de uma sentença de controle se comparadas ao decréscimo na legibilidade causado pela complexidade aumentada. Note que múltiplas entradas são possíveis apenas em linguagens que incluem gotos e rótulos (*labels*) para sentenças.

Nesse ponto, o leitor deve estar se perguntando por que múltiplas saídas de estruturas de controle não são consideradas uma questão de projeto. O motivo é que todas as linguagens de programação permitem alguma forma de múltiplas saídas de estruturas de controle. Isso ocorre pela seguinte razão: se todas as saídas de uma estrutura de controle são restritas a transferir o controle para a primeira sentença após a estrutura – onde o controle fluiria se a estrutura não tivesse uma saída explícita –, não existe prejuízo à legibilidade e perigo algum. No entanto, se uma saída puder ter um alvo irrestrito e, dessa forma, resultar em uma transferência de controle para qualquer lugar na unidade de programa que contenha a estrutura de controle, o prejuízo para a legibilidade é o mesmo que para uma sentença goto em qualquer outro lugar do programa. Linguagens que têm uma sentença goto permitem que ela apareça em qualquer lugar, inclusive em uma estrutura de controle. Logo, a questão é a inclusão de um goto, e não se múltiplas saídas de estruturas de controle são permitidas.

8.2 SENTENÇAS DE SELEÇÃO

Uma **sentença de seleção** fornece os meios para escolher entre dois ou mais caminhos de execução em um programa. Tais sentenças são partes fundamentais e essenciais de todas as linguagens de programação, conforme provado por Böhm e Jacopini.

Sentenças de seleção se enquadram em duas categorias gerais: dois caminhos e n caminhos, ou seleção múltipla. Sentenças de seleção de dois caminhos são discutidas na Seção 8.2.1; sentenças de seleções múltiplas estão na Seção 8.2.2.

8.2.1 Sentenças de seleção de dois caminhos

Apesar de as sentenças de seleção de dois caminhos das linguagens imperativas contemporâneas serem bastante similares, existem algumas variações em seus projetos. A forma geral de um seletor de dois caminhos é:

if expressão_de_controle
  cláusula então
  cláusula senão

8.2.1.1 Questões de projeto

As questões de projeto para seletores de dois caminhos podem ser resumidas a:

- Qual é a forma e o tipo da expressão que controla a seleção?
- Como são especificadas as cláusulas então e senão?
- Como o significado dos seletores aninhados deve ser especificado?

8.2.1.2 A expressão de controle

Expressões de controle são especificadas entre parênteses se a palavra reservada **then** (ou algum outro marcador sintático) não for usada para introduzir a cláusula então. Nos casos em que a palavra reservada **then** é usada (ou um marcador alternativo), existe menos necessidade de parênteses, então eles são normalmente omitidos, como em Ruby.

Em C89, que não tinha um tipo de dados booleano, expressões aritméticas eram usadas como expressões de controle. Isso também pode ser feito em Python, C99 e C++. Entretanto, nessas linguagens, tanto expressões aritméticas quanto booleanas podem ser usadas. Em outras linguagens contemporâneas, apenas expressões booleanas podem ser usadas para expressões de controle.

8.2.1.3 Forma da cláusula

Em muitas linguagens, as cláusulas então e senão aparecem como sentenças simples ou como sentenças compostas. Uma variação disso é Perl, em que todas as cláusulas então e senão devem ser sentenças compostas, mesmo que tenham apenas uma sentença. Muitas linguagens usam chaves para formar sentenças compostas, que servem como corpos das cláusulas então e senão. Em Python e Ruby, as cláusulas então e senão são sequências de sentenças, em vez de sentenças compostas. Nessas linguagens, a sentença de seleção completa é terminada com uma palavra reservada.

Python usa endentação para especificar sentenças compostas. Por exemplo,

```
if x > y :
  x = y
  print "case 1"
```

Todas as sentenças endentadas igualmente são incluídas na sentença composta.[1] Note que, em vez do **then**, é usado um sinal de dois pontos para introduzir a cláusula então em Python.

As variações na forma da cláusula têm implicações para a especificação do significado dos seletores aninhados, conforme discutido na próxima subseção.

8.2.1.4 Aninhamento de seletores

Lembre-se de que o problema da ambiguidade sintática de uma gramática simples para uma sentença de seleção de dois caminhos foi discutido no Capítulo 3. Tal gramática ambígua era:

```
<if_stmt> → if <logic_expr> then <stmt>
          | if <logic_expr> then <stmt> else <stmt>
```

A questão é que, quando uma sentença de seleção é aninhada na cláusula então de uma sentença de seleção, não fica claro a qual `if` uma cláusula senão deve ser associada. O problema é refletido na semântica das sentenças de seleção. Considere o seguinte código parecido com Java:

```
if (sum == 0)
  if (count == 0)
    result = 0;
else
    result = 1;
```

Essa sentença pode ser interpretada de duas maneiras, dependendo se a cláusula senão casa com a primeira cláusula então ou com a segunda. Note que a endentação parece indicar que a cláusula senão casa com a primeira cláusula então. Entretanto, com as exceções de Python e F#, a endentação não tem efeito na semântica das linguagens contemporâneas e é ignorada pelos seus compiladores.

O X da questão nesse exemplo é que a cláusula senão segue duas cláusulas então sem uma cláusula senão intermediária, e não existe um indicador sintático para especificar uma associação da cláusula senão com uma das cláusulas então. Em Java, como em outras linguagens imperativas, a semântica estática da linguagem especifica que a cláusula senão sempre casa com a cláusula então anterior mais próxima que ainda não está casada. Uma regra de semântica estática, em vez de uma entidade sintática, é usada para prover a desambiguação. Portanto, no exemplo, a cláusula senão seria casada com a segunda cláusula então. A desvantagem de usar uma regra, em vez de alguma entidade sintática, é que, apesar de o programador querer dizer que a cláusula senão é a alternativa à primeira cláusula então e o compilador achar que a estrutura está sintaticamente

[1] A sentença seguinte à sentença composta deve ter a mesma endentação que o `if`.

correta, sua semântica é a oposta. Para forçar a semântica alternativa em Java, o **if** interno é colocado em uma sentença composta, como em

```
if (sum == 0) {
  if  (count == 0)
    result = 0;
}
else
    result = 1;
```

C, C++ e C# têm o mesmo problema de Java com o aninhamento de sentenças de seleção. Como Perl requer que todas as cláusulas então e senão sejam compostas, ela não enfrenta esse problema. Em Perl, o código anterior seria escrito como segue:

```
if (sum == 0) {
  if (count == 0) {
    result = 0;
  }
} else {
    result = 1;
}
```

Se a semântica alternativa fosse necessária, o código seria

```
if (sum == 0) {
  if (count == 0) {
    result = 0;
  }
  else {
    result = 1;
  }
}
```

Outra maneira de evitar a questão das sentenças de seleção aninhadas é usar uma forma alternativa de elaborar sentenças compostas. Considere a estrutura sintática da sentença **if** de Java. A cláusula então segue a expressão de controle, e a cláusula senão é introduzida pela palavra reservada **else**. Quando a cláusula então é uma única sentença e a cláusula senão está presente, apesar de não existir a necessidade de marcar o final, a palavra reservada **else**, na verdade, indica o final da cláusula então. Quando a cláusula então é uma sentença composta, ela é terminada com uma chave de fechamento. Entretanto, se a última cláusula em um **if**, seja ela um então ou um senão, não é composta, não existe uma entidade sintática para marcar o final da sentença de seleção como um todo. O uso de uma palavra especial para esse propósito resolve a questão da semântica de seletores aninhados e melhora a legibilidade da sentença. Esse é o projeto escolhido para a sentença de seleção em Ruby e Lua. Como exemplo, considere a seguinte sentença em Ruby:

```
if a > b then sum = sum + a
  acount = acount + 1
else sum = sum + b
```

```
  bcount = bcount + 1
end
```

O projeto dessa sentença é mais regular que aquele das sentenças de seleção das linguagens baseadas em C, porque a forma é a mesma, independentemente do número de sentenças nas cláusulas então e senão. (Isso também é verdade para Perl.) Lembre-se de que, em Ruby, as cláusulas então e senão consistem em sequências de sentenças, em vez de em sentenças compostas. A primeira interpretação do exemplo de seleção no início desta seção, no qual a cláusula else casava com o **if** aninhado, pode ser escrita em Ruby como:

```
if sum == 0  then
  if count == 0 then
    result = 0
  else
    result = 1
  end
end
```

Como a palavra reservada **end** fecha o **if** aninhado, é claro que a cláusula senão casa com a cláusula então mais interna.

A segunda interpretação da sentença de seleção no início desta seção, na qual a cláusula else casava com o **if** externo, pode ser escrita em Ruby como:

```
if sum == 0  then
  if count == 0 then
    result = 0
  end
else
    result = 1
 end
```

A seguinte sentença, escrita em Python, é semanticamente equivalente à última sentença em Ruby acima:

```
if sum == 0 :
  if count == 0 :
    result = 0
else:
  result = 1
```

Se a linha **else:** fosse endentada para começar na mesma coluna que o **if** aninhado, a cláusula senão casaria com o **if** interno.

ML não tem problema com seletores aninhados, pois não permite sentenças **if** sem a cláusula else.

8.2.1.5 Expressões seletoras

Nas linguagens funcionais ML, F# e Lisp, o seletor não é uma sentença; é uma expressão que resulta em um valor. Portanto, pode aparecer em qualquer lugar que qualquer outra expressão possa aparecer. Considere o seguinte exemplo de seletor, escrito em F#:

```
let y =
    if x > 0 then  x
    else 2 * x;;
```

Ele gera o nome `y` e o configura como `x` ou como `2 * x`, dependendo de `x` ser maior que zero.

Em F#, o tipo do valor retornado pela cláusula então de uma construção **`if`** deve ser o mesmo do valor retornado por sua cláusula senão. Se não houver nenhuma cláusula senão, a cláusula então não poderá retornar um valor de um tipo normal. Nesse caso, só poderá retornar um tipo **`unit`**, um tipo especial que significa nenhum valor. Um tipo **`unit`** é representado em código como `()`.

8.2.2 Sentenças de seleção múltipla

A sentença de **seleção múltipla** permite a seleção de uma entre qualquer número de sentenças ou de grupos de sentenças. Ela é, dessa forma, uma generalização de um seletor. Na verdade, seletores de dois caminhos podem ser construídos com um seletor múltiplo.

A necessidade de escolher entre mais de dois caminhos de controle em um programa é comum. Apesar de ser possível construir um seletor múltiplo a partir de seletores de dois caminhos e de gotos, as estruturas resultantes são deselegantes, não confiáveis e difíceis de serem lidas e escritas. Logo, a necessidade de uma estrutura especial é clara.

8.2.2.1 Questões de projeto

Algumas das questões de projeto para seletores múltiplos são similares àquelas para seletores de dois caminhos. Por exemplo, uma questão é relativa ao tipo de expressão na qual o seletor é baseado. Nesse caso, a variedade de opções é grande, em parte porque há muitas seleções possíveis. Um seletor de dois caminhos precisa de uma expressão com apenas dois valores possíveis. Outra questão é se sentenças individuais, sentenças compostas ou sequências de sentenças podem ser selecionadas. Em seguida, existe a questão de se apenas um segmento selecionável pode ser executado quando a sentença é executada. Essa não é uma questão para seletores de dois caminhos, porque eles sempre permitem que apenas uma das cláusulas esteja em um caminho de controle durante uma execução. Conforme veremos, a solução para essa questão é trocar confiabilidade por flexibilidade. Outro ponto é relativo à forma das especificações de valores de cada um dos casos da seleção. Por fim, existe a questão de qual deve ser o resultado quando a avaliação da expressão de seleção é um valor que não seleciona um dos segmentos. (Tal valor não estaria representado entre os segmentos selecionáveis.) A escolha aqui é entre não permitir que essa situação aconteça e fazer com que a sentença não efetue nada quando tal situação ocorre.

A seguir, temos um resumo dessas questões de projeto:

- Quais são a forma e o tipo da expressão que controla a seleção?
- Como são especificados os segmentos selecionáveis?
- O fluxo de execução por meio da estrutura pode incluir apenas um segmento selecionável?
- Como os valores de cada caso são especificados?

- Como valores da expressão de seleção que não estão representados devem ser manipulados, se é que o devem?

8.2.2.2 Exemplos de seletores múltiplos

A sentença de seleção múltipla da linguagem C, o **switch**, que também faz parte de C++, Java e JavaScript, é um projeto relativamente primitivo. Sua forma geral é

```
switch (expressão) {
  case expressão_constante1:sentença1;
  . . .
  case constanten: sentença_n;
  [default: sentença_n+1]
}
```

onde a expressão de controle e as expressões constantes são de algum tipo discreto. Isso inclui tipos inteiros, caracteres e tipos enumeração. As sentenças selecionáveis podem ser sequências de sentenças, sentenças compostas ou blocos. O segmento opcional **default** é usado para valores não representados da expressão de controle. Se o valor da expressão de controle não é representado e nenhum segmento padrão está presente, então a sentença não faz nada.

A sentença **switch** não fornece desvios implícitos no final de seus segmentos de código. Isso permite que o controle flua por mais de um segmento de código selecionável em uma única execução. Considere o seguinte exemplo:

```
switch (index) {
   case 1:
   case 3: odd += 1;
           sumodd += index;
   case 2:
   case 4: even += 1;
           sumeven += index;
   default: printf("Error in switch, index = %d\n", index);
}
```

Esse código imprime a mensagem de erro a cada execução. Da mesma forma, o código para as constantes 2 e 4 é executado cada vez que o código nas constantes 1 ou 3 é executado. Para separar esses segmentos logicamente, um desvio explícito deve ser incluído. A sentença **break**, um goto restrito, é normalmente usada para sair de sentenças **switch**. Ela transfere o controle para a primeira sentença após a sentença composta na qual aparece.

A seguinte sentença **switch** usa **break** para restringir cada execução a um único segmento selecionável:

```
switch (index) {
  case 1:
  case 3: odd += 1;
          sumodd += index;
          break;
  case 2:
  case 4: even += 1;
```

```
            sumeven += index;
            break;
  default: printf("Error in switch, index = %d\n", index);
}
```

Ocasionalmente, é conveniente permitir que o controle flua de um segmento de código para outro. No exemplo acima, os segmentos para os valores de caso 1 e 2 estão vazios, o que permite que o controle flua para os segmentos 3 e 4, respectivamente. Essa é a razão pela qual não existem desvios implícitos na sentença **switch**. O problema de confiabilidade desse projeto surge quando a ausência errônea de uma sentença **break** em um segmento permite que o controle flua incorretamente para o próximo segmento. Os projetistas do **switch** de C trocaram um decréscimo na confiabilidade por um aumento na flexibilidade. Estudos têm mostrado, entretanto, que a capacidade de fazer com que o controle flua de um segmento selecionável para outro raramente é usada. O **switch** de C é modelado de acordo com a sentença de seleção múltipla presente em ALGOL 68, que também não tem desvios implícitos a partir de segmentos selecionáveis.

A sentença switch de C praticamente não tem restrições a respeito do posicionamento das expressões case, tratadas como rótulos de sentenças normais. Essa permissividade pode resultar em estruturas altamente complexas dentro do corpo da sentença switch. O seguinte exemplo foi retirado de Harbison e Steele (2002).

```
switch (x)
  default:
  if (prime(x))
    case 2: case 3: case 5: case 7:
      process_prime(x);
  else
     case 4: case 6: case 8: case 9: case 10:
       process_composite(x);
```

A forma desse código pode parecer muito complexa, mas foi projetada para corrigir um problema real e funciona correta e eficientemente para resolvê-lo.[2]

O switch do Java evita esse tipo de complexidade ao proibir que expressões case apareçam em qualquer lugar, exceto no nível superior do corpo do switch.

A sentença switch de C# se difere das de seus predecessores baseados em C de duas maneiras. Primeiro, C# tem uma regra de semântica estática que proíbe a execução implícita de mais de um segmento. A regra é que cada segmento selecionável deve terminar com uma sentença de desvio incondicional explícita: seja um **break**, que transfere o controle para fora da sentença **switch**, ou um goto, que pode transferir o controle para um dos segmentos selecionáveis (ou praticamente para qualquer lugar). Por exemplo,

```
switch (value) {
   case -1:
      Negatives++;
```

[2]O problema é chamar `process_prime` quando `x` é primo e `process_composite` quando `x` não é primo. O projeto do corpo de switch resultou de uma tentativa de otimização baseada no conhecimento de que `x` era mais comumente encontrado na faixa de 1 a 10.

```
        break;
    case 0:
        Zeros++;
        goto case 1;
    case 1:
        Positives++;
    default:
        Console.WriteLine("Error in switch \n");
}
```

Note que `Console.WriteLine` é o método para mostrar cadeias em C#.

A outra maneira pela qual o **`switch`** de C# difere do de seus predecessores é que a expressão de controle e as sentenças case podem ser cadeias em C#.

O **`switch`** de PHP usa a sintaxe do **`switch`** de C, mas permite mais flexibilidade de tipos. Os valores de cada caso podem ser de qualquer um dos tipos escalares de PHP – cadeias, inteiros ou de dupla precisão. Como em C, se não existe um **`break`** no final do segmento selecionado, a execução continua no próximo segmento.

Ruby tem duas formas de construções de seleção múltipla; ambas são chamadas de *expressões case* e ambas produzem o valor da última expressão avaliada. A única versão das expressões case de Ruby descrita aqui é semanticamente similar a uma lista de sentenças if aninhadas:

```
case
when expressão_booleana then expressão
. . .
when expressão_booleana then expressão
[else expressão]
end
```

A semântica dessa expressão case é que as expressões booleanas são avaliadas uma de cada vez, de cima para baixo. O valor da expressão case é o da primeira expressão then cuja expressão booleana seja verdadeira. O else representa o valor true nessa sentença, e a cláusula else é opcional. Por exemplo,[3]

```
leap = case
       when year % 400 == 0 then true
       when year % 100 == 0 then false
       else year % 4 == 0
       end
```

Essa expressão case é avaliada como verdadeira se `year` for um ano bissexto.

A outra forma de expressão case de Ruby é semelhante ao switch de Java. Perl, Python e Lua não têm sentenças de seleção múltipla.

8.2.2.3 Implementação de estruturas de seleção múltipla

Uma sentença de seleção múltipla é essencialmente um desvio de n caminhos para segmentos de código, onde n é o número de segmentos selecionáveis. A implementação

[3] Esse exemplo é de Thomas et al. (2013).

de tal sentença deve ser feita com várias instruções de desvios condicionais. Considere novamente a forma geral da sentença switch de C, com breaks:

```
switch (expressão) {
 case expressão_constante1:sentença1;
  break;
 . . .
 case constanten: sentençan;
  break;
 [default: sentençan + 1]
}
```

Uma tradução simples dessa sentença é:

```
Código para avaliar a expressão em t
goto branches
rótulo1: código para sentença1
  goto out
. . .
rótulon: código para sentençan
  goto out
default: código para sentençan+1
  goto out
desvios: if t = expressão_constante1 goto rótulo1
   . . .
   if t = expressão_constanten goto rótulon
   goto default
out:
```

O código para os segmentos selecionáveis precede os desvios, de forma que os alvos destes são todos conhecidos quando os desvios são gerados. Uma alternativa para esses desvios condicionais codificados é colocar os valores de caso e os rótulos em uma tabela e usar uma busca linear com um laço para encontrar o rótulo correto. Isso requer menos espaço do que os condicionais codificados.

O uso de desvios condicionais ou de uma busca linear em uma tabela de casos e de rótulos é uma estratégia simples, mas ineficiente, aceitável quando o número de casos é pequeno, como menos de 10. Ela tem de fazer um número de testes equivalente, em média, a cerca de metade do número de casos para encontrar o correto. Para o caso padrão ser escolhido, todos os outros precisam ser testados. Em sentenças com 10 ou mais casos, a baixa eficiência dessa forma não é justificada por sua simplicidade.

Quando o número de casos é de 10 ou mais, o compilador pode construir uma tabela de dispersão dos rótulos dos segmentos, o que resultaria em tempos aproximadamente iguais (e curtos) para escolher qualquer um dos segmentos selecionáveis. Se a linguagem permite faixas de valores para expressões case, como em Ruby, o método de dispersão não é adequado. Para essas situações, é melhor uma tabela de busca binária de valores case e endereços de segmentos.

Se a faixa dos valores case é relativamente pequena e mais da metade da faixa completa de valores é representada, pode ser construída uma matriz cujos índices são os

valores case e cujos valores são os rótulos de segmentos. Elementos de matrizes cujos índices não estão entre os valores case representados são preenchidos com o rótulo do segmento padrão. Então, a descoberta do rótulo do segmento correto é feita por meio da indexação de matrizes, que é muito rápida.

É claro que escolher entre essas abordagens é uma carga adicional para o compilador. Em muitos compiladores, apenas dois métodos estão disponíveis. Como em outras situações, determinar e usar o método mais eficiente exige mais tempo de compilação.

8.2.2.4 Seleção múltipla com `if`

Em muitas situações, uma sentença **`switch`** ou **`case`** é inadequada para seleção múltipla (o **`case`** de Ruby é uma exceção). Por exemplo, quando as seleções devem ser feitas com base em uma expressão booleana, em vez de por meio de algum tipo ordinal, os seletores aninhados de dois caminhos podem ser usados para simular um seletor múltiplo. Para melhorar a legibilidade de seletores de dois caminhos profundamente aninhados, algumas linguagens, como Perl e Python, foram estendidas especificamente para esse uso. A extensão permite que algumas das palavras especiais sejam deixadas de fora. Em particular, sequências else-if são substituídas por uma única palavra especial, e a palavra especial de fechamento no **`if`** aninhado não é usada. O seletor aninhado é, então, chamado de **cláusula else-if**. Considere a seguinte sentença de seleção em Python (note que else-if é escrito como **`elif`** em Python):

```
if  count < 10 :
  bag1 = True
elif count < 100 :
  bag2 = True
elif count < 1000 :
  bag3 = True
```

que é equivalente a:

```
if count < 10 :
  bag1 = True
else :
  if count < 100 :
    bag2 = True
  else :
    if count < 1000 :
      bag3 = True
    else :
      bag4 = True
```

A versão else-if (a primeira) é a mais legível. Note que esse exemplo não é facilmente simulado com uma sentença **`switch`**, porque cada sentença selecionável é escolhida com base em uma expressão booleana. Logo, a sentença else-if não é uma forma redundante de **`switch`**. Na verdade, nenhum dos seletores múltiplos em linguagens contemporâneas é tão geral quanto a sentença if-then-else-if. Uma descrição em semântica operacional de uma sentença de seleção geral com cláusulas else-if, na qual os Es são expressões lógicas e os Ss são sentenças, é dada aqui:

```
 if E1 goto 1
 if E2 goto 2
 . . .
1: S1
 goto out
2: S2
 goto out
. . .
out: . . .
```

A partir dessa descrição, podemos perceber a diferença entre estruturas de seleção múltipla e sentenças else-if: em uma construção de seleção múltipla, todos os Es seriam restritos a comparações entre o valor de uma única expressão e alguns outros valores.

Linguagens que não incluem a sentença else-if podem usar a mesma estrutura de controle, apenas com um pouco mais de digitação.

O exemplo acima, da sentença if-then-else-if de Python, pode ser escrito como a sentença `case` em Ruby:

```
case
when count < 10 then bag1 = true
when count < 100 then bag2 = true
when count < 1000 then bag3 = true
end
```

Sentenças else-if são baseadas em uma sentença matemática comum, a expressão condicional.

O seletor múltiplo de Scheme, baseado em expressões condicionais matemáticas, é uma função de forma especial chamada `COND`. `COND` é uma versão ligeiramente generalizada da expressão matemática condicional; ela permite que mais de um predicado seja verdadeiro ao mesmo tempo. Como expressões matemáticas condicionais diferentes têm números diferentes de parâmetros, `COND` não exige um número fixo de parâmetros reais. Cada parâmetro para `COND` é um par de expressões no qual a primeira é um predicado (é avaliado como `#T` ou `#F`).

A forma geral de `COND` é

```
(COND
 (predicado1 expressão1)
 (predicado2 expressão2)
  . . .
 (predicadon expressãon)
 [(ELSE expressãon+1)]
)
```

onde a cláusula `ELSE` é opcional.

A semântica de `COND` é a seguinte: os predicados dos parâmetros são avaliados, um de cada vez, e ordenados a partir do primeiro, até que um deles seja avaliado como `#T`. A expressão que segue o primeiro predicado encontrado como `#T` é avaliada e seu valor é retornado como o valor de `COND`. Se nenhum dos predicados for verdadeiro e existir um `ELSE`, sua expressão será avaliada, e o valor, retornado. Se nenhum dos predicados for

verdadeiro e não existir um ELSE, o valor de COND será não especificado. Logo, todos os CONDs devem incluir um ELSE.

Considere o seguinte exemplo de chamada a COND:

```
(COND
  ((> x y) "x is greater than y")
  ((< x y) "y is greater than x")
  (ELSE "x and y are equal")
)
```

Observe que literais de cadeias são avaliados como eles próprios, de modo que, quando essa chamada a COND é avaliada, produz um resultado de cadeia.

8.3 SENTENÇAS DE ITERAÇÃO

Uma **sentença de iteração** é aquela que faz uma sentença ou uma coleção de sentenças ser executada nenhuma, uma ou mais vezes. Uma sentença de iteração é também chamada de **laço**. Todas as linguagens de programação, desde Plankalkül, incluem algum método de repetição da execução de segmentos de código. A iteração é a essência do poder da computação. Se não fosse possível algum modo de execução repetitiva de uma sentença ou coleção de sentenças, os programadores precisariam informar cada ação em sequência; os programas úteis seriam imensos, inflexíveis, levariam um tempo inaceitavelmente longo para serem escritos e exigiriam muita memória para serem armazenados.

As primeiras sentenças iterativas em linguagens de programação eram diretamente relacionadas às matrizes. Por isso, nos primeiros anos dos computadores, a computação era amplamente numérica em sua natureza, usando laços para processar dados em matrizes.

Diversas categorias de sentenças de controle de iteração foram desenvolvidas. As principais categorias são definidas a partir de como os projetistas responderam duas questões de projeto básicas:

- Como a iteração é controlada?
- Onde o mecanismo de controle deve aparecer na sentença de laço?

As possibilidades primárias para o controle de iteração são o uso de controles lógicos, de contagem ou uma combinação dos dois. As principais escolhas para a posição do mecanismo de controle são o início ou o final do laço. Aqui, início e fim são denotações lógicas, em vez de físicas. A questão não é a posição física do mecanismo de controle; em vez disso, é se o mecanismo é executado e se afeta o controle antes ou após a execução do corpo da sentença. Uma terceira opção, que permite ao usuário decidir onde colocar o controle – em cima, embaixo ou mesmo dentro do segmento controlado –, é discutida na Seção 8.3.3.

O **corpo** de uma sentença de iteração é a coleção de sentenças cuja execução é controlada pela sentença de iteração. Usamos o termo **pré-teste** a fim de dizer que o teste para completar o laço ocorre antes que o corpo do laço seja executado, e o termo **pós-teste** a fim de dizer que ele ocorre após a execução. A sentença de iteração e o corpo do laço associado formam uma **sentença de iteração**.

8.3.1 Laços controlados por contador

Uma sentença de controle iterativa de contagem tem uma variável, chamada de **variável de laço**, na qual o valor de contagem é mantido. Ela também inclui alguma forma de especificar os valores **inicial** e **final** da variável de laço e a diferença entre os valores das variáveis de laço sequenciais, chamada de **tamanho do passo**. As especificações de início, fim e tamanho do passo de um laço são chamadas de **parâmetros do laço**.

Apesar de laços controlados logicamente serem mais gerais que laços controlados por contador, eles não são necessariamente mais usados. Como os laços controlados por contador são mais complexos, seu projeto é mais trabalhoso.

Laços controlados por contador são algumas vezes suportados por instruções de máquina projetadas para esse propósito. Infelizmente, a arquitetura de máquinas pode durar mais tempo que as abordagens de programação que são mais comuns no momento do projeto da arquitetura. Por exemplo, os computadores VAX têm uma instrução muito conveniente para a implementação de laços controlados por contador com pós-teste, os quais Fortran tinha no momento do projeto do VAX (meados dos anos 1970). Na época em que os computadores VAX se tornaram amplamente usados, porém, Fortran não tinha mais um laço (ele havia sido substituído por um laço com pré-teste). Além disso, nenhuma outra linguagem amplamente utilizada da época tinha um laço de contagem pós-teste.

8.3.1.1 Questões de projeto

Existem muitas questões de projeto para sentenças iterativas controladas por contador. A natureza da variável de laço e dos parâmetros de laço fornece diversas questões de projeto. O tipo da variável de laço e dos parâmetros de laço obviamente devem ser o mesmo, ou ao menos compatível; mas que tipos devem ser permitidos? Uma escolha evidente é o tipo inteiro, mas e as enumerações, os caracteres e os pontos flutuantes? Outra questão é se a variável de laço é uma variável normal, em termos de escopo, ou se deve ter algum escopo especial. Permitir que o usuário modifique a variável ou os parâmetros de laço dentro do laço pode levar a um código muito complexo; portanto, outra questão é se a maior flexibilidade que pode ser obtida por se permitir tais mudanças vale a pena em termos da complexidade adicional. Uma questão similar envolve o número de vezes e o momento específico em que os parâmetros de laço são avaliados: se for apenas uma vez, isso resulta em laços mais simples, porém menos flexíveis.

A seguir, temos um resumo dessas questões de projeto:

- Quais são o tipo e o escopo da variável de laço?
- A variável ou os parâmetros de laço devem ser modificados no laço? E, se isso for possível, essa mudança afeta o controle do laço?
- Os parâmetros de laço devem ser avaliados apenas uma vez ou uma vez para cada iteração?

8.3.1.2 A sentença `for` das linguagens baseadas em C

A forma geral da sentença **`for`** de C é

```
for (expressão_1; expressão_2; expressão_3)
  corpo do laço
```

O corpo do laço pode ser uma única sentença, uma sentença composta ou uma sentença nula.

Como as sentenças de atribuição em C produzem resultados e podem ser consideradas expressões, as expressões em uma sentença **for** geralmente são sentenças de atribuição. A primeira expressão é para inicialização e é avaliada uma única vez, quando a execução da sentença **for** inicia. A segunda expressão é o controle do laço e é avaliada antes da execução do corpo do laço. Como é comum em C, um valor igual a zero significa falso e todos os valores diferentes de zero significam verdadeiro. Dessa forma, se o valor da segunda expressão é igual a zero, o **for** é terminado; caso contrário, as sentenças do corpo do laço são executadas. Em C99, a expressão também pode ser do tipo booleano. Um tipo booleano em C99 armazena apenas os valores 0 e 1. A última expressão no **for** é executada após cada execução do corpo do laço. Ela é bastante usada para incrementar o contador de laço. Uma descrição em semântica operacional da sentença **for** de C é mostrada a seguir. Como as expressões em C podem ser usadas como sentenças, as avaliações de expressões são mostradas como sentenças.

```
   expressão_1
loop:
    if expressão_2 = 0 goto out
    [corpo do laço]
    expressão_3
    goto loop
out: . . .
```

A seguir, temos um exemplo de um esqueleto em C da sentença **for** de C:

```
for (count = 1; count <= 10; count++)
  . . .
}
```

Todas as expressões do **for** em C são opcionais. Uma segunda expressão ausente é considerada verdadeira; assim, um **for** sem uma é potencialmente um laço infinito. Se a primeira ou a terceira expressão estiverem ausentes, nada se presume. Por exemplo, se a primeira expressão está ausente, significa que nenhuma inicialização é realizada.

Note que o **for** em C não precisa contar. Ele pode facilmente modelar contagem *e* estruturas de laço lógicas, conforme demonstrado na seção a seguir.

As escolhas de projeto do **for** de C são as seguintes: não existem variáveis de laço nem parâmetros de laço explícitos; todas as variáveis envolvidas podem ser modificadas no corpo do laço; as expressões são avaliadas na ordem informada previamente; e, embora isso possa gerar confusão, é válido desviar para o corpo de um laço **for**.

O **for** de C é um dos mais flexíveis, porque cada uma das expressões pode ser composta de várias expressões, o que por sua vez permite múltiplas variáveis de laço, que podem ser de qualquer tipo. Quando múltiplas expressões são usadas em uma única expressão de uma sentença **for**, elas são separadas por vírgulas. Todas as sentenças em C têm valores, e essa forma de expressões múltiplas não é uma exceção. O valor de tal expressão múltipla é o do último componente.

Considere a seguinte sentença **for**:

```
for (count1 = 0, count2 = 1.0;
     count1 <= 10 && count2 <= 100.0;
     sum = ++count1 + count2, count2 *= 2.5);
```

A sua descrição em semântica operacional é

```
    count1 = 0
    count2 = 1.0
loop:
     if count1 > 10 goto out
     if count2 > 100.0 goto out
    count1 = count1 + 1
    sum  = count1 + count2
    count2 = count2 * 2.5
    goto loop
out: …
```

A sentença de exemplo do **for** em C não precisa de (e não tem) um corpo de laço. Todas as ações desejadas fazem parte da sentença **for** propriamente dita, e não de seu corpo. A primeira e a terceira expressões são sentenças múltiplas. Em ambos os casos, a expressão completa é avaliada, mas o valor resultante não é usado no controle do laço.

A sentença **for** de C99 e C++ difere daquela das versões antigas de C de duas maneiras. Primeiro, além de uma expressão aritmética, ela pode usar uma expressão booleana para controle de laço. Segundo, a primeira expressão pode incluir definições de variáveis. Por exemplo,

```
for (int count = 0; count < len; count++) { . . . }
```

O escopo de uma variável definida na sentença **for** inclui desde sua definição até o final do corpo do laço.

A sentença **for** de Java e C# é parecida com a de C++, exceto que a expressão de controle de laço é restrita a **valores booleanos**.

Em todas as linguagens baseadas em C, os dois últimos parâmetros de laço são avaliados a cada iteração. Além disso, variáveis que aparecem na expressão de parâmetros de laço podem ser modificadas no corpo do laço. Logo, esses laços podem ser complexos e potencialmente menos confiáveis.

8.3.1.3 A sentença **for** de Python

A forma geral da sentença **for** de Python é

```
for variável_de_laço in objeto:
 - corpo do laço
[else:
 - cláusula senão]
```

O valor do objeto é atribuído à variável do laço, uma vez para cada execução do corpo do laço. A cláusula senão, quando presente, é executada se o laço termina normalmente.

Considere o seguinte exemplo:

```
for count in [2, 4, 6]:
   print count
```

que produz

```
2
4
6
```

Para a maioria dos laços de contagem mais simples em Python, a função **range** é usada. Essa função recebe um, dois ou três parâmetros. Os seguintes exemplos demonstram as ações de **range**:

```
range(5) returns [0, 1, 2, 3, 4]
range(2, 7) returns [2, 3, 4, 5, 6]
range(0, 8, 2) returns [0, 2, 4, 6]
```

Note que **range** nunca retorna o valor mais alto em uma dada faixa de parâmetros.

8.3.1.4 Laços controlados por contador em linguagens funcionais

Nas linguagens imperativas, os laços controlados por contador utilizam uma variável contadora, mas tais variáveis não existem nas linguagens funcionais puras. Em vez de iteração para controlar repetição, as linguagens funcionais usam recursão. Em vez de uma sentença, elas usam uma função recursiva. Laços de contagem podem ser simulados nas linguagens funcionais como segue: o contador pode ser um parâmetro para uma função que executa o corpo do laço repetidamente, o qual pode ser especificado em uma segunda função, enviada como parâmetro para a função de laço. Assim, essa função de laço recebe como parâmetros a função do corpo e o número de repetições.

A forma geral de uma função em F# para simular laços de contagem, chamada `forLoop` neste caso, é a seguinte:

```
let rec forLoop loopBody reps =
    if reps <= 0  then
        ()
    else
         loopBody()
         forLoop loopBody, (reps - 1);;
```

Nessa função, o parâmetro `loopBody` é a função com o corpo do laço, e o parâmetro `reps` é o número de repetições. A palavra reservada **rec** aparece antes do nome da função para indicar que ela é recursiva. Os parênteses vazios não fazem nada; eles estão lá porque, em F#, uma sentença vazia é inválida e todo **if** deve ter uma cláusula else.

8.3.2 Laços controlados logicamente

Em muitos casos, coleções de sentenças devem ser executadas repetidamente, mas o controle da repetição é baseado em uma expressão booleana, em vez de em um contador. Para essas situações, um laço controlado logicamente é conveniente. Na verdade, laços

controlados logicamente são mais gerais que laços controlados por contador. Todos os laços de contagem podem ser construídos com laços lógicos, mas o inverso não é verdadeiro. Além disso, lembre-se de que apenas a seleção e os laços lógicos são essenciais para expressar a estrutura de controle de qualquer diagrama de fluxo.

8.3.2.1 Questões de projeto

Como são muito mais simples que os laços controlados por contador, os laços controlados logicamente têm poucas questões de projeto.

- O controle deve ser de pré-teste ou pós-teste?
- O laço controlado logicamente deve ser uma forma especial de um laço de contagem ou uma sentença separada?

8.3.2.2 Exemplos

As linguagens de programação baseadas em C incluem tanto laços controlados logicamente com pré-teste quanto com pós-teste, que não são formas especiais de suas sentenças iterativas controladas por contador. Os laços lógicos com pré-teste e pós-teste têm as seguintes formas:

while (expressão_de_controle);
  corpo do laço

e

do
  corpo do laço
while (expressão_de_controle);

Essas duas formas sentenciais são exemplificadas pelos seguintes segmentos de código em C#:

```
sum = 0;
indat = Int32.Parse(Console.ReadLine());
while (indat >= 0) {
  sum += indat;
  indat = Int32.Parse(Console.ReadLine());
}
value = Int32.Parse(Console.ReadLine());
do {
   value /= 10;
   digits ++;
}  while (value > 0);
```

Todas as variáveis nesses exemplos são do tipo inteiro. O método `ReadLine` do objeto `Console` obtém uma linha de texto do teclado. `Int32.Parse` encontra o número em seu parâmetro do tipo cadeia, o converte no tipo **`int`** e o retorna.

Na versão com pré-teste de um laço lógico (**`while`**), a sentença ou segmento de sentença é executada enquanto a expressão for avaliada como verdadeira. Na sentença com

pós-teste (**do**), o corpo do laço é executado até a expressão ser avaliada como falsa. Em ambos os casos, a sentença pode ser composta. As descrições em semântica operacional dessas duas sentenças são as seguintes:

while

loop:
  if expressão_de_controle is false **goto** out
  [corpo do laço]
  goto loop
out: . . .

do–while

loop:
  [corpo do laço]
  if expressão_de_controle is true **goto** loop

Tanto em C quanto em C++ é válido desviar para os corpos dos laços **while** e **do**. A versão C89 usa uma expressão aritmética para controle; em C99 e em C++, ela pode ser aritmética ou booleana.

As sentenças **while** e **do** são similares àquelas de C e C++, exceto pela expressão de controle, que deve ser do tipo **booleano**. E, como Java não possui um goto, os corpos dos laços não podem ser acessados diretamente de qualquer lugar, exceto a partir de seu início.

Laços com pós-teste não são usados com frequência e podem ser perigosos, porque os programadores algumas vezes esquecem que o corpo do laço será sempre executado ao menos uma vez. O projeto sintático de colocar um controle pós-teste fisicamente após o bloco do laço, onde ele tem seu efeito semântico, ajuda a evitar tais problemas ao tornar a lógica clara.

Um laço lógico com pré-teste pode ser simulado em uma forma puramente funcional, com uma função recursiva semelhante àquela usada para simular uma sentença de laço de contagem na Seção 8.3.1.5. Em ambos os casos, o corpo do laço é escrito como uma função. A seguir está a forma geral de um laço lógico com pré-teste simulado, escrito em F#:

```
let rec whileLoop test body =
    if test() then
        body()
        whileLoop test body
     else
         ();;
```

8.3.3 Mecanismos de controle de laços posicionados pelo usuário

Em algumas situações, é conveniente para um programador escolher uma posição para o controle do laço, em vez de o início ou o final do corpo deste. Por isso, algumas

linguagens fornecem tal capacidade. Um mecanismo sintático para controle de laços posicionado pelo usuário pode ser relativamente simples, então seu projeto não é difícil. Tais laços têm a estrutura de laços infinitos, mas incluem uma ou mais saídas de laço posicionadas pelos usuários. Talvez a questão mais interessante seja se um único laço ou diversos laços aninhados podem ser abandonados. As questões de projeto para tal mecanismo são:

- O mecanismo condicional deve ser parte integrante da saída?
- É possível sair apenas de um corpo de laço ou é possível sair também dos laços que o envolvem?

C, C++, Python, Ruby e C# têm saídas não rotuladas incondicionais (**break**). Java e Perl têm saídas incondicionais rotuladas (**break** em Java, **last** em Perl).

A seguir, temos um exemplo de laços aninhados em Java, nos quais existe um break para fora do laço externo a partir do laço interno.

```
outerLoop:
  for (row = 0; row < numRows; row++)
     for  (col = 0; col < numCols; col++) {
       sum += mat[row][col];
       if (sum > 1000.0)
         break outerLoop;
     }
```

C, C++ e Python incluem uma sentença de controle não rotulada, chamada **continue**, que transfere o controle para o mecanismo do menor laço que o envolve. Essa não é uma saída, mas uma maneira de pular o resto das sentenças do laço na iteração atual, sem terminar a construção do laço. Por exemplo, considere o seguinte:

```
while (sum < 1000) {
  getnext(value);
  if (value < 0) continue;
  sum += value;
}
```

Um valor negativo faz a sentença de atribuição ser pulada e, em vez disso, o controle é transferido para a condicional no topo do laço. Por outro lado, em

```
while (sum < 1000) {
  getnext(value);
  if (value < 0) break;
  sum += value;
}
```

um valor negativo termina o laço.

Tanto **last** quanto **break** fornecem múltiplas saídas de laços, o que pode parecer prejudicial para a legibilidade. Entretanto, condições não usuais que exigem o término do laço são tão comuns que tal sentença se justifica. Além disso, a legibilidade não é seriamente prejudicada, porque o alvo de tais saídas de laço é a primeira sentença após o laço (ou de um laço que o envolve), em vez de qualquer lugar no programa. Por fim, a

alternativa de usar múltiplos breaks para deixar mais de um nível de laços é pior ainda para a legibilidade.

O objetivo das saídas de laços posicionadas pelo usuário é simples: elas atendem a uma necessidade comum de sentenças goto por meio de uma sentença de desvio altamente restrita. O alvo de um goto pode ser muitos lugares em um programa, tanto acima quanto abaixo do goto propriamente dito. Entretanto, os alvos de saídas de laços posicionadas pelo usuário devem ser abaixo da saída e só podem vir imediatamente após o final de uma sentença composta.

8.3.4 Iteração baseada em estruturas de dados

Uma sentença de iteração geral baseada em dados usa uma estrutura de dados e uma função definida pelo usuário (o iterador) para navegar nos elementos da estrutura. O iterador é chamado no início de cada iteração e, cada vez que ele é chamado, retorna um elemento de uma estrutura de dados em particular, em alguma ordem específica. Por exemplo, suponha que um programa tenha uma árvore binária definida pelo usuário composta de nós de dados, e que os dados em cada nó devam ser processados em determinada ordem. Uma sentença de iteração definida pelo usuário para a árvore configuraria sucessivamente a variável do laço para apontar para os nós da árvore, um para cada iteração. A execução inicial da sentença de iteração definida pelo usuário precisa realizar uma chamada especial para o iterador, a fim de obter o primeiro elemento da lista. O iterador deve sempre lembrar qual nó apresentou pela última vez, de forma a visitar todos os nós sem visitar nenhum mais de uma vez. Logo, um iterador deve ser sensível ao histórico. Uma sentença de iteração definida pelo usuário termina quando o iterador não encontra mais elementos.

A sentença **for** das linguagens baseadas em C, devido à sua grande flexibilidade, pode ser usada para simular uma sentença de iteração definida pelo usuário. Mais uma vez, suponha que os nós de uma árvore binária estejam para ser processados. Se o nó raiz fosse apontado por uma variável chamada `root` e se `traverse` fosse uma função que modificasse seu parâmetro para apontar para o próximo elemento de uma árvore na ordem desejada, o código a seguir poderia ser usado:

```
for (ptr = root; ptr == null; ptr = traverse(ptr)) {
  . . .
}
```

Nessa sentença, `traverse` é o iterador.

Iteradores predefinidos são usados para fornecer acesso iterativo para as matrizes únicas de PHP. O ponteiro **current** aponta para o elemento acessado pela última vez pelo iterador. O iterador **next** move **current** para o próximo elemento na matriz. O iterador **prev** move **current** para o elemento anterior. O ponteiro **current** pode ser modificado ou reiniciado para o primeiro elemento de uma matriz com o operador **reset**. O código a seguir mostra os elementos de uma matriz de números `$list`:

```
reset $list;
print ("First number: " + current($list) + "");
while ($current_value = next($list))
  print ("Next number: " + $current_value + "<br \>");
```

Sentenças de iteração definidas pelo usuário são mais importantes na programação orientada a objetos do que eram em paradigmas anteriores de desenvolvimento de software, porque os usuários de programação orientada a objetos utilizam tipos de dados abstratos rotineiramente para estruturas de dados, especialmente para coleções. Em tais casos, uma sentença de iteração definida pelo usuário e seu iterador podem ser fornecidos pelo autor da abstração de dados, porque a representação dos objetos do tipo não é conhecida pelo usuário.

Uma versão melhorada da sentença **for** foi adicionada à linguagem Java em sua versão 5.0. Essa sentença simplifica a iteração por meio dos valores em uma matriz ou dos objetos em uma coleção que implementa a interface `Iterable`. (Em Java, todas as coleções genéricas predefinidas implementam `Iterable`.) Por exemplo, se tivéssemos uma coleção `ArrayList`[4] de cadeias, chamada `myList`, a seguinte sentença iteraria por todos os seus elementos, configurando cada um como `myElement`:

```
for (String myElement : myList) { . . . }
```

Essa nova sentença é chamada de "foreach", apesar de sua palavra reservada ser **for**.

C# e F# (e as outras linguagens .NET) também têm classes de biblioteca genéricas para coleções. Por exemplo, existem classes de coleção genéricas para listas, as quais são matrizes de tamanho dinâmico, pilhas, filas e dicionários (tabela de dispersão). Todas essas coleções genéricas predefinidas têm iteradores internos utilizados implicitamente com a sentença **foreach**. Além disso, os usuários podem definir suas próprias coleções e escrever seus próprios iteradores, os quais podem implementar a interface `IEnumerator`, que permite o uso de **foreach** nessas coleções.

Por exemplo, considere o código C# a seguir:

```
List<String> names = new List<String>();
names.Add("Bob");
names.Add("Carol");
names.Add("Alice");
. . .
foreach (String name in names)
  Console.WriteLine(name);
```

Em Ruby, um **bloco** é uma sequência de código delimitada por chaves ou pelas palavras reservadas **do** e **end**. Os blocos podem ser usados com métodos especialmente escritos para criar muitas construções úteis, incluindo iteradores para estruturas de dados. Essa construção consiste em uma chamada de método seguida por um bloco. Na verdade, um bloco é um método anônimo enviado ao método (cuja chamada o precede) como parâmetro. Então, o método chamado pode chamar o bloco, o qual pode produzir saída ou objetos.

Ruby predefine vários métodos iteradores, como `times` e `upto` para laços controlados por contador, e `each` para iterações simples de matrizes e dispersões. Por exemplo, considere o seguinte exemplo de uso de `times`:

[4] `ArrayList` é uma coleção predefinida; na verdade, uma matriz de tamanho dinâmico que pode ser de qualquer tipo.

```
>> 4.times {puts "Hey!"}
Hey!
Hey!
Hey!
Hey!
=> 4
```

Observe que >> é o prompt do interpretador interativo de Ruby e que => é usado para indicar o valor de retorno da expressão. A sentença Ruby `puts` mostra seu parâmetro. Nesse exemplo, o método `times` é enviado para o objeto `4`, e o bloco é enviado junto como parâmetro. O método `times` chama o bloco quatro vezes, produzindo quatro linhas de saída. O objeto de destino, `4`, é o valor de retorno de `times`.

O iterador mais comum de Ruby é `each`, frequentemente usado para percorrer matrizes e aplicar um bloco a cada elemento.[5] Para esse propósito, é conveniente permitir que os blocos tenham parâmetros, os quais, se presentes, aparecem no início do bloco, delimitados por barras verticais (`|`). O exemplo a seguir, que usa um parâmetro de bloco, ilustra o uso de `each`:

```
>> list = [2, 4, 6, 8]
=> [2, 4, 6, 8]
>> list.each {|value| puts value}
2
4
6
8
=> [2, 4, 6, 8]
```

Nesse exemplo, o bloco é chamado para cada elemento da matriz para o qual o método `each` é enviado. O bloco produz a saída, que é uma lista dos elementos da matriz. O valor de retorno de `each` é a matriz para a qual ele é enviado.

Em vez de um laço de contagem, Ruby tem o método `upto`. Por exemplo, poderíamos ter o seguinte:

```
1.upto(5) {|x| print x, " "}
```

Isso produz a seguinte saída:

```
1 2 3 4 5
```

Também poderia ser usada uma sintaxe semelhante a um laço **`for`** de outras linguagens, como no seguinte:

```
for x in 1..5
  print x, " "
end
```

Na verdade, Ruby não tem nenhuma sentença **`for`** – construções como a anterior são convertidas por Ruby em chamadas ao método `upto`.

[5] Isso é semelhante às funções de mapa discutidas no Capítulo 15.

Consideremos agora o funcionamento dos blocos. A sentença **`yield`** é semelhante a uma chamada de método, exceto que não há nenhum objeto receptor e que a chamada é um pedido para executar o bloco anexado à chamada de método, em vez de uma chamada a um método. **`yield`** só é chamada em um método que tenha sido chamado com um bloco. Se o bloco tem parâmetros, eles são especificados em parênteses na sentença **`yield`**. O valor retornado por um bloco é o da última expressão avaliada no bloco. Esse é o processo utilizado para implementar os iteradores internos, como `times`.

8.4 DESVIO INCONDICIONAL

Uma **sentença de desvio incondicional** transfere o controle da execução para uma posição especificada no programa. A questão mais debatida acerca do projeto de linguagens no fim dos anos 1960 era se o uso de desvios incondicionais deveria fazer parte de qualquer linguagem de alto nível e, caso isso ocorresse, se seu uso deveria ser restrito. O desvio incondicional, ou goto, é a sentença mais poderosa para controlar o fluxo de execução das sentenças de um programa. Entretanto, usá-lo sem cuidados pode levar a sérios problemas. O goto tem uma força surpreendente e uma ótima flexibilidade (todas as outras estruturas de controle podem ser construídas com goto e um seletor), mas esse poder torna o seu uso perigoso. Sem restrições em seu uso, impostas ou pelo projeto da linguagem ou por padrões de programação, as sentenças goto podem tornar os programas muito difíceis de serem lidos e, como resultado, altamente não confiáveis e com elevado custo de manutenção.

Esses problemas são oriundos de uma das capacidades do goto, a de forçar qualquer sentença de programação após qualquer ordem na sequência de execução, independentemente de a sentença preceder ou seguir a executada anteriormente em ordem textual. A legibilidade é melhor quando a ordem de execução das sentenças em um programa é quase a mesma daquela na qual elas aparecem – em nosso caso, significa de cima para baixo, ou seja, a ordem a que estamos acostumados. Logo, restringir os gotos de forma que possam transferir o controle apenas para baixo em um programa atenua parcialmente o problema. Isso permite o uso de gotos para transferir o controle entre seções de código em resposta a erros ou condições não usuais, mas não permite seu uso para construir qualquer tipo de laço.

Poucas linguagens foram projetadas sem um goto – por exemplo, Java, Python e Ruby; a maioria das linguagens populares o inclui. Kernighan e Ritchie (1978) afirmam que o uso do goto é exagerado, mas mesmo assim foi ele incluído na linguagem de Ritchie, C. As linguagens que eliminaram o goto fornecem sentenças de controle adicionais, normalmente na forma de saídas de laços, para codificar uma de suas aplicações mais justificáveis.

> **» *nota histórica***
>
> Apesar de diversos estudiosos terem mencionado antes, foi Edsger Dijkstra quem publicou o primeiro texto amplamente conhecido sobre os perigos do goto. Em sua carta, ele escreveu que "a sentença goto nos moldes de hoje é primitiva demais; é um convite muito enfático para bagunçar o programa de alguém" (Dijkstra, 1968a). Nos primeiros anos após a publicação do parecer de Dijkstra sobre o goto, diversas pessoas pediram publicamente o banimento ou, ao menos, restrições ao uso do goto. Entre aqueles que não eram favoráveis à eliminação completa estava Donald Knuth (1974), que argumentou que existem ocasiões em que a eficiência do goto supera seu prejuízo para a legibilidade.

Relativamente nova, C# inclui um goto, apesar de uma das linguagens nas quais se baseia, Java, não incluir. Um uso legítimo do goto de C# é na sentença **`switch`**, conforme discutido na Seção 8.2.2.2.

Todas as sentenças de saída de laços discutidas na Seção 8.3.3 são sentenças goto camufladas. Entretanto, são gotos restritos e não prejudiciais à legibilidade. Na verdade, pode-se argumentar que elas melhoram a legibilidade, já que evitar seu uso resulta em código complicado e não natural, muito mais difícil de ser entendido.

8.5 COMANDOS PROTEGIDOS

Formas novas e bastante diferentes de seleção e de estruturas de laço foram sugeridas por Dijkstra (1975). Seu objetivo primário era fornecer sentenças de controle que suportassem uma metodologia de projeto de programas para garantir a corretude durante o desenvolvimento, e não ao verificar ou testar programas completos. Essa metodologia é descrita em Dijkstra (1976). Outro objetivo era conseguir clareza no entendimento do que é possível com os comandos protegidos. De forma simples, um segmento selecionável de uma sentença de seleção em uma de comando protegido pode ser considerado independentemente de qualquer outra parte da sentença, o que não é verdade para as sentenças de seleção das linguagens de programação comuns.

Comandos protegidos são abordados neste capítulo porque formam a base para o mecanismo linguístico desenvolvido posteriormente para programação concorrente em CSP (Hoare, 1978). Os comandos protegidos também são úteis para definir funções em Haskell, conforme discutido no Capítulo 15.

A sentença de seleção de Dijkstra tem a forma

```
if <expressão booleana> -> <sentença>
[] <expressão booleana> -> <sentença>
[] . . .
[] <expressão booleana> -> <sentença>
fi
```

A palavra reservada de fechamento, **`fi`**, é a de abertura escrita de trás para frente. Essa forma de fechar palavras reservadas é oriunda de ALGOL 68. Os pequenos blocos, chamados de *fatbars*, são usados para separar as cláusulas protegidas e permitir que as cláusulas sejam sequências de sentenças. Cada linha na sentença de seleção, que consiste em uma expressão booleana (uma guarda) e uma sentença ou sequência de sentenças, é chamada de **comando protegido**.

A sentença de seleção tem a aparência de uma seleção múltipla, mas sua semântica é diferente. Todas as expressões booleanas são avaliadas sempre que a sentença é alcançada durante a execução. Se mais de uma expressão for verdadeira, uma das sentenças correspondentes pode ser escolhida de maneira não determinística para execução. Uma implementação pode sempre escolher a sentença associada à primeira expressão booleana avaliada como verdadeira. Mas ela pode escolher qualquer sentença associada a uma expressão booleana verdadeira. Logo, a corretude do programa não depende de qual sentença é escolhida (entre aquelas associadas com expressões booleanas verdadeiras). Se nenhuma das expressões booleanas é verdadeira, ocorre um erro em tempo de execução

que causa o término do programa. Isso obriga o programador a considerar e listar todas as possibilidades. Considere o seguinte exemplo:

```
if i = 0 -> sum := sum + i
[] i > j -> sum := sum + j
[] j > i -> sum := sum + i
fi
```

Se `i = 0` e `j > i`, essa sentença escolhe não deterministicamente entre a primeira e a terceira sentenças de atribuição. Se `i` é igual a `j` e não é zero, ocorre um erro em tempo de execução, porque nenhuma das condições é verdadeira.

Essa sentença pode ser uma maneira elegante de permitir que o programador defina que a ordem de execução, em alguns casos, é irrelevante. Por exemplo, para encontrar o maior entre dois números, poderíamos usar

```
if x >= y -> max := x
[] y >= x -> max := y
fi
```

Isso computa o resultado desejado sem especificar demasiadamente a solução. Em particular, se `x` e `y` forem iguais, não interessa qual atribuímos a `max`. Essa é uma forma de abstração fornecida pela semântica não determinística da sentença.

Agora, considere esse mesmo processo codificado em um seletor de uma linguagem de programação tradicional:

```
if (x >= y)
  max = x;
else
  max = y;
```

Isso também poderia ser codificado como:

```
if (x > y)
  max = x;
else
  max = y;
```

Não existe diferença prática entre essas duas sentenças. A primeira atribui `x` a `max` quando `x` e `y` são iguais; a segunda atribui `y` a `max` na mesma circunstância. A escolha entre as duas sentenças complica a análise formal do código e a sua prova de corretude. Essa é uma das razões pelas quais os comandos protegidos foram desenvolvidos por Dijkstra.

A estrutura de laço proposta por Dijkstra tem a forma

do <expressão booleana> -> <sentença>
[] <expressão booleana> -> <sentença>
[] . . .
[] <expressão booleana> -> <sentença>
od

A semântica dessa sentença é que todas as expressões booleanas são avaliadas em cada iteração. Se mais de uma é verdadeira, uma das sentenças associadas é não deterministicamente (talvez aleatoriamente) escolhida para execução, após a qual as expressões são avaliadas novamente. Quando todas as expressões são simultaneamente avaliadas como falsas, o laço termina.

Considere o seguinte problema: dadas quatro variáveis inteiras, `q1`, `q2`, `q3` e `q4`, rearranje seus valores de forma que `q1 ? q2 ? q3 ? q4`. Sem comandos protegidos, uma solução direta é colocar os valores em uma matriz, ordená-la e então atribuir os valores da matriz de volta às variáveis escalares `q1`, `q2`, `q3` e `q4`. Embora essa solução não seja difícil, ela requer boa quantidade de código, especialmente se o processo de ordenação precisar ser incluído.

Agora, considere o seguinte código, que usa comandos protegidos para resolver o mesmo problema, mas de forma mais concisa e elegante.[6]

```
do q1 > q2 -> temp := q1; q1 := q2; q2 := temp;
[] q2 > q3 -> temp := q2; q2 := q3; q3 := temp;
[] q3 > q4 -> temp := q3; q3 := q4; q4 := temp;
od
```

As sentenças de controle de comandos protegidos de Dijkstra são interessantes, em parte porque ilustram como a sintaxe e a semântica das sentenças podem ter impacto na verificação de programas e vice-versa. A verificação de programas é praticamente impossível quando sentenças goto são usadas. Por outro lado, ela é amplamente simplificada se (1) são usados apenas laços lógicos e seleções ou (2) são usados apenas comandos protegidos. A semântica axiomática de comandos protegidos é convenientemente especificada (Gries, 1981). Deve ser óbvio, entretanto, que existe um aumento considerável de complexidade na implementação dos comandos protegidos em relação aos seus respectivos comandos determinísticos convencionais.

8.6 CONCLUSÕES

Descrevemos e discutimos uma variedade de estruturas de controle no nível de sentenças. Uma breve avaliação agora parece necessária.

Primeiro, temos o resultado teórico de que apenas sequências, seleções e laços lógicos com pré-testes são absolutamente necessários para expressar computações (Böhm e Jacopini, 1966). Esse resultado tem sido usado por aqueles que desejam banir completamente os desvios incondicionais. É claro que já existem problemas práticos o suficiente com o goto para condená-lo sem ao menos recorrer à teoria. Uma das necessidades legítimas principais para os gotos – saídas prematuras de laços – pode ser suprida com sentenças de desvio restritas, como **`break`**.

Um dos usos incorretos óbvios dos resultados de Böhm e Jacopini é argumentar contra a inclusão de *quaisquer* estruturas de controle além de seleção e laços lógicos com pré-testes. Nenhuma linguagem popular deu esse passo; além disso, duvidamos que alguma o dará, em função do efeito negativo na facilidade de escrita e na legibilidade. Programas escritos com apenas seleção e laços lógicos com pré-teste são menos naturais

[6] Este código aparece de forma um pouco diferente em Dijkstra (1975).

em sua estrutura, mais complexos e, dessa forma, mais difíceis de serem escritos e lidos. Por exemplo, a estrutura de seleção múltipla de C# aumenta bastante a facilidade de escrita da linguagem, sem um ponto negativo óbvio. Outro exemplo é a estrutura de laço de contagem de muitas linguagens, especialmente quando a sentença é simples.

A utilidade de muitas das outras estruturas de controle que têm sido propostas não está tão clara a ponto de determinarmos se vale a pena incluí-las em linguagens (Ledgard e Marcotty, 1975). Essa questão está relacionada ao tópico fundamental acerca do tamanho das linguagens (se ele deve ser minimizado ou não). Tanto Wirth (1975) quanto Hoare (1973) defendem a simplicidade no projeto de linguagens. No caso das estruturas de controle, simplicidade significa que apenas algumas sentenças devem estar presentes em uma linguagem e que devem ser simples.

A grande variedade de estruturas de controle no nível de sentenças que têm sido desenvolvidas mostra a diversidade de opiniões entre os projetistas de linguagens. Após toda a invenção, a discussão e a avaliação, não existe unanimidade de opiniões acerca do conjunto preciso de sentenças de controle que deve ser incluído em uma linguagem. A maioria das linguagens de programação, no entanto, tem sentenças de controle similares, mas ainda existe alguma variação nos detalhes de sua sintaxe e semântica. Além disso, existem discordâncias a respeito da inclusão do goto por parte das linguagens; C++ e C# o fazem, mas Java e Ruby não.

RESUMO

As sentenças de controle ocorrem em diversas categorias: seleção, seleção múltipla, iteração e desvio incondicional.

A sentença **`switch`** das linguagens baseadas em C é representativa das sentenças de seleção múltipla. A versão de C# elimina o problema de confiabilidade de suas antecessoras ao proibir a continuação implícita de um segmento selecionado para o próximo segmento selecionável.

Muitas sentenças de laço diferentes foram desenvolvidas para as linguagens de alto nível. A sentença **`for`** de C é a sentença de iteração mais flexível, apesar de essa flexibilidade causar alguns problemas de confiabilidade.

A maioria das linguagens têm sentenças de saída para seus laços; essas substituem um dos usos mais comuns das sentenças goto.

Iteradores baseados em dados são sentenças de laço para processar estruturas de dados, como listas encadeadas, dispersões e árvores. A sentença **`for`** das linguagens baseadas em C permite ao usuário criar iteradores para dados por ele definidos. A sentença **`foreach`** de Perl e C# é um iterador predefinido para estruturas de dados padrão. Nas linguagens orientadas a objetos contemporâneas, os iteradores para coleções são especificados com interfaces padrão, implementadas pelos projetistas das coleções.

Ruby contém iteradores, uma forma especial de métodos que são enviados para vários objetos. A linguagem predefine iteradores para usos comuns, mas também permite iteradores definidos pelo usuário.

O desvio incondicional, ou goto, faz parte da maioria das linguagens imperativas. Seus problemas são muito debatidos. O consenso atual é que ele deve permanecer na maioria das linguagens, mas seus perigos devem ser minimizados com disciplina na programação.

Os comandos protegidos de Dijkstra são sentenças de controle alternativas com características teóricas positivas. Apesar de os mecanismos de concorrência de CSP e as definições de funções em Haskell não terem sido adotados como sentenças de controle de uma linguagem, parte da semântica aparece nele.

QUESTÕES DE REVISÃO

1. Qual é a definição de *estrutura de controle*?
2. O que Böhm e Jocopini provaram a respeito dos diagramas de fluxo?
3. Qual é a definição de *bloco*?
4. Quais são as questões de projeto comuns a todas as sentenças de seleção e de controle de iteração?
5. Quais são as questões de projeto para estruturas de seleção?
6. O que é não usual acerca do projeto de Python em termos de sentenças compostas?
7. Em que circunstâncias um seletor em F# deve ter uma cláusula else?
8. Quais são as soluções para o problema de aninhamento para os seletores de dois caminhos?
9. Quais são as questões de projeto para sentenças de seleção múltipla?
10. Entre quais duas características de linguagem existe um compromisso ao decidir se mais de um segmento selecionável é executado em uma sentença de seleção múltipla?
11. O que é não usual acerca da sentença de seleção múltipla de C?
12. Em que linguagem anterior a sentença **`switch`** de C foi baseada?
13. Explique como a sentença **`switch`** de C# é mais segura que a de C.
14. Quais são as questões de projeto para todas as sentenças de controle iterativas?
15. Quais são as questões de projeto para as sentenças de laços controlados por contador?
16. O que é uma sentença de laço com pré-teste? O que é uma sentença de laço com pós-teste?
17. Qual é a diferença entre a sentença **`for`** de C++ e a de Java?
18. De que maneira a sentença **`for`** de C é mais flexível que a de muitas outras linguagens?
19. O que a função **`range`** faz em Python?
20. Que linguagens contemporâneas não incluem um goto?
21. Quais são as questões de projeto para sentenças de laços controlados logicamente?
22. Qual é a principal razão para a criação das sentenças de controle de laço posicionadas pelo usuário?

23. Quais são as questões de projeto para mecanismos de controle de laço posicionados pelo usuário?
24. Qual a vantagem da sentença **`break`** de Java em relação à sentença **`break`** de C?
25. Quais são as diferenças entre a sentença **`break`** de C++ e a de Java?
26. O que é um controle de iteração definido pelo usuário?
27. Qual função de Scheme implementa uma sentença de seleção múltipla?
28. Como uma linguagem funcional implementa repetição?
29. Como os iteradores são implementados em Ruby?
30. Qual linguagem predefine iteradores que podem ser chamados explicitamente para iterar pelas suas estruturas de dados predefinidas?
31. Que linguagens de programação bastante usadas pegam emprestado parte de seu projeto dos comandos protegidos de Dijkstra?

PROBLEMAS

1. Descreva três situações nas quais uma combinação entre sentenças de laços lógicos e de contagem é necessária.
2. Estude o recurso de iteração de CLU em Liskov et al. (1981) e determine suas vantagens e desvantagens.
3. Compare o conjunto de sentenças de controle de Ada com o de C# e decida qual é o melhor e por quê.
4. Quais são os prós e os contras de usar palavras reservadas únicas de fechamento em sentenças compostas?
5. Quais são os argumentos a favor e contra o uso de endentação em Python para especificar sentenças compostas em sentenças de controle?
6. Analise os potenciais problemas de legibilidade no uso de palavras reservadas de fechamento para sentenças de controle que são o inverso de suas palavras reservadas correspondentes, como as palavras reservadas **`case–esac`** de ALGOL 68. Por exemplo, considere erros de digitação comuns, como a inversão da ordem de caracteres.
7. Use o *Science Citation Index* para encontrar um artigo que faz referência a Knuth (1974). Leia tal artigo e o de Knuth e escreva um que resuma ambos os lados da questão do goto.
8. Em seu artigo sobre a questão do goto, Knuth (1974) sugere uma sentença de controle de laço que permite múltiplas saídas. Leia o artigo e escreva uma descrição em semântica operacional da sentença.
9. Quais são os argumentos a favor e contra o uso exclusivo de expressões booleanas nas sentenças de controle em Java (em oposição a permitir também o uso de expressões aritméticas, com em C++)?

10. Descreva uma situação de programação na qual a cláusula senão na sentença **for** de Python seria conveniente.
11. Descreva três situações de programação específicas que exigem um laço de pós-teste.
12. Reflita sobre a razão pela qual o controle pode ser transferido para dentro de uma sentença de laço em C.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Reescreva o seguinte segmento de pseudocódigo usando uma estrutura de laço nas linguagens especificadas:

```
  k = (j + 13) / 27
loop:
  if k > 10 then goto out
  k = k + 1
  i = 3 * k - 1
  goto loop
out: . . .
```

   a. C, C++, Java ou C#
   b. Python
   c. Ruby

   Suponha que todas as variáveis são do tipo inteiro. Discuta qual linguagem, para esse código, tem a melhor facilidade de escrita, a melhor legibilidade e a melhor combinação de ambas.

2. Refaça o exercício anterior de forma que todas as variáveis e constantes sejam do tipo ponto flutuante e troque a sentença

```
k = k + 1
```

   por

```
k = k + 1,2
```

3. Reescreva o seguinte segmento de código usando uma sentença de seleção múltipla nas seguintes linguagens:

```
if ((k == 1) || (k == 2)) j = 2 * k - 1
if ((k == 3) || (k == 5)) j = 3 * k + 1
if (k == 4) j = 4 * k - 1
if ((k == 6) || (k == 7) || (k == 8)) j = k - 2
```

   a. C, C++, Java ou C#
   b. Python
   c. Ruby

Suponha que todas as variáveis são do tipo inteiro. Discuta os méritos relativos do uso dessas linguagens para esse código em particular.

4. Considere o segmento de programa em C a seguir. Reescreva-o sem usar gotos ou **break**s.

```
j = -3;
for (i = 0; i < 3; i++) {
  switch (j + 2) {
    case 3:
    case 2: j--; break;
    case 0: j += 2; break;
    default: j = 0;
  }
  if (j > 0) break;
  j = 3 - i
}
```

5. Em uma carta para o editor da *CACM*, Rubin (1987) usa o seguinte segmento de código como evidência de que a legibilidade de algum código com gotos é melhor que o código equivalente sem gotos. Esse código encontra a primeira linha de uma matriz inteira *n* por *n*, chamada `x`, que não tem nada além de valores iguais a zero.

```
for (i = 1; i <= n; i++) {
  for (j = 1; j <= n; j++)
     if (x[i][j] != 0)
      goto reject;
  println ('First all-zero row is:', i);
  break;
reject:
 }
```

Reescreva esse código sem gotos em uma das seguintes linguagens: C, C++, Java ou C#. Compare a legibilidade de seu código com a do exemplo.

6. Considere o seguinte problema de programação: os valores de três variáveis inteiras – `primeira`, `segunda` e `terceira` – devem ser colocados em três variáveis `max`, `meio` e `min`, com os significados óbvios, sem usar matrizes ou subprogramas definidos pelo usuário ou predefinidos. Escreva duas soluções para esse problema, uma que use seleções aninhadas e outra que não as utilize. Compare a complexidade e a legibilidade esperada das duas.

7. Reescreva o segmento de programa em C do Exercício de programação 4, usando sentenças **if** e goto em C.

8. Reescreva o segmento de programa em C do Exercício de programação 4 em Java, sem usar uma sentença **switch**.

9. Transforme para C a seguinte chamada a COND de Scheme e configure o valor resultante como y.

```
(COND
  ((> x 10) x)
  ((< x 5) (* 2 x))
  ((= x 7) (+ x 10))
)
```

Esta página foi deixada em branco intencionalmente.

9

Subprogramas

9.1 Introdução

9.2 Fundamentos dos subprogramas

9.3 Questões de projeto para subprogramas

9.4 Ambientes de referenciamento local

9.5 Métodos de passagem de parâmetros

9.6 Parâmetros que são subprogramas

9.7 Chamada indireta de subprogramas

9.8 Questões de projeto para funções

9.9 Subprogramas sobrecarregados

9.10 Subprogramas genéricos

9.11 Operadores sobrecarregados definidos pelo usuário

9.12 Fechamentos

9.13 Corrotinas

Subprogramas são os elementos fundamentais dos programas e, assim, estão entre os conceitos mais importantes no projeto de linguagens de programação. Exploraremos agora o projeto de subprogramas, incluindo métodos de passagem de parâmetro, ambientes de referenciamento local, subprogramas sobrecarregados, subprogramas genéricos, os apelidos e os efeitos colaterais problemáticos associados aos subprogramas. Incluímos também discussões sobre subprogramas chamados indiretamente, fechamentos e corrotinas.

Os métodos de implementação de subprogramas são discutidos no Capítulo 10.

9.1 INTRODUÇÃO

Dois recursos de abstração fundamentais podem ser incluídos em uma linguagem de programação: abstração de processos e abstração de dados. No princípio da história das linguagens de programação de alto nível, somente a abstração de processos era contemplada. Na forma de subprogramas, ela é um conceito central em todas as linguagens de programação. Contudo, nos anos 1980 muitas pessoas começaram a acreditar que a abstração de dados era igualmente importante. A abstração de dados é discutida em detalhes no Capítulo 11.

O primeiro computador programável, a Máquina Analítica de Babbage, construída nos anos 1840, tinha a capacidade de reutilizar coleções de cartões de instrução em vários lugares diferentes de um programa. Em uma linguagem de programação moderna, tal coleção de sentenças é escrita como um subprograma. Esse reúso resulta em economia de espaço de memória e tempo de codificação. Tal reúso é também uma abstração, pois os detalhes da computação do subprograma são substituídos em um programa por uma sentença que chama o subprograma. Em vez de descrever como alguma computação deve ser feita em um programa, essa descrição (a coleção de sentenças no subprograma) é representada por uma sentença de chamada, efetivamente abstraindo os detalhes. Isso aumenta a legibilidade de um programa por enfatizar sua estrutura lógica, ao passo que oculta seus detalhes de baixo nível.

Os métodos das linguagens orientadas a objetos são intimamente relacionados aos subprogramas discutidos neste capítulo. As principais maneiras pelas quais os métodos diferem dos subprogramas são o modo como são chamados e suas associações com classes e objetos. Embora essas características especiais dos métodos sejam discutidas no Capítulo 12, os recursos que compartilham com os subprogramas, como os parâmetros e as variáveis locais, são tratados neste capítulo.

9.2 FUNDAMENTOS DOS SUBPROGRAMAS

9.2.1 Características gerais dos subprogramas

Todos os subprogramas discutidos neste capítulo, exceto as corrotinas, descritas na Seção 9.13, têm as seguintes características:

- Cada subprograma tem um único ponto de entrada.

- A unidade de programa chamadora é suspensa durante a execução do subprograma chamado, o que significa que existe apenas um subprograma em execução em determinado momento.
- Quando a execução do subprograma termina, o controle sempre retorna para o chamador.

As alternativas a isso resultam em corrotinas e em unidades concorrentes (Capítulo 13).

Em sua maioria, os subprogramas têm nomes, embora alguns sejam anônimos. A Seção 9.12 mostra exemplos de subprogramas anônimos em C#.

9.2.2 Definições básicas

A **definição de um subprograma** descreve a interface e as ações da abstração do subprograma. Uma **chamada de subprograma** é o pedido explícito para que um subprograma específico seja executado. Diz-se que um subprograma está **ativo** se, depois de ser chamado, iniciou a execução, mas ainda não a concluiu. Os dois tipos fundamentais de subprogramas – procedimentos e funções – são definidos e discutidos na Seção 9.2.4.

O **cabeçalho de um subprograma**, que é a primeira parte da definição, tem várias finalidades. Primeiramente, ele especifica que a unidade sintática seguinte é a definição de um subprograma de algum tipo em particular.[1] Nas linguagens que possuem mais de um tipo de subprograma, o tipo do subprograma normalmente é especificado com uma palavra especial. Segundo, se o subprograma não é anônimo, o cabeçalho fornece um nome para ele. Terceiro, o cabeçalho pode especificar uma lista de parâmetros.

Considere os exemplos de cabeçalho a seguir:

```
def adder parâmetros) :
```

Esse é o cabeçalho de um subprograma em Python chamado `adder`. Os cabeçalhos de subprograma em Ruby também começam com **`def`**. O cabeçalho de um subprograma em JavaScript começa com **`function`**.

Em C, o cabeçalho de uma função chamada **`adder`** poderia ser o seguinte:

```
void adder (parâmetros)
```

A palavra reservada **`void`** nesse cabeçalho indica que o subprograma não retorna um valor.

O corpo dos subprogramas define suas ações. Nas linguagens baseadas em C (e em algumas outras – por exemplo, JavaScript), o corpo de um subprograma é delimitado por chaves. Em Ruby, uma sentença **`end`** finaliza o corpo de um subprograma. Assim como nas sentenças compostas, as sentenças no corpo de uma função em Python devem ser endentadas, e o final do corpo é indicado pela primeira sentença não endentada.

Uma característica das funções Python que as diferencia das funções de outras linguagens de programação comuns é que as sentenças de função **`def`** são executáveis. Quando uma sentença **`def`** é executada, ela atribui o nome dado ao corpo da função

[1] Algumas linguagens de programação incluem os dois tipos de subprogramas: procedimentos e funções.

dada. Até que a sentença **def** de uma função seja executada, a função não pode ser chamada. Considere o exemplo de esqueleto a seguir:

```
if . . .
  def fun(. . .):
    . . .
else
  def fun(. . .):
    . . .
```

Se a cláusula then dessa construção de seleção é executada, essa versão da função `fun` pode ser chamada, mas não a versão da cláusula else. Do mesmo modo, se a cláusula else é escolhida, sua versão da função pode ser chamada, mas a que está na cláusula then não.

Os métodos de Ruby diferem dos subprogramas de outras linguagens de programação de várias maneiras interessantes. Eles frequentemente são estipulados em definições de classe, mas também podem ser especificados fora delas, no caso em que são considerados métodos do objeto-raiz `Object`. Tais métodos podem ser chamados sem um receptor de objetos, como se fossem funções em C ou C++. Se um método Ruby é chamado sem um receptor, **self** é presumido. Se não existe um método com esse nome na classe, as classes envolventes são pesquisadas, até `Object`, se necessário.

Todas as funções Lua são anônimas, embora possam ser definidas com uma sintaxe que faz com que pareçam ter nomes. Por exemplo, considere as seguintes funções idênticas da função `cube`:

```
function cube(x) return x * x * x end

cube = function (x) return x * x * x end
```

A primeira delas usa sintaxe convencional, enquanto a forma da segunda ilustra mais precisamente o anonimato de funções.

O **perfil de parâmetros** de um subprograma contém o número, a ordem e os tipos de seus parâmetros formais. O **protocolo** de um subprograma é seu perfil de parâmetros, mais seu tipo de retorno (se for uma função). Nas linguagens nas quais os subprogramas têm tipos, esses tipos são definidos pelo protocolo do subprograma.

Os subprogramas podem ter tanto declarações como definições. Essa forma se compara com as declarações e definições de variáveis em C, na qual as declarações são usadas para fornecer informação de tipo, mas não para definir variáveis. As declarações de subprograma fornecem o protocolo do subprograma, mas não incluem seus corpos. Elas são necessárias nas linguagens que não permitem referências diretas a subprogramas. Tanto no caso das variáveis como no dos subprogramas, declarações são necessárias para verificação estática de tipos. No caso dos subprogramas, deve ser verificado o tipo dos parâmetros. Declarações de função são comuns em programas C e C++, onde são denominadas **protótipos**. Muitas vezes, essas declarações são colocadas em arquivos de cabeçalho.

Na maioria das outras linguagens (exceto C e C++), os subprogramas não precisam de declarações, pois não há o requisito de que subprogramas sejam definidos antes de serem chamados.

9.2.3 Parâmetros

Normalmente, os subprogramas descrevem computações. Há duas maneiras pelas quais um subprograma sem método pode acessar os dados que deve processar: por meio de acesso direto a variáveis não locais (declaradas em outro lugar, mas visíveis no subprograma) ou por meio de passagem de parâmetros. Os dados passados por meio de parâmetros são acessados usando-se nomes locais ao subprograma. A passagem de parâmetros é mais flexível que o acesso direto a variáveis não locais. Basicamente, um subprograma com acesso por meio de parâmetros aos dados a serem processados é uma computação parametrizada. Ele pode fazer sua computação em quaisquer dados que receba por meio de seus parâmetros (presumindo-se que os tipos dos parâmetros são os esperados pelo subprograma). Se o acesso aos dados é feito por meio de variáveis não locais, o único modo de fazer a computação em diferentes dados é atribuindo novos valores a essas variáveis não locais entre as chamadas ao subprograma. O acesso sistemático a variáveis não locais pode reduzir a confiabilidade. Variáveis que são visíveis para o subprograma onde o acesso é frequentemente desejado acabam sendo visíveis também onde o acesso a elas não é necessário. Esse problema foi discutido no Capítulo 5.

Embora os métodos também acessem dados externos por meio de referências e parâmetros não locais, o principal dado a ser processado por um método é o objeto por meio do qual é chamado. Contudo, quando um método acessa dados não locais, os problemas de confiabilidade são os mesmos dos subprogramas sem método. Além disso, em uma linguagem orientada a objetos, o acesso do método a variáveis de classe (aquelas associadas à classe e não a um objeto) está relacionado ao conceito de dados não locais e deve ser evitado quando possível. Nesse caso, assim como no caso de uma função C acessar dados não locais, o método pode ter o efeito colateral de alterar algo que não seja seus parâmetros ou dados locais. Tais alterações complicam a semântica do método e o tornam menos confiável.

As linguagens de programação funcionais puras, como Haskell, não possuem dados mutáveis; portanto, as funções nelas escritas são incapazes de alterar a memória de alguma maneira – elas simplesmente efetuam cálculos e retornam o valor resultante (ou uma função, pois funções são valores em uma linguagem funcional pura).

Em algumas situações é conveniente transmitir computações, em vez de dados, como parâmetros para subprogramas. Nesses casos, o nome do subprograma que implementa essa computação pode ser usado como parâmetro. Essa forma de parâmetro é discutida na Seção 9.6. Parâmetros de dados são discutidos na Seção 9.5.

Os parâmetros do cabeçalho do subprograma são denominados **parâmetros formais**. Às vezes, eles são considerados variáveis fictícias, pois não são variáveis no sentido usual: na maioria dos casos, são vinculadas ao armazenamento somente quando o subprograma é chamado, e essa vinculação muitas vezes se dá por meio de algumas outras variáveis de programa.

As sentenças de chamada a subprogramas devem incluir o nome do subprograma e uma lista de parâmetros a serem vinculados aos parâmetros formais do subprograma. Esses parâmetros são chamados de **parâmetros reais**.[2] Eles devem ser diferenciados dos parâmetros formais, porque os dois normalmente têm diferentes restrições em suas formas e, é claro, seus usos são bastante diferentes.

Na maioria das linguagens de programação, a correspondência entre parâmetros reais e formais – ou a vinculação de parâmetros reais a parâmetros formais – é feita pela posição: o primeiro parâmetro real é vinculado ao primeiro parâmetro formal e assim por diante. Tais parâmetros são chamados de **parâmetros posicionais**. Esse é um método eficiente e seguro de relacionar parâmetros reais aos seus formais correspondentes, desde que as listas de parâmetros sejam relativamente curtas.

No entanto, quando as listas são longas, é fácil que o programador cometa erros na ordem dos parâmetros reais da lista. Uma solução para esse problema é fornecer **parâmetros de palavra-chave**, nos quais o nome do parâmetro formal ao qual um parâmetro real deve ser vinculado é especificado com o parâmetro real em uma chamada. A vantagem dos parâmetros de palavra-chave é que podem aparecer em qualquer ordem na lista de parâmetros reais. As funções de Python podem ser chamadas com essa técnica, como em

```
sumer(length = my_length,
      list = my_array,
      sum = my_sum)
```

onde a definição de `sumer` tem os parâmetros formais `length`, `list` e `sum`.

A desvantagem dos parâmetros de palavra-chave é que o usuário do subprograma precisa saber os nomes dos parâmetros formais.

Além dos parâmetros de palavra-chave, algumas linguagens (por exemplo, Python) permitem parâmetros posicionais. Parâmetros de palavra-chave e posicionais podem ser misturados em uma chamada, como em

```
sumer(my_length,
      sum = my_sum,
      list = my_array)
```

A única restrição dessa estratégia é que, depois que um parâmetro de palavra-chave aparece na lista, todos os parâmetros restantes devem ser de palavra-chave. Essa restrição é necessária porque uma posição pode não ser mais bem definida depois que um parâmetro de palavra-chave aparecer.

Em Python, Ruby, C++ e PHP, os parâmetros formais podem ter valores padrão. Um valor padrão é usado se nenhum parâmetro real é passado para o parâmetro formal no cabeçalho do subprograma. Considere o seguinte cabeçalho de função em Python:

```
def compute_pay(income, exemptions = 1, tax_rate)
```

[2] Alguns autores chamam os parâmetros reais de *argumentos* e os parâmetros formais apenas de *parâmetros*.

O parâmetro formal `exemptions` pode estar ausente em uma chamada a `compute_pay`; quando estiver, o valor `1` será usado. Nenhuma vírgula é incluída para um parâmetro real ausente em uma chamada Python, pois a única utilidade de uma vírgula assim seria indicar a posição do próximo parâmetro, o que, neste caso, não é necessário, pois todos os parâmetros reais após um parâmetro real ausente devem ser de palavra-chave. Por exemplo, considere a seguinte chamada:

```
pay = compute_pay(20000.0, tax_rate = 0.15)
```

Em C++, que não oferece suporte para parâmetros de palavra-chave, as regras para parâmetros padrão são necessariamente diferentes. Os parâmetros padrão devem aparecer por último, pois os parâmetros são associados de acordo com a posição. Quando um parâmetro padrão é omitido em uma chamada, todos os parâmetros formais restantes devem ter valores padrão. Um cabeçalho de função em C++ para a função `compute_pay` pode ser escrito como segue:

```
float compute_pay(float income, float tax_rate,
                  int exemptions = 1)
```

Observe que os parâmetros são reorganizados de modo que aquele que tem o valor padrão seja o último. Um exemplo de chamada para a função `compute_pay` em C++ é

```
pay = compute_pay(20000.0, 0.15);
```

Na maioria das linguagens que não têm valores padrão para parâmetros formais, o número de parâmetros reais em uma chamada deve casar com o número de parâmetros formais no cabeçalho de definição do subprograma. No entanto, em C, C++, Perl, JavaScript e Lua, isso não é exigido. Quando, em uma chamada, existem menos parâmetros reais do que parâmetros formais em uma definição de função, é responsabilidade do programador garantir que a correspondência de parâmetros (que é sempre posicional) e a execução do subprograma sejam lógicas.

Embora esse projeto, que permite um número variável de parâmetros, seja claramente propenso a erros, às vezes também é conveniente. Por exemplo, a função `printf` de C pode imprimir qualquer número de itens (valores de dados e/ou literais).

C# permite que os métodos aceitem um número variável de parâmetros, desde que sejam do mesmo tipo. O método especifica seu parâmetro formal com o modificador **`params`**. A chamada pode enviar um vetor ou uma lista de expressões, cujos valores são colocados em um vetor pelo compilador e fornecidos para o método chamado. Por exemplo, considere o método:

```
public void DisplayList(params int[] list) {
   foreach (int next in list) {
      Console.WriteLine("Next value {0}", next);
   }
}
```

Se `DisplayList` está definido para a classe `MyClass` e temos as declarações

```
Myclass myObject = new Myclass;
int[] myList = new int[6] {2, 4, 6, 8, 10, 12};
```

`DisplayList` poderia ser chamada com um dos seguintes:

```
myObject.DisplayList(myList);
myObject.DisplayList(2, 4, 3 * x - 1, 17);
```

Ruby suporta uma configuração de parâmetro real complicada, mas altamente flexível. Os parâmetros iniciais são expressões cujos objetos-valor são passados para os parâmetros formais correspondentes. Os parâmetros iniciais podem ser seguidos por uma lista de pares chave => valor, os quais são colocados em uma dispersão anônima, e uma referência para essa dispersão é passada para o próximo parâmetro formal. Isso é usado como substituto para os parâmetros de palavra-chave, que Ruby não aceita. O item de dispersão pode ser seguido por um único parâmetro precedido por um asterisco. Esse parâmetro é chamado de *parâmetro de vetor formal*. Quando o método é chamado, o parâmetro de vetor formal é configurado de forma a referenciar um novo objeto `Array`. Todos os parâmetros reais restantes são atribuídos aos elementos do novo objeto `Array`. Se o parâmetro real correspondente ao parâmetro de vetor formal é um vetor, ele também deve ser precedido por um asterisco e deve ser o último parâmetro real.[3] Assim, Ruby permite um número variável de parâmetros, de modo similar a C#. Como os vetores de Ruby podem armazenar tipos diferentes, não há nenhuma obrigatoriedade de que os parâmetros reais passados ao vetor sejam do mesmo tipo.

O seguinte exemplo de esqueleto de definição e chamada de função ilustra a estrutura de parâmetros de Ruby:

```
list = [2, 4, 6, 8]
def tester(p1, p2, p3, *p4)
  . . .
end . . .
tester('first', mon => 72, tue => 68, wed => 59, *list)
```

Dentro de `tester`, os valores de seus parâmetros formais são os seguintes:

```
p1 is 'first'
p2 is {mon => 72, tue => 68, wed => 59}
p3 is 2
p4 is [4, 6, 8]
```

Python aceita parâmetros semelhantes aos de Ruby.

Lua usa um mecanismo simples para oferecer suporte a um número variável de parâmetros – tais parâmetros são representados por reticências (. . .). Essas reticências po-

[3]Isso não é exatamente verdade, porque o parâmetro de vetor formal pode ser seguido por uma referência de método ou de função, a qual é precedida por um e comercial (&).

dem ser tratadas como um vetor ou como uma lista de valores que podem ser atribuídos a uma lista de variáveis. Por exemplo, considere os dois exemplos de função a seguir:

```
function multiply (. . .)
  local product = 1
  for i, next  in  ipairs{. . .}  do
    product = product * next
   end return  sum
end
```

`ipairs` é um iterador para vetores (ele retorna o índice e o valor dos elementos de um vetor, um elemento por vez). `{. . .}` é um vetor dos valores de parâmetro real.

```
function DoIt (. . .)
  local a, b, c = . . .
  . . .
end
```

Suponha que `DoIt` seja chamada com o seguinte:

```
doit(4, 7, 3)
```

Nesse exemplo, `a`, `b` e `c` serão inicializados na função com os valores `4`, `7` e `3`, respectivamente.

O parâmetro de três pontos não precisa ser o único – ele pode aparecer no final de uma lista de parâmetros formais nomeados.

9.2.4 Procedimentos e funções

Existem duas categorias distintas de subprogramas – procedimentos e funções –, ambas as quais podem ser vistas como estratégias para ampliar a linguagem. Os subprogramas são conjuntos de sentenças que definem computações parametrizadas. As funções retornam valores – os procedimentos, não. Na maioria das linguagens que não incluem procedimentos como uma forma distinta de subprograma, as funções podem ser definidas de forma a não retornarem valores e serem usadas como procedimentos. As computações de um procedimento são representadas por sentenças de chamada simples. Na verdade, procedimentos definem novas sentenças. Por exemplo, se determinada linguagem não tem uma sentença de ordenação, um usuário pode criar um procedimento para ordenar vetores de dados e usar uma chamada para esse procedimento, no lugar da sentença de ordenação não disponível. Apenas linguagens mais antigas, como Fortran e Ada, suportam procedimentos.

Os procedimentos podem produzir resultados na unidade de programa chamadora por meio de dois métodos: (1) se houver variáveis que não são parâmetros formais, mas que ainda são visíveis no procedimento e na unidade de programa chamadora, o procedimento poderá alterá-las; e (2) se o procedimento tiver parâmetros formais que permitam a transferência de dados para o chamador, esses parâmetros poderão ser alterados.

Estruturalmente, as funções se assemelham aos procedimentos, mas são semanticamente modeladas como funções matemáticas. Se uma função é um modelo fiel, não produz efeitos colaterais; ou seja, não modifica seus parâmetros nem quaisquer variáveis definidas fora da função. Tal função retorna um valor – esse é seu único efeito desejado. Na maioria das linguagens de programação, as funções têm efeitos colaterais.

As funções são chamadas pela presença de seus nomes em expressões, junto com os parâmetros exigidos. O valor produzido pela execução de uma função é retornado para o código chamador, efetivamente substituindo a chamada em si. Por exemplo, o valor da expressão `f(x)` é qualquer valor produzido por `f` quando chamada com o parâmetro `x`. Para uma função que não produz efeitos colaterais, o valor retornado é seu único efeito.

As funções estabelecem novos operadores definidos pelo usuário. Por exemplo, se uma linguagem não tem um operador de exponenciação, pode ser escrita uma função que retorne o valor de um de seus parâmetros elevado à potência de outro parâmetro. Em C++, seu cabeçalho poderia ser

```
float power(float base, float exp)
```

o qual poderia ser chamado com

```
result = 3.4 * power(10.0, x)
```

A biblioteca C++ padrão já inclui uma função semelhante, chamada `pow`. Compare isso com a mesma operação em Perl, na qual a exponenciação é uma operação interna:

```
result = 3.4 * 10.0 ** x
```

Em algumas linguagens de programação, os usuários podem sobrecarregar operadores, definindo novas funções para eles. Os operadores sobrecarregados definidos pelo usuário são discutidos na Seção 9.11.

9.3 QUESTÕES DE PROJETO PARA SUBPROGRAMAS

Subprogramas são estruturas complexas e, como consequência, há uma longa lista de questões envolvidas em seu projeto. Um problema óbvio é a escolha de um ou mais métodos de passagem de parâmetros a ser usado. A ampla variedade de estratégias utilizadas em várias linguagens é um reflexo da diversidade de opiniões sobre o assunto. Uma questão intimamente relacionada é se os tipos de parâmetros serão verificados em relação aos tipos dos parâmetros formais correspondentes.

A natureza do ambiente local de um subprograma impõe, até certo ponto, a natureza do subprograma. A questão mais importante aqui é se as variáveis locais são alocadas estática ou dinamicamente.

Em seguida, existe o problema de as definições do subprograma poderem ser aninhadas. Outra questão é se nomes de subprograma podem ser passados como parâmetros. Se isso for permitido e a linguagem possibilitar que os subprogramas sejam aninhados, há a questão do ambiente de referenciamento correto de um subprograma passado como parâmetro.

Como vimos no Capítulo 5, os efeitos colaterais de funções podem causar problemas. Assim, restrições sobre os efeitos colaterais são uma questão de projeto para as funções. Os tipos e o número de valores que podem ser retornados de funções são outras questões de projeto.

Por último, há as questões de os subprogramas poderem ser sobrecarregados ou genéricos. Um **subprograma sobrecarregado** é aquele cujo nome é igual ao de outro subprograma no mesmo ambiente de referenciamento. Um **subprograma genérico** é aquele cuja computação pode ser feita em dados de diferentes tipos, em diferentes chamadas. Um **fechamento** é um subprograma aninhado e seu ambiente de referenciamento, os quais, juntos, permitem que o subprograma seja chamado a partir de qualquer lugar em um programa.

A seguir, temos um resumo dessas questões de projeto para subprogramas em geral. Mais questões, especificamente associadas às funções, são discutidas na Seção 9.10.

- As variáveis locais são alocadas estática ou dinamicamente?
- Definições de subprograma podem aparecer em outras definições de subprograma?
- Qual método (ou métodos) de passagem de parâmetros é usado?
- Os tipos dos parâmetros são verificados em relação aos tipos dos parâmetros formais?
- Se subprogramas podem ser passados como parâmetros e podem ser aninhados, qual é o ambiente de referenciamento de um subprograma passado?
- São permitidos efeitos colaterais funcionais?
- Quais tipos de valores podem ser retornados de funções?
- Quantos valores podem ser retornados de funções?
- Os subprogramas podem ser sobrecarregados?
- Os subprogramas podem ser genéricos?
- Se a linguagem permite subprogramas aninhados, fechamentos são suportados?

Essas questões e exemplos de projeto são discutidos nas seções a seguir.

9.4 AMBIENTES DE REFERENCIAMENTO LOCAL

Esta seção discute as questões relacionadas às variáveis definidas dentro de subprogramas. A questão das definições de subprograma aninhado também é abordada sucintamente.

9.4.1 Variáveis locais

Os subprogramas podem definir suas próprias variáveis, estabelecendo assim ambientes de referenciamento local. As variáveis definidas dentro de subprogramas são denominadas **variáveis locais**, pois seu escopo normalmente é o corpo do subprograma em que são definidas.

Na terminologia do Capítulo 5, variáveis locais podem ser estáticas ou dinâmicas da pilha. Se as variáveis locais são dinâmicas da pilha, elas são vinculadas ao armazenamento no início da execução do subprograma e desvinculadas quando essa execução termina. As variáveis locais dinâmicas da pilha têm diversas vantagens, e a principal delas é a flexibilidade. É fundamental que subprogramas recursivos tenham variáveis locais dinâmicas da pilha. Outra vantagem é que o armazenamento de variáveis locais em um subprograma ativo pode ser compartilhado com as variáveis locais de todos os subprogramas inativos. Essa não é uma vantagem tão importante quanto era quando os computadores tinham memórias menores.

As principais desvantagens das variáveis locais dinâmicas da pilha são as seguintes: primeiro, há o custo do tempo exigido para alocar, inicializar (quando necessário) e liberar tais variáveis para cada chamada ao subprograma. segundo, os acessos às variáveis locais dinâmicas da pilha devem ser indiretos, enquanto os acessos às variáveis estáticas podem ser diretos.[4] O acesso indireto é necessário porque o lugar na pilha onde determinada variável local residirá só pode ser determinado durante a execução (consulte o Capítulo 10). Por fim, quando todas as variáveis locais são dinâmicas da pilha, os subprogramas não podem ser sensíveis ao histórico de execução; isto é, não podem reter valores de dados de variáveis locais entre chamadas. Às vezes é conveniente escrever subprogramas sensíveis ao histórico de execução. Um exemplo comum da necessidade de um subprograma sensível ao histórico de execução é aquele cuja tarefa é gerar números pseudoaleatórios. Cada chamada a tal subprograma computa um número pseudoaleatório, usando o último que computou. Portanto, ele deve armazenar o último número em uma variável local estática. Corrotinas e os subprogramas usados em construções de laço de interação (discutidas no Capítulo 8) são outros exemplos de subprogramas que precisam ser sensíveis ao histórico de execução.

A principal vantagem das variáveis locais estáticas em relação às variáveis locais dinâmicas da pilha é o fato de serem um pouco mais eficientes – elas não apresentam nenhuma sobrecarga em tempo de execução para alocação e liberação. Além disso, se forem acessadas diretamente, esses acessos obviamente são mais eficientes. E, é claro, elas permitem que os subprogramas sejam sensíveis ao histórico de execução. A maior desvantagem das variáveis locais estáticas é sua incapacidade de suportar recursão. Além disso, seu armazenamento não pode ser compartilhado com as variáveis locais de outros subprogramas inativos.

Na maioria das linguagens contemporâneas, as variáveis locais de um subprograma são dinâmicas da pilha por padrão. Nas funções de C e C++, as variáveis locais são dinâmicas da pilha, a menos que sejam especificamente declaradas como **`static`**. Por exemplo, na função C (ou C++) a seguir, a variável `sum` é estática e `count` é dinâmica da pilha.

```
int adder(int list[], int listlen) {
  static int sum = 0;
  int count;
  for (count = 0; count < listlen; count ++)
    sum += list [count];
```

[4]Em algumas implementações, as variáveis estáticas também podem ser acessadas indiretamente, eliminando essa desvantagem.

```
  return  sum;
}
```

Os métodos de C++, Java e C# só têm variáveis locais dinâmicas da pilha.

Em Python, as únicas declarações usadas em definições de método são para variáveis globais. Qualquer variável declarada como global em um método deve ser definida fora dele. Uma variável definida fora do método pode ser referenciada nele sem ser declarada como global, mas tal variável não pode ser atribuída no método. Se o nome de uma variável global é atribuído em um método, ela é implicitamente declarada como local, e a atribuição não a perturba. Todas as variáveis locais em métodos Python são dinâmicas da pilha.

Em Lua, são declaradas somente variáveis com escopo restrito. Qualquer bloco, incluindo o corpo de uma função, pode declarar variáveis locais com a declaração **`local`**, como no seguinte:

```
local sum
```

Todas as variáveis não declaradas em Lua são globais. De acordo com Ierusalimschy (2006), em Lua, o acesso às variáveis locais é mais rápido que o acesso às variáveis globais.

9.4.2 Subprogramas aninhados

A ideia de aninhar subprogramas se originou com ALGOL 60. O objetivo era atingir a capacidade de criar uma hierarquia de lógica e escopos. Se um subprograma é necessário apenas dentro de outro subprograma, por que não colocá-lo lá e ocultá-lo do restante do programa? Como, normalmente, o escopo estático é usado em linguagens que permitem aninhar subprogramas, isso também proporciona uma maneira altamente estruturada de conceder acesso a variáveis não locais nos subprogramas envolventes. Lembre-se de que, no Capítulo 5, foram discutidos os problemas derivados disso. Por um longo tempo, as únicas linguagens que permitiam subprogramas aninhados eram as diretamente descendentes de ALGOL 60, ou seja, ALGOL 68, Pascal e Ada. Muitas outras linguagens, incluindo todas as descendentes diretas de C, não permitem aninhamento de subprogramas. Recentemente, algumas linguagens novas voltaram a permiti-lo. Entre elas estão JavaScript, Python, Ruby e Lua. Além disso, a maioria das linguagens de programação funcionais permite aninhar subprogramas.

9.5 MÉTODOS DE PASSAGEM DE PARÂMETROS

Métodos de passagem de parâmetros são as maneiras pelas quais os parâmetros são transmitidos para e/ou dos subprogramas chamados. Vamos nos concentrar primeiramente nos diferentes modelos semânticos de métodos de passagem de parâmetros. Em seguida, discutiremos os vários modelos de implementação desenvolvidos pelos projetistas de linguagem para esses modelos semânticos. Depois, examinaremos as escolhas de projeto de várias linguagens e discutiremos os métodos utilizados para aplicar os modelos de implementação. Por último, vamos ver as considerações de projeto analisadas por um projetista de linguagem na escolha entre os métodos.

9.5.1 Modelos semânticos de passagem de parâmetros

Os parâmetros formais são caracterizados por um dos três diferentes modelos semânticos a seguir: (1) podem receber dados do parâmetro real correspondente; (2) podem transmitir dados para o parâmetro real; ou (3) podem fazer as duas coisas. Esses modelos são denominados **modo de entrada**, **modo de saída** e **modo de entrada e saída**, respectivamente. Por exemplo, considere um subprograma que recebe como parâmetros dois vetores de valores `int` – `list1` e `list2`. O subprograma deve somar `list1` a `list2` e retornar o resultado como uma versão revisada de `list2`. Além disso, deve criar um vetor a partir dos dois dados e retorná-lo. Para esse subprograma, `list1` deve ser modo de entrada, pois não deve ser alterado pelo subprograma. Já `list2` deve ser modo de entrada e saída, porque o subprograma precisa do valor dado do vetor e deve retornar seu novo valor. O terceiro vetor deve ser modo de saída, pois não há nenhum valor inicial para ele e seu valor computado deve ser retornado para o chamador.

Existem dois modelos conceituais para a transferência de dados na transmissão de parâmetros: ou um valor real é copiado (no chamador, no chamado ou em ambos), ou um caminho de acesso é transmitido. Mais comumente, o caminho de acesso é um ponteiro ou uma referência simples. A Figura 9.1 ilustra os três modelos semânticos de passagem de parâmetros quando valores são copiados.

9.5.2 Modelos de implementação de passagem de parâmetros

Diversos modelos foram desenvolvidos pelos projetistas de linguagem para guiar a implementação dos três modos básicos de transmissão de parâmetros. Nas seções a seguir, discutimos vários deles, com suas vantagens e desvantagens relativas.

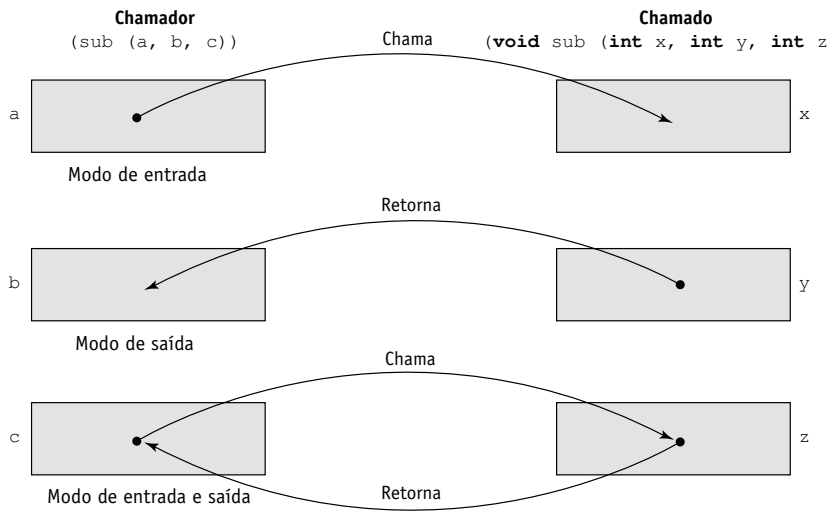


FIGURA 9.1
Os três modelos semânticos de passagem de parâmetros quando são usadas movimentações físicas.

9.5.2.1 Passagem por valor

Quando um parâmetro é **passado por valor**, o valor do parâmetro real é usado para inicializar o parâmetro formal correspondente, o qual atua então como variável local no subprograma, implementando assim a semântica modo de entrada.

Normalmente, a passagem por valor é implementada por cópia, pois muitas vezes os acessos são mais eficientes com essa estratégia. Ela poderia ser implementada por meio da transmissão de um caminho de acesso ao valor do parâmetro real no chamador, mas isso exigiria que o valor estivesse em uma célula protegida contra escrita (uma que só possa ser lida). Nem sempre é simples impor a proteção contra escrita. Por exemplo, suponha que o subprograma para o qual o parâmetro foi passado o retransmita para outro subprograma. Esse é outro motivo para se usar transferência por cópia. Conforme veremos na Seção 9.5.4, C++ fornece um método conveniente e eficaz de especificar proteção contra escrita para passar por valor parâmetros transmitidos por caminho de acesso.

A vantagem da passagem por valor é que, para escalares, ela é rápida, tanto em termos de custo de vinculação como de tempo de acesso.

A principal desvantagem do método de passagem por valor, caso sejam utilizadas cópias, é que é necessário armazenamento adicional para o parâmetro formal, ou no subprograma chamado ou em alguma área fora do chamador e do subprograma chamado. Além disso, o parâmetro real precisa ser copiado na área de armazenamento para o parâmetro formal correspondente. O armazenamento e as operações de cópia podem ser dispendiosos, caso o parâmetro seja grande, como um vetor com muitos elementos.

9.5.2.2 Passagem por resultado

Passagem por resultado é um modelo de implementação para parâmetros de modo de saída. Quando um parâmetro é passado por resultado, nenhum valor é transmitido para o subprograma. O parâmetro formal correspondente atua como uma variável local, mas, imediatamente antes que o controle seja devolvido para o chamador, seu valor é transmitido de volta para o parâmetro real do chamador, o qual obviamente deve ser uma variável. (Como o chamador referenciaria o resultado computado se fosse um literal ou uma expressão?)

O método de passagem por resultado tem as vantagens e desvantagens da passagem por valor, além de algumas desvantagens adicionais. Se os valores são retornados por cópia (e não por caminhos de acesso), como normalmente acontece, a passagem por resultado exige também o armazenamento e as operações de cópia extras necessárias para a passagem por valor. Como na passagem por valor, a dificuldade da implementação da passagem por resultado pela transmissão de um caminho de acesso normalmente resulta em sua implementação por cópia. Nesse caso, o problema é garantir que o valor inicial do parâmetro real não seja usado no subprograma chamado.

Um problema adicional no modelo de passagem por resultado é que pode haver uma colisão de parâmetro real, como a criada com a chamada

```
sub(p1, p1)
```

Em `sub`, supondo que os dois parâmetros formais tenham nomes diferentes, ambos obviamente recebem valores diferentes. Então, qualquer um dos dois que seja copiado por último em seu parâmetro real correspondente se tornará o valor de `p1` no chamador. Assim, a ordem em que os parâmetros reais são copiados determina seu valor. Por exemplo,

considere o método em C# a seguir, que especifica o método de passagem por resultado com o especificador de saída (`out`) em seu parâmetro formal.[5]

```
void Fixer(out int x, out int y) {
  x = 17;
  y = 35;
}
. . .
f.Fixer(out  a,  out  a);
```

Se, no final da execução de `Fixer`, o parâmetro formal `x` for atribuído primeiro ao seu parâmetro real correspondente, o valor do parâmetro real `a` no chamador será `35`. Se `y` for atribuído primeiro, o valor do parâmetro real `a` no chamador será `17`.

Como a ordem pode ser dependente da implementação em algumas linguagens, diferentes implementações podem produzir diferentes resultados.

Chamar um procedimento com dois parâmetros reais idênticos também pode levar a diferentes tipos de problemas quando outros métodos de passagem de parâmetro são usados, como discutido na Seção 9.5.2.4.

Outro problema que pode ocorrer na passagem por resultado é que o implementador escolha dois momentos diferentes para avaliar os endereços dos parâmetros reais: no momento da chamada ou no momento do retorno. Por exemplo, considere o seguinte método em C# e o seguinte código:

```
void DoIt(out int x, int index){
  x = 17;
  index = 42;
}
. . .
sub = 21;
f.DoIt(list[sub], sub);
```

O endereço de `list[sub]` muda entre o início e o fim do método. O implementador deve escolher o momento de vincular esse parâmetro a um endereço – no momento da chamada ou no momento do retorno. Se o endereço for computado na entrada do método, o valor `17` será retornado para `list[21]`; se for computado imediatamente antes do retorno, `17` será retornado para `list[42]`. Isso torna impossível portar programas entre uma implementação que opta por avaliar os endereços para parâmetros de modo de saída no início de um subprograma e outra que opta por fazer essa avaliação no fim. Uma maneira óbvia de evitar esse problema é o projetista da linguagem especificar quando o endereço a ser usado para retornar o valor do parâmetro deve ser computado.

9.5.2.3 Passagem por valor-resultado

Passagem por valor-resultado é um modelo de implementação para parâmetros de modo de entrada e saída no qual são copiados os valores reais. Trata-se, na verdade, de

[5] O especificador **out** também deve ser descrito no parâmetro real correspondente.

uma combinação de passagem por valor e passagem por resultado. O valor do parâmetro real é usado para inicializar o parâmetro formal correspondente, o qual atua então como uma variável local. Na verdade, os parâmetros formais da passagem por valor devem ter um armazenamento local associado ao subprograma chamado. Ao término do subprograma, o valor do parâmetro formal é transmitido de volta para o parâmetro real.

Às vezes, a passagem por valor-resultado é chamada de **passagem por cópia**, pois o parâmetro real é copiado no parâmetro formal na entrada do subprograma e, então, copiado de volta no término do subprograma.

A passagem por valor-resultado compartilha com a passagem por valor e com a passagem por resultado as desvantagens de exigir armazenamento múltiplo para parâmetros e o tempo para copiar os valores. Com a passagem por resultado, ela compartilha os problemas associados à ordem em que os parâmetros reais são atribuídos.

As vantagens da passagem por valor-resultado são relacionadas à passagem por referência; portanto, estão discutidas na Seção 9.5.2.4.

9.5.2.4 Passagem por referência

Passagem por referência é um segundo modelo de implementação para parâmetros de modo de entrada e saída. Contudo, em vez de copiar valores de dados, como na passagem por valor-resultado, o método de passagem por referência transmite um caminho de acesso (normalmente apenas um endereço) para o subprograma chamado. Isso fornece o caminho de acesso para a célula que armazena o parâmetro real. Assim, o subprograma chamado pode acessar o parâmetro real na unidade de programa chamadora. Com efeito, o parâmetro real é compartilhado com o subprograma chamado.

A vantagem da passagem por referência é que o processo de passagem em si é eficiente tanto em termos de tempo como de espaço. Não é exigido espaço duplicado e nenhuma cópia é necessária.

No entanto, existem várias desvantagens. Primeiro, o acesso aos parâmetros formais será mais lento que aos parâmetros da passagem por valor, devido ao nível adicional de endereçamento indireto exigido.[6] Segundo, se é necessária apenas comunicação unilateral com o subprograma chamado, podem ser feitas alterações acidentais e erradas no parâmetro real.

Outro problema da passagem por referência é que apelidos podem ser criados. Esse problema deve ser esperado, pois a passagem por referência torna caminhos de acesso disponíveis para os subprogramas chamados, fornecendo assim acesso a variáveis não locais. O problema desses tipos de criação de apelidos é o mesmo de outras circunstâncias: é prejudicial à legibilidade e, portanto, à confiabilidade. Isso também torna a verificação de programas mais difícil.

Os parâmetros da passagem por referência podem criar apelidos de várias maneiras. Primeiro, podem ocorrer colisões entre os parâmetros reais. Considere uma função em C++ que tem dois parâmetros a serem passados por referência, como em

```
void fun(int &first, int &second)
```

[6] Isso é discutido com mais detalhes na Seção 9.5.3.

Se a chamada a `fun` passar a mesma variável duas vezes, como em

```
fun(total, total)
```

então `first` e `second` em `fun` serão apelidos.

Segundo, colisões entre elementos de vetor também podem criar apelidos. Por exemplo, suponha que a função `fun` seja chamada com dois elementos de vetor especificados com índices de variável, como em

```
fun(list[i], list[j])
```

Se esses dois parâmetros são passados por referência e `i` é igual a `j`, então `first` e `second` são apelidos novamente.

Terceiro, se dois dos parâmetros formais de um subprograma são elementos de um vetor e do vetor inteiro, e ambos são passados por referência, então uma chamada como

```
fun1(list[i], list)
```

poderia resultar na criação de apelidos em `fun1`, pois `fun1` pode acessar todos os elementos de `list` por meio do segundo parâmetro e um único elemento por meio de seu primeiro parâmetro.

Outra maneira de criar apelidos com parâmetros passados por referência é por meio de colisões entre parâmetros formais e variáveis não locais visíveis. Por exemplo, considere o código C a seguir:

```
int * global;
void main() {
   . . .
   sub(global);
   . . .
}
 void  sub(int  * param) {
   . . .
}
```

Dentro de `sub`, `param` e `global` são apelidos.

Todas essas possíveis situações de criação de apelidos são eliminadas se é usada passagem por valor em vez de passagem por referência. Contudo, além da criação de apelidos, às vezes surgem outros problemas, como discutido na Seção 9.5.2.3.

9.5.2.5 Passagem por nome

Passagem por nome é um método de transmissão de parâmetros de modo de entrada e saída que não corresponde a um único modelo de implementação. Quando parâmetros são passados por nome, na verdade o parâmetro real é substituído textualmente pelo parâmetro formal correspondente em todas as suas ocorrências no subprograma. Esse método é bem diferente dos discutidos até aqui, em que os parâmetros formais são vinculados a valores reais ou a endereços no momento da chamada do subprograma. Um parâmetro formal passado por nome é vinculado a um método de acesso no momento da chamada do subprograma, mas a vinculação real a um valor ou a um endereço é adiada até que o parâmetro formal seja atribuído ou referenciado. A implementação de um pa-

râmetro passado por nome exige que um subprograma seja passado para o subprograma chamado a fim de avaliar o endereço ou valor do parâmetro formal. O ambiente de referenciamento do subprograma passado também deve ser passado. Esse subprograma/ambiente de referenciamento é um fechamento (consulte a Seção 9.12).[7] Os parâmetros passados por nome têm implementação complexa e são ineficientes. Além disso, aumentam significativamente a complexidade do programa, diminuindo com isso sua legibilidade e sua confiabilidade.

Como a passagem por nome não faz parte de nenhuma linguagem amplamente utilizada, não é discutida mais a fundo aqui. No entanto, ela é usada em tempo de compilação pelas macros nas linguagens assembly e para os parâmetros genéricos dos subprogramas genéricos em C++, Java 5.0 e C# 2005, conforme discutido na Seção 9.9.

9.5.3 Implementação de métodos de passagem de parâmetros

Analisaremos agora como os vários modelos de implementação de passagem de parâmetros são realmente implementados.

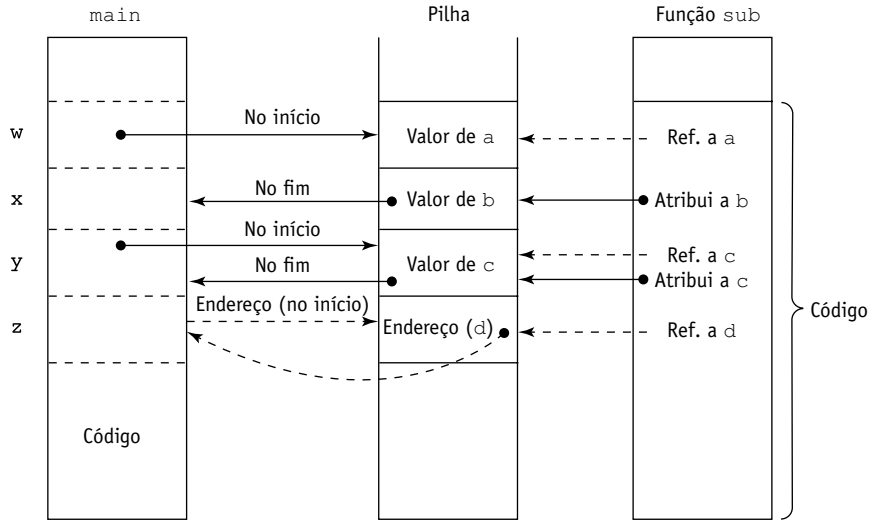
Na maioria das linguagens contemporâneas, a comunicação de parâmetros ocorre por intermédio da pilha de tempo de execução. Ela é inicializada e mantida pelo sistema de tempo de execução, o qual gerencia a execução de programas. A pilha de tempo de execução é usada extensivamente para vinculação de controle de subprogramas e passagem de parâmetros, conforme discutido no Capítulo 10. Na discussão a seguir, presumimos que a pilha é usada para toda transmissão de parâmetros.

Os parâmetros passados por valor têm seus valores copiados em locais da pilha. Os locais da pilha servem, então, como armazenamento para os parâmetros formais correspondentes. Os parâmetros passados por resultado são implementados como o oposto da passagem por valor. Os valores atribuídos aos parâmetros reais passados por resultado são colocados na pilha, de onde podem ser recuperados pela unidade de programa chamadora, ao término do subprograma chamado. Os parâmetros passados por valor-resultado podem ser implementados diretamente a partir de sua semântica, como uma combinação de passagem por valor e passagem por resultado. O local da pilha para tal parâmetro é inicializado pela chamada e, então, é usado como uma variável local no subprograma chamado.

Talvez os parâmetros passados por referência sejam os mais simples de implementar. Independentemente do tipo do parâmetro real, apenas seu endereço deve ser colocado na pilha. No caso de literais, o endereço do literal é colocado na pilha. No caso de uma expressão, o compilador deve gerar código para avaliá-la, o qual deve ser executado imediatamente antes da transferência de controle para o subprograma chamado. Então, o endereço da célula de memória na qual o código coloca o resultado de sua avaliação é colocado na pilha. O compilador deve garantir que o subprograma chamado não altere parâmetros que são literais ou expressões.

O acesso aos parâmetros formais no subprograma chamado é feito por endereçamento indireto do local do endereço na pilha. A implementação de passagem por valor, por resultado, por valor-resultado e por referência, em que é usada a pilha de tempo de execução, é mostrada na Figura 9.2. O subprograma `sub` é chamado a partir de `main`

[7] Esses fechamentos eram chamados originalmente de *thunks* (em ALGOL 60).



Cabeçalho da função: **void** sub (**int** a, **int** b, **int** c, **int** d)
Chamada da função em main: sub (w,x,y,z)
(passa w por valor, x por resultado, y por valor-resultado, z por referência)

FIGURA 9.2
Uma possível implementação de pilha dos métodos de passagem de parâmetros comuns.

com a chamada `sub(w, x, y, z)`, onde `w` é passado por valor, `x` é passado por resultado, `y` é passado por valor-resultado e `z` é passado por referência.

9.5.4 Métodos de passagem de parâmetros de algumas linguagens comuns

C usa passagem por valor. A semântica de passagem por referência (modo de entrada e saída) é obtida pelo uso de ponteiros como parâmetros. O valor do ponteiro se torna disponível para a função chamada e nada é copiado de volta. Contudo, como o que foi passado é um caminho de acesso aos dados do chamador, a função chamada pode alterar esses dados. C copiou esse uso do método de passagem por valor de ALGOL 68. Tanto em C como em C++, os parâmetros formais podem ser tipados como ponteiros para constantes. Os parâmetros reais correspondentes não precisam ser constantes, pois nesses casos sofrem coerção para constantes. Isso permite que parâmetros de ponteiro ofereçam a eficiência da passagem por referência com a semântica unilateral da passagem por valor. A proteção contra escrita desses parâmetros na função chamada é especificada implicitamente.

C++ inclui um tipo de ponteiro especial, chamado *tipo de referência* (discutido no Capítulo 6), que é frequentemente usado para parâmetros. Os parâmetros de referência são desreferenciados implicitamente na função ou no método, e sua semântica é a passagem por referência. C++ também permite que parâmetros de referência sejam definidos como constantes. Por exemplo, poderíamos ter

```
void fun(const int &p1, int p2, int &p3) { . . . }
```

» *nota histórica*

ALGOL 60 introduziu o método de passagem por nome. A linguagem também permitia passagem por valor como opção. Principalmente devido à dificuldade em sua implementação, os parâmetros passados por nome não foram levados de ALGOL 60 para nenhuma das linguagens subsequentes que se tornaram populares (fora SIMULA 67).

» *nota histórica*

ALGOL W (Wirth e Hoare, 1966) introduziu o método de passagem por valor-resultado de passagem de parâmetros como uma alternativa à ineficiência da passagem por nome e aos problemas da passagem por referência.

onde `p1` é passado por referência, mas não pode ser alterado na função `fun`, `p2` é passado por valor e `p3` é passado por referência. Nem `p1` nem `p3` precisam ser desreferenciados explicitamente em `fun`.

Parâmetros constantes e parâmetros de modo de entrada não são exatamente iguais. Os parâmetros constantes claramente implementam o modo de entrada. No entanto, em todas as linguagens imperativas comuns, exceto Ada, os parâmetros de modo de entrada podem ser atribuídos no subprograma, mesmo que essas alterações nunca sejam refletidas nos valores dos parâmetros reais correspondentes. Os parâmetros constantes nunca podem ser atribuídos.

Como em C e C++, todos os parâmetros Java são passados por valor. Contudo, como os objetos só podem ser acessados por meio de variáveis de referência, os parâmetros de objeto são, na verdade, passados por referência. Embora uma referência de objeto passada como parâmetro não possa ser alterada no subprograma chamado, o objeto referenciado pode, se estiver disponível um método para causar a alteração. Como variáveis de referência não podem apontar para variáveis escalares diretamente e Java não tem ponteiros, escalares não podem ser passadas por referência em Java (embora uma referência a um objeto que contenha uma escalar possa).

O método de passagem de parâmetros padrão de C# é por valor. A passagem por referência pode ser especificada precedendo-se um parâmetro formal e seu parâmetro real correspondente com **`ref`**. Por exemplo, considere o seguinte esqueleto de método e chamada em C#:

```
void sumer(ref int oldSum, int newOne) { . . . }
. . .
sumer(ref sum, newValue);
```

O primeiro parâmetro para `sumer` é passado por referência; o segundo é passado por valor.

C# também suporta parâmetros de modo de saída, os quais são parâmetros passados por referência que não precisam de valores iniciais. Tais parâmetros são especificados na lista de parâmetros formais com o modificador de saída (**`out`**).

A passagem de parâmetros de PHP é semelhante à de C#, exceto que o parâmetro real ou o parâmetro formal pode especificar passagem por referência. A passagem por referência é especificada precedendo-se um ou ambos os parâmetros por um e comercial.

Perl emprega um modo primitivo de passagem de parâmetros. Todos os parâmetros reais são implicitamente colocados em um vetor predefinido, chamado `@_` (por incrível que pareça!). O subprograma recupera os valores (ou endereços) do parâmetro real desse vetor. O detalhe mais curioso sobre esse vetor é sua natureza mágica, que fica clara pelo fato de que seus elementos são, na verdade, apelidos para os parâmetros reais. Portanto,

se um elemento de `@_` é alterado no subprograma chamado, essa alteração é refletida no parâmetro real correspondente na chamada, supondo que haja um (o número de parâmetros reais não precisa ser igual ao número de parâmetros formais) e que ele seja uma variável.

O método de passagem de parâmetros de Python e Ruby é chamado **passagem por atribuição**. Como todos os valores de dados são objetos, toda variável é uma referência para um objeto. Na passagem por atribuição, o valor do parâmetro real é atribuído ao parâmetro formal. Portanto, a passagem por atribuição é, na verdade, uma passagem por referência, pois os valores de todos os parâmetros reais são referências. Entretanto, apenas em certos casos isso resulta na semântica de passagem por referência. Por exemplo, muitos objetos são basicamente imutáveis. Em uma linguagem orientada a objetos pura, o processo de alteração do valor de uma variável com uma sentença de atribuição, como em

```
x = x + 1
```

não altera o objeto referenciado por `x`. Em vez disso, ele o incrementa por `1`, criando assim outro objeto (com o valor `x + 1`), e altera `x` para referenciar o novo objeto. Portanto, quando uma referência a um objeto escalar é passada para um subprograma, o objeto referenciado não pode ser alterado no local. Como a referência é passada por valor, mesmo que o parâmetro formal seja alterado no subprograma, essa alteração não tem efeito no parâmetro real no chamador.

Agora, suponha que seja passada como parâmetro uma referência para um vetor. Se o parâmetro formal correspondente recebe um novo objeto vetor, não há nenhum efeito no chamador. Contudo, se o parâmetro formal é usado para atribuir um valor a um elemento do vetor, como em

```
list[3] = 47
```

o parâmetro real é afetado. Portanto, alterar a referência do parâmetro formal não tem nenhum efeito no chamador, mas alterar um elemento do vetor passado como parâmetro tem.

9.5.5 Verificação de tipos de parâmetros

Agora é amplamente aceito que a confiabilidade do software exija que seja verificada a consistência dos tipos dos parâmetros reais com os tipos dos parâmetros formais correspondentes. Sem tal verificação de tipos, pequenos erros tipográficos podem levar a erros de programa que podem ser difíceis de diagnosticar, pois não são detectados pelo compilador nem pelo sistema de tempo de execução. Por exemplo, na chamada de função

```
result = sub1(1)
```

o parâmetro real é uma constante inteira. Se o parâmetro formal de `sub1` for um tipo de ponto flutuante, nenhum erro será detectado sem a verificação de tipos do parâmetro. Embora um 1 inteiro e um 1 de ponto flutuante tenham o mesmo valor, as representações

desses dois são muito diferentes. `sub1` não pode produzir um resultado correto, dado um valor de parâmetro real inteiro, se espera um valor de ponto flutuante.

As primeiras linguagens de programação, como Fortran 77 e a versão original de C, não exigiam verificação de tipos de parâmetros; a maioria das linguagens posteriores exige. Contudo, as linguagens relativamente recentes Perl, JavaScript e PHP não exigem.

C e C++ exigem uma análise mais aprofundada sobre a questão da verificação de tipos de parâmetros. Em C original, nem o número de parâmetros nem seus tipos eram verificados. Em C89, os parâmetros formais de funções podem ser definidos de duas maneiras. Eles podem ser definidos como na linguagem C original; ou seja, os nomes dos parâmetros são listados nos parênteses e as declarações de tipo para eles vêm em seguida, como na função a seguir:

```
double sin(x)
  double x;
  { . . . }
```

O uso dessa forma evita a verificação de tipos, permitindo assim que chamadas como

```
double value;
int count;
. . .
value = sin(count);
```

sejam válidas, embora nunca sejam corretas.

A alternativa à estratégia de definição da linguagem C original é chamada de método do **protótipo**, no qual os tipos do parâmetro formal são incluídos na lista, como em

```
double sin(double x)
 { . . . }
```

Se essa versão de `sin` é invocada com a mesma chamada, isto é, com o seguinte, isso também é válido:

```
value = sin(count);
```

O tipo do parâmetro real (**`int`**) é verificado em relação ao do parâmetro formal (**`double`**). Embora não casem, **`int`** pode sofrer coerção para **`double`** (é uma coerção de alargamento), de modo que a conversão é feita. Se a conversão não é possível (por exemplo, se o parâmetro real tiver sido um vetor), ou se o número de parâmetros está errado, então é detectado um erro semântico. Assim, em C89, o usuário escolhe se os parâmetros terão o tipo verificado.

Em C99 e C++, todas as funções devem ter seus parâmetros formais na forma de protótipo. Contudo, a verificação de tipos pode ser evitada para alguns dos parâmetros, substituindo-se a última parte da lista de parâmetro por reticências, como em

```
int printf(const char* format_string, . . .);
```

Uma chamada para `printf` deve incluir pelo menos um parâmetro, um ponteiro para uma cadeia de caracteres literais. Fora isso, tudo (incluindo nada) é válido. `printf` de-

termina se existem mais parâmetros pela presença de códigos de formatação no parâmetro da cadeia. Por exemplo, o código de formatação para saída inteira é `%d`. Isso aparece como parte da cadeia, como no seguinte:

```
printf("The sum is %d\n", sum);
```

O `%` indica à função `printf` que há mais um parâmetro.

Outra questão interessante nas coerções de parâmetro real para formal surge quando primitivas podem ser passadas por referência, como em C#. Suponha que uma chamada para um método passe um valor **`float`** para um parâmetro formal **`double`**. Se esse parâmetro é passado por valor, o valor **`float`** sofre coerção para **`double`** e não há problema. Essa coerção em particular é muito útil, pois permite que uma biblioteca forneça versões double de subprogramas que podem ser usadas para valores **`float`** e **`double`**. No entanto, suponha que o parâmetro seja passado por referência. Quando o valor do parâmetro formal **`double`** for retornado para o parâmetro real **`float`** no chamador, o valor transbordará sua localização. Para evitar esse problema, C# exige que o tipo de um parâmetro real **`ref`** case exatamente com o tipo de seu parâmetro formal correspondente (nenhuma coerção é permitida).

Em Python e Ruby não há verificação de tipos de parâmetros, pois nessas linguagens a tipagem é um conceito diferente. Objetos têm tipos, mas variáveis não; portanto, os parâmetros formais são desprovidos de tipos. Isso invalida a própria ideia de verificação de tipos de parâmetros.

9.5.6 Matrizes multidimensionais como parâmetros

As funções de mapeamento de armazenamento utilizadas para mapear valores de índice de referências a elementos de matrizes multidimensionais para endereços na memória foram discutidas detalhadamente no Capítulo 6. Em algumas linguagens, como C e C++, quando uma matriz multidimensional é passada como parâmetro para um subprograma, o compilador precisa criar a função de mapeamento para essa matriz enquanto vê apenas o texto do subprograma (não o subprograma chamador). Isso é verdade porque os subprogramas podem ser compilados separadamente dos programas que os chamam. Considere o problema da passagem de uma matriz para uma função em C. As matrizes multidimensionais em C são, na verdade, matrizes de matrizes, e são armazenadas na ordem principal de linha. A seguir está uma função de mapeamento de armazenamento de ordem principal de linha para matrizes, quando o limite inferior de todos os índices é 0 e o tamanho do elemento é 1:

```
endereço(mat[i, j]) = endereço(mat[0,0]) + i *
                      número_de_colunas + j
```

Observe que essa função de mapeamento precisa do número de colunas, mas não do número de linhas. Portanto, em C e C++, quando uma matriz é passada como parâmetro, o parâmetro formal deve incluir o número de colunas no segundo par de colchetes. Isso está ilustrado no seguinte esqueleto de programa em C:

```
void fun(int matrix[][10]) {
 . . . }
```

```
void main() {
  int mat[5][10];
  . . .
  fun(mat);
  . . .
}
```

O problema desse método de passagem de matrizes como parâmetros é que ele não permite que o programador escreva uma função que aceite matrizes com números diferentes de colunas; uma nova função deve ser escrita para cada matriz com um número de colunas diferente. Se as funções lidam com matrizes multidimensionais, isso, na verdade, impossibilita a escrita de funções flexíveis que possam ser reutilizadas de maneira efetiva. Em C e C++, há um modo de contornar o problema, devido à inclusão de aritmética de ponteiros. A matriz pode ser passada como um ponteiro e suas dimensões também podem ser passadas como parâmetros. Então, a função pode avaliar a função de mapeamento de armazenamento escrita pelo usuário, utilizando aritmética de ponteiros sempre que um elemento da matriz precisar ser referenciado. Por exemplo, considere o protótipo de função a seguir:

```
void fun(float *mat_ptr,
         int num_rows,
         int num_cols);
```

A sentença a seguir pode ser usada para mover o valor da variável `x` para o elemento `[row][col]` da matriz de parâmetros em `fun`:

```
*(mat_ptr + (row * num_cols) + col) = x;
```

Embora isso funcione, obviamente é difícil de ler e, devido à sua complexidade, é propenso a erro. A dificuldade para ler pode ser atenuada com o uso de uma macro para definir a função de mapeamento de armazenamento, como segue

```
#define mat_ptr(r,c) (*mat_ptr + ((r) *
                     (num_cols) + (c)))
```

Com isso, a sentença pode ser escrita como

```
mat_ptr(row,col) = x;
```

Outras linguagens usam estratégias diferentes para lidar com o problema da passagem de matrizes multidimensionais.

Em Java e C#, matrizes são objetos. Todos têm uma dimensão, mas os elementos podem ser matrizes. Cada matriz herda uma constante nomeada (`length` em Java e `Length` em C#), configurada com o tamanho da matriz quando o objeto matriz é criado. O parâmetro formal de uma matriz aparece com dois conjuntos de colchetes vazios, como no método Java a seguir:

```
float sumer(float mat[][]) {
   float sum = 0.0f;
   for (int row = 0; row < mat.length; row++) {
     for (int col = 0; col < mat[row].length; col++) {
```

```
        sum += mat[row][col];
      }  //** for (int row . . .
    }  //** for (int col . . .
    return  sum;
}
```

Como cada matriz tem seu próprio valor de tamanho, em uma matriz as linhas podem ter comprimentos diferentes.

9.5.7 Considerações de projeto

Duas considerações importantes estão envolvidas na escolha dos métodos de passagem de parâmetros: eficiência e se é necessária transferência de dados unidirecional ou bidirecional.

Os princípios de engenharia de software contemporâneos prescrevem que o acesso por código de subprograma a dados fora do subprograma deve ser minimizado. Com esse objetivo em mente, os parâmetros de modo de entrada devem ser usados sempre que nenhum dado for retornado ao chamador por meio de parâmetros. Os parâmetros de modo de saída devem ser usados quando nenhum dado for transferido para o subprograma chamado, mas o subprograma precisar transmitir dados de volta para o chamador. Por fim, os parâmetros de modo de entrada e saída devem ser usados apenas quando dados precisarem se mover nas duas direções, entre o chamador e o subprograma chamado.

Há uma consideração prática que está em conflito com esse princípio. Às vezes, é justificável passar caminhos de acesso para transmissão unilateral de parâmetros. Por exemplo, quando uma matriz grande é passada para um subprograma que não a modifica, talvez seja preferível um método unilateral. Contudo, a passagem por valor exigiria que a matriz inteira fosse movida para uma área de armazenamento local do subprograma. Isso seria dispendioso tanto em termos de tempo quanto de espaço. Por isso, matrizes grandes frequentemente são passadas por referência. Esse é precisamente o motivo pelo qual a definição de Ada 83 permitia aos implementadores escolher entre os dois métodos de parâmetros estruturados. Os parâmetros de referência à constante de C++ oferecem outra solução. Uma estratégia alternativa seria permitir que o usuário escolhesse entre os métodos.

A escolha de um método de passagem de parâmetros para funções está relacionada a outra questão de projeto: os efeitos colaterais funcionais. Essa questão é discutida na Seção 9.10.

9.5.8 Exemplos de passagem de parâmetros

Considere a função C a seguir:

```
void swap1(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}
```

Suponha que essa função seja chamada com

```
swap1(c, d);
```

Lembre-se de que C usa passagem por valor. As ações de `swap1` podem ser descritas pelo pseudocódigo a seguir:

```
a = c    - Traz o valor do primeiro parâmetro
b = d    - Traz o valor do segundo parâmetro
temp = a
a = b
b = temp
```

Embora `a` acabe com o valor de `d` e `b` acabe com o valor de `c`, os valores de `c` e `d` ficam intactos, pois nada é transmitido de volta para o chamador.

Podemos modificar a função `swap` de C para lidar com parâmetros de ponteiro, a fim de obter o efeito de passagem por referência:

```
void swap2(int *a, int *b) {
 int temp = *a;
 *a = *b;
 *b = temp;
}
```

`swap2` pode ser chamado com

```
swap2(&c, &d);
```

As ações de `swap2` podem ser descritas com o seguinte:

```
a = &c      - Traz o endereço do primeiro parâmetro
b = &d      - Traz o endereço do segundo parâmetro
temp = *a
*a = *b
*b = temp
```

Nesse caso, a operação de troca é bem-sucedida: os valores de `c` e `d` são realmente trocados. `swap2` pode ser escrita em C++ com parâmetros de referência, como segue:

```
void swap2(int &a, int &b) {
   int temp = a;
  a = b;
  b = temp;
}
```

Essa operação de troca simples não é possível em Java, pois essa linguagem não tem ponteiros nem o tipo de referências de C++. Em Java, uma variável de referência pode apontar apenas para um objeto, não para um valor escalar.

A semântica da passagem por valor-resultado é idêntica à da passagem por referência, exceto quando está envolvido o uso de apelidos. Ada usa passagem por valor-resultado para parâmetros escalares do modo de entrada e saída. Para explorar a passa-

gem por valor-resultado, considere a função `swap3` a seguir, a qual presumimos que usa parâmetros passados por valor-resultado. Ela está escrita em uma sintaxe semelhante à de Ada.

```
procedure swap3(a : in out Integer, b : in out Integer) is
  temp : Integer;
   begin temp := a;
   a := b;
   b := temp;
   end  swap3;
```

Suponha que `swap3` seja chamada com

```
swap3(c, d);
```

As ações de `swap3` com essa chamada são

```
addr_c = &c       - Traz o endereço do primeiro parâmetro
addr_d = &d       - Traz o endereço do segundo parâmetro
a = *addr_c       - Traz o valor do primeiro parâmetro
b = *addr_d       - Traz o valor do segundo parâmetro
temp = a
a = b
b = temp
*addr_c = a       - Leva o valor do primeiro parâmetro
*addr_d = b       - Leva o valor do segundo parâmetro
```

Então, mais uma vez, esse subprograma de troca funciona corretamente. A seguir, considere a chamada

```
swap3(i, list[i]);
```

Nesse caso, as ações são

```
addr_i = &i           - Traz o endereço do primeiro parâmetro
addr_listi= &list[i]  - Traz o endereço do segundo parâmetro
a = *addr_i           - Traz o valor do primeiro parâmetro
b = *addr_listi       - Traz o valor do segundo parâmetro
temp = a
a = b
b = temp
*addr_i = a           - Leva o valor do primeiro parâmetro
*addr_listi = b       - Leva o valor do segundo parâmetro
```

Novamente, o subprograma funciona corretamente, nesse caso porque os endereços para os quais os valores dos parâmetros são retornados são computados no momento da chamada, não no momento do retorno. Se os endereços dos parâmetros reais fossem computados no momento do retorno, os resultados seriam errados.

Por fim, precisamos explorar o que acontece quando a criação de apelidos está envolvida na passagem por valor-resultado e na passagem por referência. Considere o esqueleto de programa a seguir, escrito em sintaxe do tipo C:

```
int i = 3; /* i é uma variável global */
void fun(int a, int b) {
  i = b;
}
void main() {
  int list[10];
  list[i] = 5;
  fun(i, list[i]);
}
```

Em `fun`, se for usada passagem por referência, `i` e `a` serão apelidos. Se for usada passagem por valor-resultado, `i` e `a` não serão apelidos. As ações de `fun`, supondo passagem por valor-resultado, são as seguintes:

```
addr_i = &i            - Traz o endereço do primeiro parâmetro
addr_listi = &list[i]  - Traz o endereço do segundo parâmetro
a = *addr_i            - Traz o valor do primeiro parâmetro
b = *addr_listi        - Traz o valor do segundo parâmetro
i = b                  - Configura i como 5
*addr_i = a            - Leva o valor do primeiro parâmetro
*addr_listi = b        - Leva o valor do segundo parâmetro
```

Nesse caso, a atribuição à global `i` em `fun` altera seu valor de `3` para `5`, mas a cópia de volta do primeiro parâmetro formal (a penúltima linha no exemplo) a configura novamente como `3`. A observação mais importante aqui é que, se for usada passagem por referência, a cópia de volta não fará parte da semântica e `i` permanecerá `5`. Observe também que, como o endereço do segundo parâmetro é computado no início de `fun`, qualquer alteração na global `i` não tem qualquer efeito no endereço usado no final para retornar o valor de `list[i]`.

9.6 PARÂMETROS QUE SÃO SUBPROGRAMAS

Em programação ocorrem várias situações que são tratadas mais convenientemente se os nomes de subprograma puderem ser enviados como parâmetros para outros subprogramas. Um exemplo comum ocorre quando um subprograma precisa amostrar alguma função matemática. Por exemplo, um subprograma que faz integração numérica calcula a área sob o grafo de uma função amostrando a função em vários pontos diferentes. Quando tal subprograma é escrito, deve ser utilizável por qualquer função dada; não deve ser necessário reescrevê-lo para cada função que precise ser integrada. Portanto, é natural que o nome de uma função de programa que avalia a função matemática a ser integrada seja enviado como parâmetro para o subprograma de integração.

Embora essa ideia seja aparentemente simples, os detalhes de como ela funciona podem ser confusos. Se fosse necessária apenas a transmissão do código do subprograma, ela poderia ser feita passando-se um único ponteiro. Contudo, surgem duas complicações.

Primeiro, há a questão da verificação de tipos dos parâmetros das ativações do subprograma passado como parâmetro. Em C e C++, funções não podem ser passadas como parâmetros, mas ponteiros para funções podem. O tipo de um ponteiro para uma função inclui o protocolo da função. Como o protocolo inclui todos os tipos de parâmetro, tais parâmetros podem ser completamente verificados quanto ao tipo.

A segunda complicação com parâmetros que são subprogramas só ocorre em linguagens que permitem subprogramas aninhados. A questão é qual ambiente de referenciamento deve ser usado para executar o subprograma passado. São três escolhas:

- O ambiente da sentença de chamada que executa o subprograma passado (**vinculação rasa** – *shallow binding*)
- O ambiente da definição do subprograma passado (**vinculação profunda** – *deep binding*)
- O ambiente da sentença de chamada que passou o subprograma como parâmetro real (**vinculação** ad hoc – *ad hoc binding*)

O exemplo de programa a seguir, escrito com a sintaxe do JavaScript, ilustra essas escolhas:

```
function sub1() {
  var x;
  function sub2() {
    alert(x);  // Cria uma caixa de diálogo com o valor de x
    };
  function  sub3() {
    var  x;
    x = 3;
    sub4(sub2);
    };
  function  sub4(subx) {
    var  x;
    x = 4;
    subx();
    };
  x = 1;
  sub3();
  };
```

Considere a execução de `sub2` quando é chamado em `sub4`. Para vinculação rasa, o ambiente de referenciamento dessa execução é o de `sub4`; portanto, a referência a `x` em `sub2` é vinculada à variável local `x` em `sub4`, e a saída do programa é `4`. Para vinculação profunda, o ambiente de referenciamento da execução de `sub2` é o de `sub1`; portanto, a referência a `x` em `sub2` é vinculada à variável local `x` em `sub1`, e a saída é `1`. Para ad hoc, a vinculação é com a variável local `x` em `sub3`, e a saída é `3`.

» *nota histórica*

A definição original de Pascal (Jensen e Wirth, 1974) permitia passar subprogramas como parâmetros sem incluir suas informações de tipo de parâmetro. Se a compilação independente fosse possível (não era possível no Pascal original), o compilador nem mesmo permitiria a verificação do número correto de parâmetros. Na ausência de compilação independente, é possível verificar a consistência de parâmetros, mas essa é uma tarefa muito complexa, normalmente não executada.

Em alguns casos, o subprograma que declara um subprograma também passa esse subprograma como parâmetro. Nesses casos, a vinculação profunda e a ad hoc são iguais. A vinculação ad hoc não tem sido usada porque, como é possível supor, o ambiente no qual o procedimento aparece como parâmetro não tem nenhuma conexão natural com o subprograma passado.

A vinculação rasa não é adequada para linguagens de escopo estático com subprogramas aninhados. Por exemplo, suponha que o procedimento `Sender` passe o procedimento `Sent` como parâmetro para o procedimento `Receiver`. O problema é que `Receiver` pode não estar no ambiente estático de `Sent`, tornando, com isso, altamente não natural `Sent` ter acesso às variáveis de `Receiver`. Por outro lado, em uma linguagem assim, é perfeitamente normal qualquer subprograma, incluindo um enviado como parâmetro, ter seu ambiente de referenciamento determinado pela posição léxica de sua definição. Portanto, é mais lógico essas linguagens usarem vinculação profunda. Algumas linguagens de escopo dinâmico usam vinculação rasa.

9.7 CHAMADA INDIRETA DE SUBPROGRAMAS

Existem situações nas quais os subprogramas devem ser chamados indiretamente. Elas ocorrem com mais frequência quando o subprograma específico a ser chamado não é conhecido até o momento da execução. A chamada ao subprograma é feita por meio de um ponteiro ou de uma referência para o subprograma, configurado(a) durante a execução, antes de a chamada ser feita. As duas aplicações mais comuns de chamadas indiretas de subprograma são o tratamento de evento em interfaces gráficas do usuário, que agora fazem parte de praticamente todas as aplicações Web, assim como de muitas que não são para a Web; e os *callbacks*, nos quais um subprograma é chamado e instruído a notificar o chamador quando tiver concluído seu trabalho. Como sempre, nosso interesse não está nesses tipos específicos de programação, mas no suporte de linguagem de programação para eles.

O conceito de chamadas indiretas a subprogramas não foi desenvolvido recentemente. C e C++ permitem que um programa defina um ponteiro para uma função, por meio do qual ela pode ser chamada. Em C++, ponteiros para funções são tipados de acordo com o tipo de retorno e os tipos de parâmetros da função; portanto, tal ponteiro só pode apontar para funções com um protocolo específico. Por exemplo, a declaração a seguir define um ponteiro (`pfun`) que pode apontar para qualquer função que receba um **`float`** e um **`int`** como parâmetros e retorne um **`float`**:

```
float (*pfun)(float, int);
```

Qualquer função com o mesmo protocolo desse ponteiro pode ser usada como valor inicial do ponteiro ou ser atribuída a ele em um programa. Em C e C++, um nome de

função sem parênteses depois, como um nome de vetor sem colchetes, é o endereço da função (ou vetor). Assim, os dois exemplos a seguir são maneiras válidas de fornecer um valor inicial ou atribuir um valor a um ponteiro em uma função:

```
int myfun2 (int, int);  // Uma declaração de função
int (*pfun2)(int, int) = myfun2;  // Cria um ponteiro e
                                  // o inicializa
                                  // de modo a apontar para myfun2
pfun2 = myfun2;  // Atribui o endereço de uma função a um
                 // ponteiro
```

Agora a função `myfun2` pode ser chamada com uma das sentenças a seguir:

```
(*pfun2)(first, second); pfun2(first, second);
```

A primeira delas desreferencia explicitamente o ponteiro `pfun2`, o que é válido, mas desnecessário.

Os ponteiros de função de C e C++ podem ser enviados como parâmetros e retornados de funções, embora funções não possam ser usadas diretamente em nenhuma dessas tarefas.

Em C#, o poder e a flexibilidade dos ponteiros de método são ampliados pelo fato de eles serem tornados objetos. Eles são chamados **representantes**, porque, em vez de chamar um método, um programa delega essa ação para um representante.

Para usar um representante, primeiramente a classe representante (delegate) deve ser definida com um protocolo de método específico. A instanciação de um representante armazena o nome de um método com o protocolo do representante que é capaz de chamar. A sintaxe de uma declaração de representante é igual à de uma declaração de método, exceto que a palavra reservada **delegate** é inserida imediatamente antes do tipo de retorno. Por exemplo, poderíamos ter o seguinte:

```
public delegate int Change(int x);
```

Esse representante pode ser instanciado com qualquer método que receba um parâmetro **int** e retorne um **int**. Por exemplo, considere a seguinte declaração de método:

```
static int fun1(int x);
```

O representante `Change` pode ser instanciado pelo envio do nome desse método para o construtor do representante, como no seguinte:

```
Change chgfun1 = new Change(fun1);
```

Isso pode ser reduzido para o seguinte:

```
Change chgfun1 = fun1;
```

A seguir está um exemplo de chamada a `fun1` por meio do representante `chgfun1`:

```
chgfun1(12);
```

Os objetos de uma classe representante podem armazenar mais de um método. Um segundo método pode ser adicionado com o operador +=, como no seguinte:

```
Change chgfun1 += fun2;
```

Isso coloca `fun2` no representante de `chgfun1`, mesmo que `chgfun1` tivesse anteriormente o valor **`null`**. Todos os métodos armazenados em uma instância de representante são chamados na ordem em que foram colocados na instância. Isso se chama **representante multicast**. Independentemente do que é retornado pelos métodos, é retornado somente o valor ou objeto retornado pelo último a ser chamado. Evidentemente, isso significa que, na maioria dos casos, **`void`** é retornado pelos métodos chamados por meio de um representante multicast.

Em nosso exemplo, um método estático é colocado no representante `Change`. Métodos de instância também podem ser chamados por meio de um representante, no caso em que o representante deve armazenar uma referência para o método. Os representantes também podem ser genéricos.

Eles são usados para tratamento de eventos pelos aplicativos .NET e também para implementar fechamentos (consulte a Seção 9.12).

Como acontece em C e C++, o nome de uma função sem os parênteses em Python é um ponteiro para essa função. Ada 95 tem ponteiros para subprogramas, mas Java não. Em Python e Ruby, assim como na maioria das linguagens funcionais, os subprogramas são tratados como dados, de modo que podem ser atribuídos a variáveis. Portanto, nessas linguagens há pouca necessidade de ponteiros para subprogramas.

9.8 QUESTÕES DE PROJETO PARA FUNÇÕES

As questões de projeto a seguir são específicas para funções:

- Efeitos colaterais são permitidos?
- Quais tipos de valores podem ser retornados?
- Quantos valores podem ser retornados?

9.8.1 Efeitos colaterais funcionais

Devido aos efeitos colaterais de funções chamadas em expressões, conforme descrito no Capítulo 5, parâmetros para funções sempre devem ser modo de entrada. Na verdade, algumas linguagens exigem isso – por exemplo, as funções Ada só podem ter parâmetros formais de modo de entrada. Esse requisito efetivamente impede que uma função cause efeitos colaterais por meio de seus parâmetros ou da criação de apelidos de parâmetros e variáveis globais. No entanto, na maioria das outras linguagens imperativas as funções podem ter parâmetros passados por valor ou por referência, permitindo assim as funções que causam efeitos colaterais e a criação de apelidos.

As linguagens funcionais puras, como Haskell, não têm variáveis, de modo que suas funções não podem ter efeitos colaterais.

9.8.2 Tipos de valores retornados

A maioria das linguagens de programação imperativas restringe os tipos que podem ser retornados por suas funções. C permite que qualquer tipo seja retornado por suas funções, exceto vetores e funções. Esses podem ser manipulados por valores de retorno de tipo ponteiro. C++ é como C, mas também permite que tipos definidos pelo usuário, ou classes, sejam retornados por suas funções. Ada, Python, Ruby e Lua são as únicas linguagens entre as imperativas atuais cujas funções (e/ou métodos) podem retornar valores de qualquer tipo. No caso de Ada, entretanto, como funções não são tipos, não podem ser retornadas a partir de funções. Evidentemente, ponteiros para funções podem ser retornados por funções.

Em algumas linguagens de programação, os subprogramas são objetos de primeira classe, o que significa que podem ser passados como parâmetros, retornados de funções e atribuídos a variáveis. Métodos são objetos de primeira classe em algumas linguagens imperativas, por exemplo, Python, Ruby e Lua. O mesmo vale para as funções da maioria das linguagens funcionais.

Nem Java nem C# podem ter funções, embora seus métodos sejam semelhantes às funções. Em ambas, qualquer tipo ou classe pode ser retornado pelos métodos. Como métodos não são tipos, não podem ser retornados.

9.8.3 Número de valores retornados

Na maioria das linguagens, apenas um valor pode ser retornado de uma função. Contudo, nem sempre esse é o caso. Ruby permite o retorno de mais de um valor de um método. Se uma sentença **`return`** em um método Ruby não é seguida por uma expressão, **`nil`** é retornado. Se é seguida por uma expressão, é retornado o valor da expressão. Se é seguida por mais de uma expressão, é retornado um vetor dos valores de todas as expressões.

Lua também permite que as funções retornem vários valores. Esses valores vêm após a sentença **`return`** como uma lista separada por vírgulas, como no seguinte:

```
return 3, sum, index
```

A forma da sentença que chama a função determina o número de valores recebidos pelo chamador. Se a função é chamada como um procedimento, isto é, como uma sentença, todos os valores de retorno são ignorados. Se a função retorna três valores e todos são mantidos pelo chamador, ela é chamada como no exemplo a seguir:

```
a, b, c = fun()
```

Em F#, vários valores podem ser retornados; eles são colocados em uma tupla e esta deve ser a última expressão na função.

9.9 SUBPROGRAMAS SOBRECARREGADOS

Um operador sobrecarregado tem vários significados. O significado de uma instância em particular de um operador sobrecarregado é determinado pelos tipos de seus operandos. Por exemplo, se o operador * tem dois operandos de ponto flutuante em um programa Java, ele especifica multiplicação de ponto flutuante. Mas se o mesmo operador tem dois operandos inteiros, especifica multiplicação de inteiros.

Um **subprograma sobrecarregado** é aquele cujo nome é igual ao de outro subprograma no mesmo ambiente de referenciamento. Toda versão de um subprograma sobrecarregado deve ter um protocolo exclusivo; isto é, deve ser diferente dos outros no número, na ordem ou nos tipos de seus parâmetros, e possivelmente em seu tipo de retorno. O significado de uma chamada a um subprograma sobrecarregado é determinado pela lista de parâmetros reais e/ou possivelmente pelo tipo do valor retornado. Embora não seja necessário, os subprogramas sobrecarregados normalmente implementam o mesmo processo.

C++, Java e C# incluem subprogramas sobrecarregados predefinidos. Por exemplo, muitas classes em C++, Java e C# têm construtores sobrecarregados. Como cada versão de um subprograma sobrecarregado tem um perfil de parâmetros exclusivo, o compilador pode fazer a desambiguação das ocorrências de chamadas a eles pelos parâmetros de tipo diferente. Infelizmente, isso não é tão simples. As coerções de parâmetro, quando permitidas, complicam muito o processo de desambiguação. Para resumir, a questão é esta: se nenhum perfil de parâmetros do método casar com o número e os tipos dos parâmetros reais em uma chamada de método, mas dois ou mais métodos tiverem perfis de parâmetros que podem ser casados por meio de coerções, qual método deve ser chamado? Para um projetista de linguagem responder a essa pergunta, ele deve decidir como vai classificar todas as diferentes coerções para que o compilador possa escolher o método que "melhor" casa com a chamada. Essa tarefa pode ser complicada. Para entender o grau de complexidade desse processo, sugerimos ao leitor consultar as regras de desambiguação de chamadas de método usadas em C++ (Stroustrup, 1997).

Como C++, Java e C# permitem expressões de modo misto, o tipo de retorno é irrelevante para a desambiguação de funções (ou métodos) sobrecarregadas. O contexto da chamada não permite determinar o tipo de retorno. Por exemplo, se um programa em C++ tivesse duas funções chamadas `fun` e ambas recebessem um parâmetro `int`, mas uma retornasse um `int` e a outra retornasse um `float`, o programa não compilaria, porque o compilador não poderia determinar qual versão de `fun` deveria ser usada.

Os usuários também podem escrever várias versões de subprogramas com o mesmo nome em Java, C++, C# e F#. Mais uma vez, em C++, Java e C#, os métodos sobrecarregados definidos pelo usuário mais comuns são os construtores.

Os subprogramas sobrecarregados que têm parâmetros padrão podem levar a chamadas de subprograma ambíguas. Por exemplo, considere o código C++ a seguir:

```
void fun(float b = 0.0);
void fun();
. . .
fun();
```

A chamada é ambígua e causará um erro de compilação.

9.10 SUBPROGRAMAS GENÉRICOS

O reúso pode contribuir muito para a produtividade do software. Um modo de aumentar a reutilização do software é reduzir a necessidade de criar subprogramas distintos que implementem o mesmo algoritmo em diferentes tipos de dados. Por exemplo, não deveria ser necessário um programador escrever quatro diferentes subprogramas de ordenação para ordenar quatro vetores que só diferem no tipo de elemento.

Um subprograma **polimórfico** recebe parâmetros de tipos diferentes em diferentes ativações. Os subprogramas sobrecarregados fornecem um tipo específico de polimorfismo, chamado **polimorfismo ad hoc**. Os subprogramas sobrecarregados não precisam se comportar de modo semelhante.

As linguagens que suportam programação orientada a objetos normalmente suportam polimorfismo de subtipo. **Polimorfismo de subtipo** significa que uma variável de tipo T pode acessar qualquer objeto de tipo T ou de qualquer tipo derivado de T.

Um tipo mais geral de polimorfismo é fornecido pelos métodos de Python e Ruby. Lembre-se de que, nessas linguagens, as variáveis não têm tipos, de modo que os parâmetros formais também não têm tipos. Portanto, um método funcionará para qualquer tipo de parâmetro real, desde que os operadores usados nos parâmetros formais no método sejam definidos.

O **polimorfismo paramétrico** é fornecido por um subprograma que recebe parâmetros genéricos usados em expressões de tipo que descrevem os tipos dos parâmetros do subprograma. Diferentes instanciações desses subprogramas podem receber parâmetros genéricos distintos, produzindo subprogramas que recebem diferentes tipos de parâmetros. Todas as definições paramétricas de subprogramas se comportam da mesma forma. Os subprogramas parametricamente polimórficos são frequentemente chamados de subprogramas **genéricos**. C++, Java 5.0+, C# 2005+ e F# fornecem um tipo de polimorfismo paramétrico em tempo de compilação.

9.10.1 Funções genéricas em C++

As funções genéricas em C++ recebem o nome descritivo de *funções template*. A definição de uma função template tem a seguinte forma geral

`template` <parâmetros do template>
– a definição de uma função que pode incluir os parâmetros do template

Um parâmetro de template (deve haver pelo menos um) tem uma das formas a seguir

`class` identificador
`typename` identificador

A forma `class` é usada para nomes de tipo. A forma `typename` é usada para passar um valor para a função template. Por exemplo, às vezes é conveniente passar um valor inteiro para o tamanho de um vetor na função template.

Um template pode receber outro template – na prática, é frequentemente uma classe template que define um tipo genérico definido pelo usuário, como um parâmetro, mas não vamos considerar essa opção aqui.[8]

Como exemplo de função template, considere o seguinte:

```
template <class Type>
Type max(Type first, Type second) {
  return  first > second ? first : second;
}
```

onde `Type` é o parâmetro que especifica o tipo de dado no qual a função vai operar. Essa função template pode ser instanciada para qualquer tipo para o qual o operador > for definido. Por exemplo, se ela fosse instanciada com **`int`** como parâmetro, seria

```
int max(int  first, int second) {
  return first > second ? first : second;
}
```

Embora esse processo pudesse ser definido como uma macro, a macro teria a desvantagem de não funcionar corretamente se os parâmetros fossem expressões com efeitos colaterais. Por exemplo, suponha que a macro fosse definida como

```
#define max(a, b) ((a) > (b)) ? (a) : (b)
```

Essa definição é genérica, pois funciona para qualquer tipo numérico. Contudo, ela nem sempre funciona corretamente se chamada com um parâmetro que tenha um efeito colateral, como em

```
max(x++, y)
```

que produz

```
((x++) > (y) ? (x++) : (y))
```

Sempre que o valor de `x` for maior que o de `y`, `x` será incrementado duas vezes.

As funções template de C++ são instanciadas implicitamente, ou quando a função é nomeada em uma chamada, ou quando seu endereço é obtido com o operador `&`. Por exemplo, a função template `max` anterior seria instanciada duas vezes pelo segmento de código a seguir – uma vez para parâmetros de tipo **`int`** e uma vez para parâmetros de tipo **`char`**:

```
int a, b, c;
char d, e, f;
. . .
```

[8] As classes template são discutidas no Capítulo 11.

```
c = max(a, b);
f = max(d, e);
```

A seguir está um subprograma de ordenação genérico em C++:

```
template <class Type>
void generic_sort(Type list[], int len) {
  int top, bottom;
  Type temp;
  for (top = 0; top < len - 2; top++)
    for (bottom = top + 1; bottom < len - 1; bottom++)
      if (list[top] > list[bottom]) {
        temp = list[top];
        list[top] = list[bottom];
        list[bottom] = temp;
      }  //** fim do if (list[top] . . .
}  //** fim de generic_sort
```

A seguir, temos um exemplo de instanciação dessa função template:

```
float flt_list[100];
. . .
generic_sort(flt_list, 100);
```

As funções template de C++ são uma espécie de primo pobre de um subprograma no qual os tipos dos parâmetros formais são vinculados dinamicamente aos tipos dos parâmetros reais em uma chamada. Nesse caso, apenas uma cópia do código é necessária, enquanto na estratégia de C++ deve ser criada, em tempo de compilação, uma cópia para cada tipo diferente exigido, e a vinculação de chamadas de subprograma aos subprogramas é estática.

9.10.2 Métodos genéricos em Java 5.0

O suporte para tipos e métodos genéricos foi adicionado à Java na versão 5.0. O nome de uma classe genérica em Java 5.0 é especificado por um nome, seguido por uma ou mais variáveis de tipo, delimitadas por sinais de menor que e maior que. Por exemplo,

generic_class<T>

onde `T` é a variável de tipo. Os tipos genéricos são discutidos em mais detalhes no Capítulo 11.

Os métodos genéricos de Java diferem dos subprogramas genéricos de C++ de várias maneiras importantes. Primeiro, os parâmetros genéricos devem ser classes – eles não podem ser tipos primitivos. Esse requisito inviabiliza um método genérico que imite nosso exemplo em C++, no qual os tipos de componente de vetores são genéricos e podem ser primitivos. Em Java, os componentes de vetores (ao contrário dos contêineres) não podem ser genéricos. Segundo, embora os métodos genéricos de Java possam ser instanciados qualquer número de vezes, apenas uma cópia do código é criada. A versão

interna de um método genérico, chamada de método *puro*, opera em objetos da classe `Object`. No ponto em que o valor genérico de um método genérico é retornado, o compilador insere um *cast* para o tipo correto. Terceiro, em Java, as restrições podem ser especificadas na faixa de classes que podem ser passadas como parâmetros genéricos para o método genérico. Tais restrições são denominadas **limites**.

Como exemplo do método genérico de Java 5.0, considere o seguinte esqueleto de definição de método:

```
public static <T> T doIt(T[] list) {
  . . .
}
```

Ele define um método chamado `doIt` que recebe um vetor de elementos de um tipo genérico. O nome do tipo genérico é `T` e deve ser um vetor. A seguir está um exemplo de chamada a `doIt`:

```
doIt<String>(myList);
```

Agora, considere a seguinte versão de `doIt`, que tem um limite para seu parâmetro genérico:

```
public static <T extends Comparable> T doIt(T[] list) {
  . . .
}
```

Ela define um método que recebe um parâmetro de vetor genérico, cujos elementos são de uma classe que implementa a interface `Comparable`. Essa é a restrição, ou o limite, sobre o parâmetro genérico. A palavra reservada **`extends`** parece indicar que a classe genérica torna a classe seguinte uma subclasse. Nesse contexto, entretanto, **`extends`** tem um significado diferente. A expressão `<T extends TipoLimitante>` especifica que `T` deve ser um "subtipo" do tipo limitante. Assim, nesse contexto, **`extends`** significa que a classe (ou interface) genérica estende a classe limitante (se o limite for uma classe) ou implementa a interface limitante (se o limite for uma interface). O limite garante que os elementos de qualquer instanciação do genérico podem ser comparados com o método `Comparable`, `compareTo`.

Se um método genérico tem duas ou mais restrições sobre seu tipo genérico, elas são adicionadas à cláusula **`extends`**, separadas por e comercial (`&`). Além disso, os métodos genéricos podem ter mais de um parâmetro genérico.

Java 5.0 suporta *tipos curingas*. Por exemplo, `Collection<?>` é uma classe curinga para classes de coleção. Esse tipo pode ser usado para qualquer tipo de coleção de quaisquer componentes de classe. Por exemplo, considere o método genérico a seguir:

```
void  printCollection(Collection<?> c) {
   for  (Object e: c) {
     System.out.println(e);
  }
}
```

Esse método imprime os elementos de qualquer classe `Collection`, independentemente da classe de seus componentes. Algum cuidado deve ser tomado com objetos do tipo curinga. Por exemplo, como os componentes de determinado objeto desse tipo têm um tipo, outros objetos de outros tipos não podem ser adicionados à coleção. Por exemplo, considere

```
Collection<?> c = new ArrayList<String>();
```

Seria inválido usar o método `add` para colocar algo nessa coleção, a menos que seu tipo fosse `String`.

Os tipos curinga podem ser restritos, como acontece com os tipos que não são curingas. Tais tipos são denominados *tipos curinga limitados*. Por exemplo, considere o seguinte cabeçalho de método:

```
public void drawAll(ArrayList<? extends Shape> things)
```

Aqui, o tipo genérico é um tipo curinga que é uma subclasse da classe `Shape`. Esse método poderia ser escrito para extrair qualquer objeto cujo tipo fosse uma subclasse de `Shape`.

9.10.3 Métodos genéricos em C# 2005

Os métodos genéricos de C# 2005 têm capacidade semelhante aos de Java 5.0, exceto que não há suporte para tipos curinga. Uma característica exclusiva dos métodos genéricos de C# 2005 é que os parâmetros de tipo reais de uma chamada podem ser omitidos caso o compilador possa deduzir o tipo não especificado. Por exemplo, considere o esqueleto de definição de classe:

```
class MyClass {
  public static T DoIt<T>(T p1) {
    . . .
  }
}
```

O método `DoIt` pode ser chamado sem se especificar o parâmetro genérico se o compilador puder deduzir o tipo genérico a partir do parâmetro real na chamada. Por exemplo, as duas chamadas a seguir são válidas:

```
int myInt = MyClass.DoIt(17); // Chama DoIt<int>
string myStr = MyClass.DoIt('apples');
  // Chama DoIt<string>
```

9.10.4 Funções genéricas em F#

O sistema de inferência de tipos de F# nem sempre é capaz de determinar o tipo dos parâmetros ou o tipo de retorno de uma função. Quando isso acontece, para algumas funções, F# infere um tipo genérico para os parâmetros e para o valor de retorno. Isso

se chama **generalização automática**. Por exemplo, considere a seguinte definição de função:

```
let getLast (a, b, c) = c;;
```

Como nenhuma informação de tipo foi incluída, todos os tipos dos parâmetros e do valor de retorno são inferidos como genéricos. Como essa função não inclui qualquer computação, essa é uma função genérica simples.

As funções podem ser definidas de modo a ter parâmetros genéricos, como no exemplo a seguir:

```
let printPair (x: 'a) (y: 'a) =
    printfn "%A %A" x y;;
```

A especificação de formatação `%A` serve para qualquer tipo. O apóstrofo na frente do tipo chamado a `a` especifica-o como um tipo genérico.[9] Essa definição de função funciona (com parâmetros genéricos) porque nenhuma operação de tipo restrito é incluída. Operadores aritméticos são exemplos de operações de tipo restrito. Por exemplo, considere a seguinte definição de função:

```
let adder x y = x + y;;
```

A inferência de tipos define o tipo de `x` e `y` e o tipo de retorno como **`int`**. Como não há coerção de tipo em F#, a chamada a seguir é inválida:

```
adder 2.5 3.6;;
```

Mesmo que o tipo dos parâmetros fosse definido como genérico, o operador + faria os tipos de `x` e `y` serem **`int`**.

O tipo genérico também poderia ser especificado explicitamente em sinais de menor que e maior que, como no seguinte:

```
let printPair2<'T> x y =
   printfn "%A %A" x y;;
```

Essa função deve ser chamada com um tipo,[10] como no seguinte:

```
printPair2<float> 3.5 2.4;;
```

Devido à inferência de tipos e à falta de coerções de tipo, as funções genéricas de F# são bem menos úteis, especialmente para cálculos numéricos, que as de C++, Java 5.0+ e C# 2005+.

[9] Não há nada de especial quanto a `a` – poderia ser qualquer identificador válido. Por convenção, são usadas letras minúsculas do início do alfabeto.

[10] A convenção declara explicitamente que os tipos genéricos são nomeados com letras maiúsculas a partir de `T`.

9.11 OPERADORES SOBRECARREGADOS DEFINIDOS PELO USUÁRIO

Operadores podem ser sobrecarregados pelo usuário em Ada, C++, Python e Ruby. Suponha que seja criada uma classe Python para suportar números complexos e operações aritméticas sobre eles. Um número complexo pode ser representado com dois valores de ponto flutuante. Para eles, a classe `Complex` teria membros chamados `real` e `imag`. Em Python, as operações aritméticas binárias são implementadas como chamadas de método enviadas para o primeiro operando, e o segundo operando é enviado como parâmetro. Para adição, o método é denominado `__add__`. Por exemplo, a expressão `x + y` é implementada como `x.__add__(y)`. A fim de sobrecarregar + para a adição de objetos da nova classe `Complex`, só precisamos fornecer a `Complex` um método denominado `__add__` que efetue a operação. Tal método aparece a seguir:

```
def __add__ (self, second):
  return Complex(self.real + second.real, self.imag +
   second.imag)
```

Na maioria das linguagens que suportam programação orientada a objetos, uma referência para o objeto atual é enviada implicitamente com cada chamada de método. Em Python, essa referência deve ser enviada explicitamente; é por isso que `self` é o primeiro parâmetro de nosso método `__add__`.

O exemplo de método de adição poderia ser escrito para uma classe complexa em C++, como segue[11]:

```
Complex operator +(Complex &second) {
  return Complex(real + second.real, imag + second.imag);
}
```

9.12 FECHAMENTOS

Definir um **fechamento** é simples: é um subprograma e o ambiente de referenciamento onde foi definido. O ambiente de referenciamento é necessário se o subprograma pode ser chamado a partir de qualquer lugar arbitrário no programa. Explicar um fechamento não é tão simples.

Se uma linguagem de programação de escopo estático não permite subprogramas aninhados, os fechamentos não têm utilidade, de modo que tal linguagem não oferece suporte para eles. Todas as variáveis no ambiente de referenciamento de um subprograma em tal linguagem (suas variáveis locais e as variáveis globais) são acessíveis, independentemente do lugar no programa onde o subprograma é chamado.

Quando subprogramas podem ser aninhados, além das variáveis locais e globais, o seu ambiente de referenciamento pode incluir as variáveis definidas em todos os subprogramas que o envolvem. No entanto, isso não é problema se o subprograma só pode

[11]Tanto C++ como Python têm classes predefinidas para números complexos; portanto, nossos exemplos de métodos são desnecessários, exceto como ilustrações.

ser chamado em lugares onde todos os escopos que o envolvem são ativos e visíveis. Isso se torna um problema se um subprograma pode ser chamado em outro lugar. Isso pode acontecer se o subprograma puder ser passado como parâmetro ou atribuído a uma variável, o que permite que seja chamado praticamente a partir de qualquer lugar no programa. Há um problema associado: o subprograma pode ser chamado depois de um ou mais de seus subprogramas aninhadores ter terminado, o que normalmente significa que as variáveis definidas em tais subprogramas aninhadores foram liberadas – elas não existem mais. Para que o subprograma possa ser chamado a partir de qualquer lugar no programa, seu ambiente de referenciamento deve estar disponível em todo lugar em que ele possa ser chamado. Portanto, as variáveis definidas nos subprogramas aninhadores talvez precisem de tempos de vida do programa inteiro, em vez de apenas o tempo durante o qual o subprograma em que foram definidos estiver ativo. Diz-se que uma variável cujo tempo de vida é o do programa inteiro tem **extensão ilimitada**. Isso normalmente significa que ela deve ser dinâmica do monte, em vez de dinâmica da pilha.

Quase todas as linguagens de programação funcionais, a maioria das linguagens de *scripting* e pelo menos uma linguagem principalmente imperativa, C#, suportam fechamentos. Essas linguagens têm escopo estático, permitem subprogramas aninhados[12] e possibilitam que subprogramas sejam passados como parâmetros. A seguir, temos um exemplo de fechamento escrito em JavaScript:

```
function makeAdder(x) {
    return function(y) {return x + y;}
}
. . .
var add10 = makeAdder(10);
var add5 = makeAdder(5);
document.write("Add 10 to 20: " + add10(20) +
"<br />");
document.write("Add 5 to 20: " + add5(20) +
"<br />");
```

A saída desse código, supondo que foi embutido em um documento HTML e exibido com um navegador, é a seguinte:

```
Add 10 to 20: 30
Add 5 to 20: 25
```

Nesse exemplo, o fechamento é a função anônima definida dentro da função `makeAdder`, a qual `makeAdder` retorna. A variável `x` referenciada na função de fechamento é vinculada ao parâmetro enviado para `makeAdder`. A função `makeAdder` é chamada duas vezes, uma com o parâmetro `10` e uma com o `5`. Cada uma dessas chamadas retorna uma versão diferente do fechamento, pois são vinculadas a diferentes valores de `x`. A primeira chamada a `makeAdder` cria uma função que soma `10` ao seu parâmetro; a segunda cria uma função que soma `5` ao seu parâmetro. As duas versões da função são vinculadas a diferentes ativações de `makeAdder`. Obviamente, o tempo de vida da

[12]Em C#, os únicos métodos que podem ser aninhados são representantes anônimos e expressões lambda.

versão de x criada quando makeAdder é chamada deve ultrapassar o tempo de vida do programa.

Essa mesma função de fechamento pode ser escrita em C# com um representante anônimo aninhado. O tipo de método de aninhamento é especificado como uma função que recebe um **int** como parâmetro e retorna um representante anônimo. O tipo de retorno é especificado com a notação especial para tais representantes, Func<int, int>. O primeiro tipo nos sinais de menor que e maior que é o tipo do parâmetro. Tal representante pode encapsular métodos que têm apenas um parâmetro. O segundo tipo é o tipo de retorno do método encapsulado pelo representante.

```
static Func<int, int> makeAdder(int x) {
  return delegate(int y) { return x + y;};
}
. . .
Func<int, int> Add10 = makeAdder(10);
Func<int, int> Add5 = makeAdder(5);
Console.WriteLine("Add 10 to 20: {0}", Add10(20));
Console.WriteLine("Add 5 to 20: {0}", Add5(20));
```

A saída desse código é exatamente a mesma do exemplo anterior de fechamento com JavaScript.

O representante anônimo poderia ser escrito como uma expressão lambda. A seguir está uma substituição para o corpo do método makeAdder que usa uma expressão lambda em vez do representante:

```
return y => x + y
```

Os blocos de Ruby são implementados de modo a poderem referenciar variáveis visíveis na posição em que foram definidas, mesmo que sejam chamados em um lugar no qual essas variáveis teriam desaparecido. Isso transforma tais blocos em fechamentos.

9.13 CORROTINAS

Corrotina é um tipo especial de subprograma. Em vez de viabilizar um relacionamento mestre-escravo entre um subprograma chamador e um chamado existente em subprogramas convencionais, as corrotinas chamadoras e chamadas são mais equitativas. Na verdade, o mecanismo de controle de corrotinas é muitas vezes chamado de **modelo de controle de unidades simétrico**.

As corrotinas podem ter vários pontos de entrada, os quais são controlados por elas próprias. Elas também têm meios de manter seu status entre ativações. Isso significa que devem ser sensíveis ao histórico de execução e, assim, ter variáveis estáticas locais. Execuções secundárias de uma corrotina frequentemente começam em pontos diferentes de seu início. Por isso, a invocação de uma corrotina é denominada **retomada** (resume) em vez de chamada.

Por exemplo, considere o esqueleto de corrotina:

```
sub co1(){
  . . .
  resume  co2();
  . . .
  resume  co3();
  . . .
}
```

Na primeira vez que `co1` é retomada, sua execução começa na primeira sentença e executa para baixo até (e incluindo) a retomada de `co2`, o que transfere o controle para `co2`. Na próxima vez que `co1` é retomada, sua execução começa na primeira sentença após sua chamada a `co2`. Quando `co1` é retomada pela terceira vez, sua execução começa na primeira sentença após a retomada de `co3`.

Uma das características comuns dos subprogramas é mantida nas corrotinas: apenas uma está realmente em execução em dado momento.

Como vimos no exemplo acima, em vez de executar até o final, muitas vezes uma corrotina executa parcialmente e, então, transfere o controle para alguma outra corrotina; e, quando é reiniciada, uma corrotina retoma a execução imediatamente após a sentença que utilizou para transferir o controle para outro lugar. Esse tipo de sequência de execução intercalada está relacionado ao funcionamento dos sistemas operacionais de multiprogramação. Embora possam estar em apenas um processador, em tal sistema, todos os programas em execução parecem ser concorrentes enquanto compartilham o processador. No caso das corrotinas, isso às vezes é chamado de **quase concorrência**.

Normalmente, as corrotinas são criadas em um aplicativo por uma unidade de programa chamada unidade mestra, que não é uma corrotina. Quando criadas, elas executam seu código de inicialização e, então, retornam o controle para essa unidade mestra. Quando a família de corrotinas inteira é construída, o programa mestre retoma uma das corrotinas e, então, os membros da família retomam um ao outro em alguma ordem, até que seu trabalho tenha terminado, se puder ser terminado. Se a execução de uma corrotina atinge o final de sua seção de código, o controle é transferido para a unidade mestra que a criou. Esse é o mecanismo para finalizar a execução da coleção de corrotinas, quando isso é desejável. Em alguns programas, as corrotinas executam sempre que o computador está funcionando.

Um exemplo de problema que pode ser resolvido com esse tipo de coleção de corrotinas é a simulação de um jogo de cartas. Suponha que o jogo tenha quatro jogadores, todos usando a mesma estratégia. Tal jogo pode ser simulado fazendo-se com que uma unidade de programa mestra crie uma família de quatro corrotinas, cada uma com uma coleção, ou mão, de cartas. Então, o programa mestre poderia iniciar a simulação retomando uma das corrotinas de jogador, a qual, depois de ter jogado sua vez, poderia retomar a próxima corrotina de jogador e assim por diante, até que o jogo terminasse.

Suponha que as unidades de programa `A` e `B` sejam corrotinas. A Figura 9.3 mostra duas maneiras como uma sequência de execução envolvendo `A` e `B` poderia ocorrer.

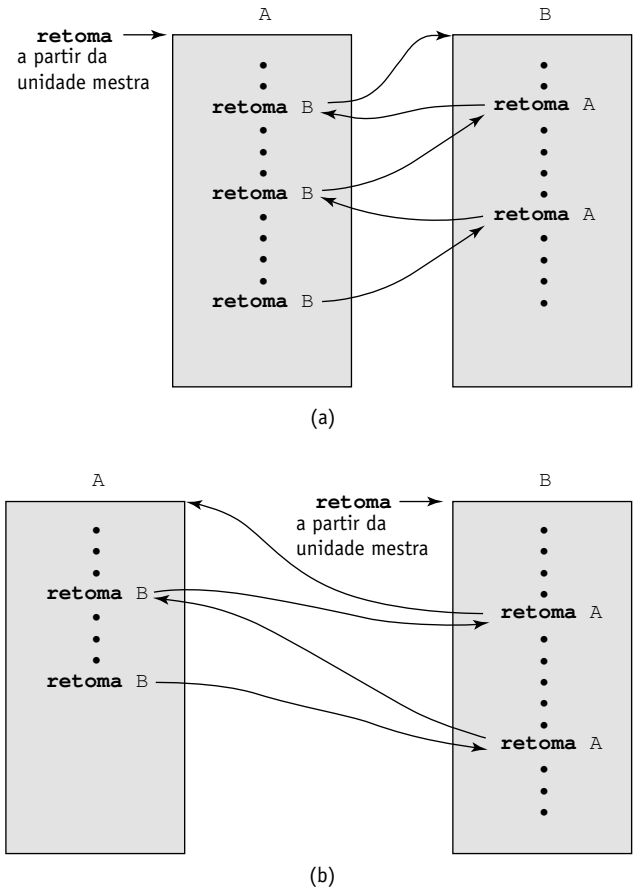


FIGURA 9.3
Duas possíveis sequências de controle de execução para duas corrotinas sem laços.

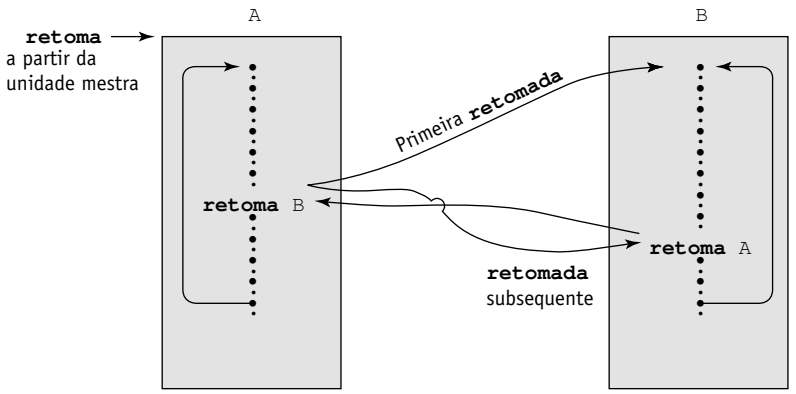


FIGURA 9.4
Sequência de execução de corrotina com laços.

Na Figura 9.3a, a execução da corrotina `A` é iniciada pela unidade mestra. Após alguma execução, `A` inicia `B`. Quando a corrotina `B`, na Figura 9.3a, faz com que o controle volte para a corrotina `A`, a semântica é que `A` continua a partir de onde terminou sua última execução. Em particular, suas variáveis locais têm os valores deixados nelas pela ativação anterior. A Figura 9.3b mostra uma sequência de execução alternativa das corrotinas `A` e `B`. Nesse caso, `B` é iniciada pela unidade mestra.

Em vez de ter os padrões mostrados na Figura 9.3, frequentemente uma corrotina possui um laço contendo uma retomada. A Figura 9.4 mostra a sequência de execução desse cenário. Nesse caso, `A` é iniciada pela unidade mestra. Dentro de seu laço principal, `A` retoma `B`, que, por sua vez, retoma `A` em seu laço principal.

Entre as linguagens contemporâneas, somente Lua suporta corrotinas integralmente.[13]

RESUMO

As abstrações de processo são representadas nas linguagens de programação por subprogramas. A definição de um subprograma descreve as ações representadas por ele. A chamada de um subprograma ativa essas ações. O cabeçalho de um subprograma identifica sua definição e fornece sua interface, denominada protocolo.

Parâmetros formais são os nomes utilizados pelos subprogramas para fazer referência aos parâmetros reais dados em chamadas de subprograma. Em Python e Ruby, parâmetros de vetor e formais de dispersão são usados para suportar números variáveis de parâmetros. Lua e JavaScript também suportam números variáveis de parâmetros. Os parâmetros reais podem ser associados aos parâmetros formais por posição ou por palavra-chave. Os parâmetros podem ter valores padrão.

Subprogramas podem ser funções, as quais modelam funções matemáticas e são usadas para definir novas operações, ou procedimentos, os quais definem novas sentenças.

Nos subprogramas, as variáveis locais podem ser dinâmicas da pilha, oferecendo suporte para recursão, ou estáticas, fornecendo eficiência e variáveis locais sensíveis ao histórico de execução.

JavaScript, Python, Ruby e Lua permitem aninhar definições de subprograma.

Existem três modelos de semântica de passagem de parâmetros fundamentais – modo de entrada, modo de saída e modo de entrada e saída – e várias estratégias para implementá-los. São elas: passagem por valor, passagem por resultado, passagem por valor-resultado, passagem por referência e passagem por nome. Na maioria das linguagens, os parâmetros são passados na pilha de tempo de execução.

A criação de apelidos pode ocorrer quando são usados parâmetros passados por referência, tanto entre dois ou mais parâmetros como entre um parâmetro e uma variável não local acessível.

Parâmetros que são matrizes multidimensionais apresentam alguns problemas para o projetista de linguagens, pois o subprograma chamado precisa saber como computar a função de mapeamento de armazenamento para eles. Isso exige mais do que apenas o nome da matriz.

[13] Contudo, os geradores de Python são uma forma de corrotinas.

Os parâmetros que são nomes de subprograma fornecem um serviço necessário, mas podem ser difíceis de entender. A opacidade reside no ambiente de referenciamento que está disponível quando um subprograma passado como parâmetro é executado.

C e C++ suportam ponteiros para funções. C# tem representantes, os quais são objetos que podem armazenar referências para métodos. Os representantes podem oferecer suporte para chamadas multicast, armazenando mais de uma referência de método.

Ada, C++, C#, Ruby e Python permitem sobrecarga tanto de subprogramas como de operadores. Os subprogramas podem ser sobrecarregados contanto que possa ser feita a desambiguação das diversas versões pelos tipos de seus parâmetros ou valores retornados. Definições de função podem ser usadas para dar significados adicionais a operadores.

Em C++, Java 5.0 e C# 2005, os subprogramas podem ser genéricos, e o polimorfismo paramétrico pode ser usado para que os tipos desejados para seus objetos de dados possam ser passados para o compilador, o qual pode então construir unidades para os tipos solicitados.

O projetista de um recurso de função em uma linguagem deve decidir quais restrições serão impostas para os valores retornados e também o número de valores de retorno.

Um fechamento é um subprograma e seu ambiente de referenciamento. Fechamentos são úteis em linguagens que permitem subprogramas aninhados, têm escopo estático e possibilitam que subprogramas sejam retornados de funções e atribuídos a variáveis.

Uma corrotina é um subprograma especial que possui várias entradas. Pode ser usada para fornecer execução intercalada de subprogramas.

QUESTÕES DE REVISÃO

1. Quais são as três características gerais dos subprogramas?
2. O que significa para um subprograma estar ativo?
3. O que consta no cabeçalho de um subprograma?
4. Qual característica dos subprogramas Python os diferencia dos de outras linguagens?
5. Quais linguagens permitem um número variável de parâmetros?
6. O que é um parâmetro de vetor formal em Ruby?
7. O que é um perfil de parâmetros? O que é um protocolo de subprograma?
8. O que são parâmetros formais? O que são parâmetros reais?
9. Quais são as vantagens e desvantagens dos parâmetros de palavra-chave?
10. Quais são as diferenças entre uma função e um procedimento?
11. Quais são as questões de projeto para subprogramas?
12. Quais são as vantagens e desvantagens das variáveis locais dinâmicas?

13. Quais são as vantagens e desvantagens das variáveis locais estáticas?
14. Quais linguagens permitem aninhar definições de subprograma?
15. Quais são os três modelos semânticos de passagem de parâmetros?
16. Quais são os modos, os modelos conceituais de transferência, as vantagens e as desvantagens dos métodos de passagem de parâmetros por valor, por resultado, por valor-resultado e por referência?
17. Descreva as maneiras pelas quais podem ocorrer apelidos com parâmetros passados por referência.
18. Qual é a diferença entre a maneira como C original e C89 lidam com um parâmetro real cujo tipo não é idêntico ao do parâmetro formal correspondente?
19. Quais são as duas considerações de projeto fundamentais para métodos de passagem de parâmetros?
20. Descreva o problema da passagem de matrizes multidimensionais como parâmetros.
21. Qual é o nome do método de passagem de parâmetros usado em Ruby?
22. Quais são as duas questões que surgem quando nomes de subprograma são parâmetros?
23. Defina *vinculação rasa* e *profunda* para ambientes de referência de subprogramas passados como parâmetros.
24. O que é um subprograma sobrecarregado?
25. O que é polimorfismo paramétrico?
26. O que faz uma função template C++ ser instanciada?
27. De que maneiras fundamentais os parâmetros genéricos para um método genérico Java 5.0 diferem dos parâmetros genéricos dos métodos C++?
28. Se um método Java 5.0 retorna um tipo genérico, que tipo de objeto é retornado na realidade?
29. Se um método genérico Java 5.0 é chamado com três parâmetros genéricos diferentes, quantas versões do método serão geradas pelo compilador?
30. Quais são as questões de projeto para funções?
31. Quais são as duas linguagens que permitem retornar vários valores de uma função?
32. O que exatamente é um representante?
33. Qual é a principal desvantagem das funções genéricas em F#?
34. O que é fechamento?
35. Quais são as características da linguagem que tornam os fechamentos úteis?
36. Quais linguagens permitem ao usuário carregar operadores?
37. De que maneiras as corrotinas são diferentes dos subprogramas convencionais?

PROBLEMAS

1. Quais são os argumentos contra e a favor do fato de um programa de usuário criar definições adicionais para operadores existentes, como acontece em Python e C++? Você acha tal sobrecarga de operador definida pelo usuário boa ou ruim? Justifique sua resposta.
2. Na maioria das implementações de Fortran IV, todos os parâmetros eram passados por referência, usando apenas transmissão de caminho de acesso. Cite as vantagens e desvantagens dessa escolha de projeto.
3. Argumente a favor da decisão dos projetistas de Ada 83 de permitir que o implementador escolha entre implementar parâmetros de modo de entrada e saída por cópia ou por referência.
4. Suponha que você queira escrever um método que imprima um cabeçalho em uma nova página de saída, junto com um número de página que é 1 na primeira ativação e aumenta em 1 a cada ativação subsequente. Isso pode ser feito sem parâmetros e sem referência para variáveis não locais em Java? Isso pode ser feito em C#?
5. Considere o programa a seguir, escrito em sintaxe C:

```
void swap(int a, int b) {
  int temp;
  temp = a;
  a = b;
  b = temp;
}
void main() {
  int value = 2, list[5] = {1, 3, 5, 7, 9};
  swap(value, list[0]);
  swap(list[0], list[1]);
  swap(value, list[value]);
}
```

   Para cada um dos seguintes métodos de passagem de parâmetros, quais são todos os valores das variáveis `value` e `list` após cada uma das três chamadas a `swap`?

   a. Passados por valor

   b. Passados por referência

   c. Passados por valor-resultado

6. Apresente um argumento contra o fornecimento de variáveis locais estáticas e dinâmicas em subprogramas.
7. Considere o programa a seguir, escrito em sintaxe C:

```
void fun (int first, int second) {
  first += first;
  second += second;
```

```
}
void main() {
  int list[2] = {1, 3};
  fun(list[0], list[1]);
}
```

Para cada um dos seguintes métodos de passagem de parâmetro, quais são os valores do vetor `list` após a execução?

a. Passado por valor
b. Passado por referência
c. Passado por valor-resultado

8. Argumente contra o projeto de C de fornecer somente subprogramas.
9. Com base em um livro-texto sobre Fortran, aprenda a sintaxe e a semântica das funções de declaração. Justifique a existência delas em Fortran.
10. Estude os métodos de sobrecarga de operadores definida pelo usuário em C++ e Ada e escreva um relatório comparando os dois com base em nossos critérios de avaliação de linguagens.
11. C# suporta parâmetros de modo de saída, mas nem Java nem C++ o fazem. Explique a diferença.
12. Pesquise o dispositivo de Jensen, que foi um uso de parâmetros passados por nome, e redija uma breve descrição do que ele é e como pode ser usado.
13. Estude os mecanismos iteradores de Ruby e CLU e liste suas semelhanças e diferenças.
14. Reflita sobre subprogramas aninhados nas linguagens de programação – por que eles não são permitidos em muitas linguagens contemporâneas?
15. Liste pelo menos dois argumentos contra o uso de parâmetros passados por nome.
16. Escreva uma comparação detalhada dos subprogramas genéricos de Java 5.0 e C# 2005.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Escreva um programa, em uma linguagem que você conheça, para determinar a razão do tempo exigido para passar um vetor grande por referência e o tempo exigido para passar o mesmo vetor por valor. Torne o vetor tão grande quanto possível na máquina e na implementação usadas. Passe o vetor quantas vezes for necessário para obter cronometragens razoavelmente precisas das operações de passagem.
2. Escreva um programa em C# ou Ada que determine quando o endereço de um parâmetro de modo de saída é computado (no momento da chamada ou no momento em que a execução do subprograma termina).

3. Escreva um programa em Perl que passe um literal por referência para um subprograma, o qual tenta alterar o parâmetro. Dada a filosofia de projeto geral de Perl, explique os resultados.
4. Repita o exercício anterior em C#.
5. Escreva um programa em alguma linguagem que tenha variáveis locais estáticas e dinâmicas da pilha em subprogramas. Crie seis matrizes grandes (no mínimo 100 x 100) no subprograma – três estáticas e três dinâmicas da pilha. Preencha duas das matrizes estáticas e duas das matrizes dinâmicas da pilha com números aleatórios na faixa de 1 a 100. O código do subprograma deve efetuar um grande número de operações de multiplicação matricial nas matrizes estáticas e cronometrar o processo. Então, deve repetir isso com as matrizes dinâmicas da pilha. Compare e explique os resultados.
6. Escreva um programa em C# que inclua dois métodos chamados um grande número de vezes. Um vetor grande é passado aos dois métodos, um por valor e um por referência. Compare os tempos necessários para chamar esses dois métodos e explique a diferença. Certifique-se de chamá-los um número de vezes suficiente para ilustrar a diferença no tempo exigido.
7. Usando a sintaxe de qualquer linguagem de que você goste, escreva um programa que produza comportamento diferente dependendo de a passagem de seu parâmetro ser por referência ou por valor-resultado.
8. Escreva uma função genérica em C++ que receba um vetor de elementos genéricos e um escalar do mesmo tipo dos elementos do vetor. O tipo dos elementos do vetor e do escalar é o parâmetro genérico. A função deve procurar no vetor o escalar dado e retornar o índice do escalar no vetor. Se o escalar não estiver no vetor, a função deve retornar `-1`. Teste a função para tipos `int` e `float`.
9. Crie um subprograma e um código de chamada em que a passagem por referência e a passagem por valor-resultado de um ou mais parâmetros produzam resultados diferentes.

10

Implementação de subprogramas

10.1 A semântica geral de chamadas e retornos

10.2 Implementação de subprogramas “simples”

10.3 Implementação de subprogramas com variáveis locais dinâmicas da pilha

10.4 Subprogramas aninhados

10.5 Blocos

10.6 Implementação de escopo dinâmico

O propósito deste capítulo é explorar a implementação de subprogramas. A discussão fornecerá ao leitor algum conhecimento sobre como a ligação entre subprogramas funciona e por que ALGOL 60 foi um desafio para os escritores de compiladores desavisados no início dos anos 1960. Começamos com a situação mais simples – subprogramas não aninhados com variáveis locais estáticas –, avançamos para subprogramas mais complicados com variáveis locais dinâmicas da pilha e concluímos com subprogramas aninhados com variáveis locais dinâmicas da pilha e escopo estático. A crescente dificuldade de implementar subprogramas em linguagens com subprogramas aninhados é causada pela necessidade de incluir mecanismos para acessar variáveis não locais.

O método de cadeia estática para acessar variáveis não locais em linguagens com escopo estático é abordado em detalhes. Em seguida, são descritas técnicas para implementar blocos. Por fim, são discutidos vários métodos de implementação de acesso a variáveis não locais em uma linguagem de escopo dinâmico.

10.1 A SEMÂNTICA GERAL DE CHAMADAS E RETORNOS

Juntas, as operações de chamada e retorno de subprogramas são chamadas de **ligação de subprogramas**. A implementação de subprogramas deve ser baseada na semântica da ligação de subprogramas da linguagem a ser implementada.

Uma chamada a um subprograma em uma linguagem típica tem diversas ações a ela associadas. O processo de chamada deve incluir a implementação de qualquer método de passagem de parâmetros usado. Se as variáveis locais não são estáticas, o processo de chamadas deve alocar armazenamento para as variáveis locais declaradas no subprograma chamado e vincular essas variáveis a esse armazenamento. Ele deve salvar o estado de execução da unidade de programa chamadora. O estado de execução é tudo que é necessário para retomar a execução dessa unidade. Isso inclui os valores de registro, os bits de estado da CPU e o ponteiro de ambiente (PE). O PE, discutido com mais detalhes na Seção 10.3, é usado para acessar parâmetros e variáveis locais durante a execução de um subprograma. O processo de chamada também deve providenciar a transferência de controle para o código do subprograma e garantir que o controle possa retornar ao local apropriado quando a execução do subprograma estiver completa. Por fim, se a linguagem suporta subprogramas aninhados, o processo de chamada deve criar algum mecanismo para fornecer acesso às variáveis não locais visíveis para o subprograma chamado.

As ações exigidas do retorno de um subprograma são menos complicadas do que as de uma chamada. Se o subprograma tem parâmetros do modo de saída ou do modo de entrada e saída e eles são implementados por cópia, a primeira ação do processo de retorno é mover os valores locais dos parâmetros formais associados para os parâmetros reais. A seguir, ele deve liberar o armazenamento usado para variáveis locais e restaurar o estado de execução da unidade de programa chamadora. Por fim, o controle deve ser devolvido à unidade de programa chamadora.

10.2 IMPLEMENTAÇÃO DE SUBPROGRAMAS "SIMPLES"

Começamos com a tarefa de implementar subprogramas simples. Neste caso, "simples" significa que os subprogramas não podem ser aninhados e que todas as variáveis locais

são estáticas. As primeiras versões de Fortran são exemplos de linguagens que tinham esse tipo de subprograma.

A semântica de uma chamada a um subprograma "simples" requer as seguintes ações:

1. Gravar o estado da execução da unidade de programa atual.
2. Calcular e passar os parâmetros.
3. Passar o endereço de retorno para o subprograma chamado.
4. Transferir o controle para o subprograma chamado.

A semântica de um retorno de um subprograma simples requer as seguintes ações:

1. Se existirem parâmetros com passagem por valor-resultado ou passagem por modo de saída, os seus valores atuais são movidos ou disponibilizados para os parâmetros reais correspondentes.
2. Se o subprograma é uma função, o valor funcional é movido para um local acessível ao chamador.
3. O estado da execução do chamador é restaurado.
4. O controle é transferido de volta para o chamador.

As ações de chamada e de retorno exigem armazenamento para o seguinte:

- Informações de estado sobre o chamador
- Parâmetros
- Endereço de retorno
- Valor de retorno para funções
- Variáveis temporárias usadas pelo código dos subprogramas

Esses itens, junto com as variáveis locais e o código de subprograma, formam a coleção completa de informações que um subprograma precisa para executar e retornar o controle para o chamador.

A questão agora é a distribuição das ações de chamada e retorno para o chamador e o chamado. Para subprogramas simples, a resposta é óbvia para a maioria das partes do processo. As três últimas ações de uma chamada devem ser feitas pelo chamador. A gravação do estado da execução do chamador pode ser feita por ambos. No caso do retorno, a primeira, a terceira e a quarta ações devem ser realizadas pelo chamado. Mais uma vez, o restabelecimento do estado de execução do chamador pode ser feito pelo próprio chamador ou pelo chamado. Em geral, as ações de ligação do chamado podem ocorrer em dois momentos diferentes: no início de sua execução ou no fim. Às vezes elas são chamadas de **prólogo** e **epílogo** da ligação do subprograma. No caso de um subprograma simples, todas as ações de ligação do chamado ocorrem no final da execução; portanto, não há necessidade de um prólogo.

Um subprograma simples consiste em duas partes: o código real do subprograma, que é constante, e as variáveis locais e os dados listados anteriormente, que podem mu-

dar quando o subprograma é executado. No caso dos subprogramas simples, ambas as partes têm tamanhos fixos.

O formato, ou layout, da parte que não é código de um subprograma é chamado de **registro de ativação**, porque os dados que descreve são relevantes apenas durante a ativação ou execução do subprograma. A forma de um registro de ativação é estática. Uma **instância de registro de ativação** é um exemplo concreto de registro de ativação, uma coleção de dados na forma de registro de ativação.

Como as linguagens com subprogramas simples não suportam recursividade, pode haver apenas uma versão ativa de um subprograma de cada vez. Portanto, pode haver apenas uma instância do registro de ativação para um subprograma. Um possível layout para registros de ativação é mostrado na Figura 10.1. O estado da execução do chamador salvo é omitido aqui e no restante deste capítulo, porque ele é simples e irrelevante para a discussão.

Como uma instância de um registro de ativação para um subprograma "simples" tem tamanho fixo, ele pode ser alocado estaticamente. Na verdade, ele poderia ser anexado à parte do código do subprograma.

A Figura 10.2 mostra um programa formado por um programa principal e três subprogramas: `A`, `B` e `C`. Embora a figura mostre todos os segmentos de código separados de todas as instâncias de registro de ativação, em alguns casos, as instâncias de registro de ativação são anexadas aos seus segmentos de código associados.

A construção do programa completo mostrado na Figura 10.2 não é feita inteiramente pelo compilador. Na verdade, se a linguagem permite compilação independente, as quatro unidades do programa – `MAIN`, `A`, `B` e `C` – podem ter sido compiladas em dias diferentes ou mesmo em anos diferentes. No momento em que cada unidade é compilada, o código de máquina para ela, com uma lista de referências aos subprogramas externos, é escrito em um arquivo. O programa executável mostrado na Figura 10.2 é unido pelo **ligador**, o qual faz parte do sistema operacional. (Algumas vezes, os ligadores são chamados de *carregadores*, *ligadores/carregadores* ou *editores de ligação*.) Quando o ligador é chamado para um programa principal, sua primeira tarefa é encontrar os arquivos que contêm os subprogramas traduzidos referenciados no programa e carregá-los na memória. Então, o ligador deve configurar os endereços de destino de todas as chamadas aos subprogramas no programa principal com os endereços de entrada desses subprogramas. O mesmo deve ser feito para as chamadas de subprogramas nos subprogramas carregados e para as chamadas a subprogramas em bibliotecas. No exemplo anterior, o ligador foi chamado para `MAIN`. O ligador precisou encontrar o código de máquina dos programas `A`, `B` e `C`, com suas instâncias de registro de ativação, e carregá-los na memória com o código para `MAIN`. Depois, ele teve de corrigir os endereços de destino para todas as chamadas para `A`, `B`, `C` e para quaisquer subprogramas de biblioteca chamados em `A`, `B`, `C` e `MAIN`.

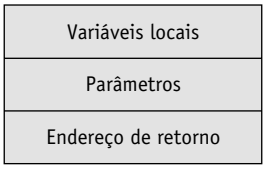


FIGURA 10.1
Um registro de ativação para subprograma simples.

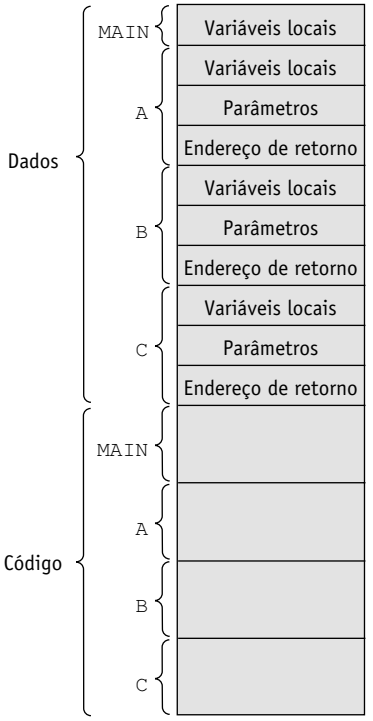


FIGURA 10.2
O código e os registros de ativação de um programa com subprogramas simples.

10.3 IMPLEMENTAÇÃO DE SUBPROGRAMAS COM VARIÁVEIS LOCAIS DINÂMICAS DA PILHA

Agora, examinaremos a implementação da ligação de subprogramas em linguagens nas quais as variáveis locais são dinâmicas da pilha, mais uma vez enfocando as operações de chamada e de retorno.

Uma das vantagens mais importantes das variáveis locais dinâmicas da pilha é o suporte à recursão. Logo, linguagens que usam essas variáveis também suportam recursão.

Uma discussão sobre a complexidade adicional necessária quando subprogramas podem ser aninhados será feita na Seção 10.4.

10.3.1 Registros de ativação mais complexos

A ligação de subprogramas em linguagens que usam variáveis locais dinâmicas da pilha é mais complexa do que a ligação de subprogramas simples pelas seguintes razões:

- O compilador precisa gerar código que faça alocação e liberação implícitas de variáveis locais.
- A recursão adiciona a possibilidade de múltiplas ativações simultâneas de um subprograma, ou seja, pode existir mais de uma instância (execução incompleta) de

um subprograma em dado momento, com ao menos uma chamada de fora do subprograma e uma ou mais chamadas recursivas. O número de ativações é limitado apenas pelo tamanho da memória da máquina. Cada uma delas requer sua instância de registro de ativação.

Na maioria das linguagens, o formato de um registro de ativação para um subprograma é conhecido em tempo de compilação. Em muitos casos, o tamanho também é conhecido para registros de ativação, porque todos os dados locais são de tamanho fixo. Esse não é o caso em algumas outras linguagens, como Ada, na qual o tamanho de uma matriz local pode depender do valor de um parâmetro real. Nesses casos, o formato é estático, mas o tamanho pode ser dinâmico. Em linguagens com variáveis locais dinâmicas da pilha, as instâncias de registros de ativação devem ser criadas dinamicamente. O registro de ativação típico para tais linguagens é mostrado na Figura 10.3.

Como o endereço de retorno, a ligação dinâmica e os parâmetros são colocados na instância de registro de ativação pelo chamador, essas entradas devem aparecer primeiro.

O endereço de retorno normalmente consiste em um ponteiro para a instrução seguinte à chamada no segmento de código da unidade de programa chamadora. A **ligação dinâmica** é um ponteiro para a base da instância de registro de ativação do chamador. Em linguagens de escopo estático, essa ligação é usada para fornecer informações de rastreamento quando um erro em tempo de execução ocorre. Em linguagens de escopo dinâmico, a ligação dinâmica é usada para acessar variáveis não locais. Os parâmetros reais no registro de ativação são os valores ou os endereços fornecidos pelo chamador.

Variáveis escalares locais são vinculadas ao armazenamento dentro de uma instância de registro de ativação. Variáveis locais que são estruturas às vezes são alocadas em outro lugar, e apenas suas descrições e um ponteiro para esse armazenamento fazem parte do registro de ativação. Variáveis locais são alocadas e possivelmente inicializadas no subprograma chamado; portanto, aparecem por último.

Considere o esqueleto de função em C:

```
void sub(float total, int part) {
  int list[5];
  float sum;
  . . .
}
```

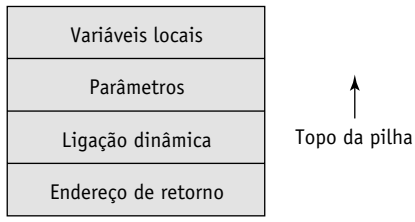


FIGURA 10.3
Um registro de ativação típico para uma linguagem com variáveis locais dinâmicas da pilha.

O registro de ativação para `sub` é mostrado na Figura 10.4.

Ativar um subprograma requer a criação dinâmica de uma instância de registro de ativação para ele. Conforme mencionado anteriormente, o formato do registro de ativação é fixado em tempo de compilação, apesar de, em algumas linguagens, seu tamanho poder depender da chamada. Como a semântica de chamada e de retorno especifica que o último subprograma chamado é o primeiro a ser completado, é razoável criar instâncias desses registros de ativação em uma pilha. Essa pilha faz parte do sistema de tempo de execução e, assim, é chamada de **pilha de tempo de execução**, apesar de normalmente nos referirmos a ela simplesmente como "pilha". Cada ativação de subprograma, seja recursiva ou não recursiva, cria uma instância de um registro de ativação na pilha. Isso fornece as cópias separadas necessárias dos parâmetros, das variáveis locais e do endereço de retorno.

Mais um item é necessário para controlar a execução de um subprograma – o PE. Inicialmente, o PE aponta para a base, ou primeiro endereço da instância de registro de ativação do programa principal. O sistema de tempo de execução deve garantir que ele sempre aponte para a base da instância do registro de ativação da unidade de programa executada. Quando um subprograma é chamado, o PE atual é salvo como a ligação dinâmica na nova instância de registro de ativação. Ele é então configurado para apontar para a base da nova instância de registro de ativação. Após o retorno do subprograma, o topo da pilha é configurado para o valor do PE atual menos um, e o PE é configurado para a ligação dinâmica da instância de registro de ativação do subprograma que completou sua execução. Reinicializar o topo da pilha efetivamente remove a instância de registro de ativação do topo.

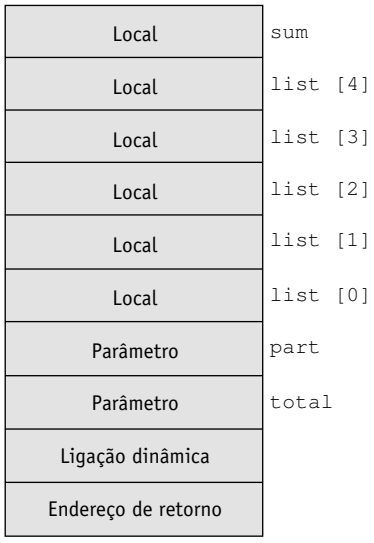


FIGURA 10.4
O registro de ativação para a função `sub`.

O PE é usado como a base do deslocamento de endereçamento do conteúdo de dados da instância de registro de ativação – parâmetros e variáveis locais.

Note que o PE atualmente usado não é armazenado na pilha de tempo de execução. Apenas versões gravadas são armazenadas como as ligações dinâmicas nas instâncias de registro de ativação.

Discutimos agora várias ações novas no processo de ligação. A lista da Seção 10.2 deve ser revisada para incluí-las. Usando o formato do registro de ativação desta seção, as novas ações são listadas abaixo:

As ações do subprograma chamador são:

1. Criar uma instância de registro de ativação.
2. Gravar o estado da execução da unidade de programa atual.
3. Calcular e passar os parâmetros.
4. Passar o endereço de retorno para o subprograma chamado.
5. Transferir o controle para o subprograma chamado.

As ações do prólogo do subprograma chamado são:

1. Salvar o PE antigo na pilha como a ligação dinâmica e criar o novo valor.
2. Alocar variáveis locais.

As ações do epílogo do subprograma chamado são:

1. Se existirem parâmetros com passagem por valor-resultado ou passagem por modo de saída, mover os valores atuais desses parâmetros para os parâmetros reais correspondentes.
2. Se o subprograma é uma função, mover o valor funcional para um local acessível ao chamador.
3. Restaurar o ponteiro da pilha configurando-o para o valor do PE atual menos um e configurar o PE para a ligação dinâmica antiga.
4. Restaurar o estado de execução do chamador.
5. Transferir o controle de volta para o chamador.

Lembre-se, do Capítulo 9, de que um subprograma está **ativo** desde o momento em que é chamado até o momento em que a execução estiver concluída. No momento em que ele se torna inativo, seu escopo local deixa de existir e seu ambiente de referenciamento não é mais significativo. Portanto, nesse momento, sua instância de registro de ativação pode ser destruída.

Os parâmetros nem sempre são transferidos na pilha. Em muitos compiladores para máquinas RISC, os parâmetros são passados em registradores. Isso porque essas máquinas normalmente têm muito mais registradores do que as máquinas CISC.

No restante deste capítulo, entretanto, supomos que os parâmetros são passados na pilha. É fácil modificar essa abordagem para que os parâmetros sejam passados em registradores.

10.3.2 Um exemplo sem recursão

Considere o seguinte esqueleto de programa em C:

```
void fun1(float r) {
  int s, t;
  . . .    <---------- 1
  fun2(s);
  . . .
}
void fun2(int x) {
  int y;
  . . .    <---------- 2
  fun3(y);
  . .
}
void fun3(int q) {
  . . .    <---------- 3
}
void main() {
  float p;
  . . .
  fun1(p);
  . . .
}
```

A sequência de chamadas de função nesse programa é

```
main chama fun1
fun1 chama fun2
fun2 chama fun3
```

O conteúdo da pilha para os pontos rotulados 1, 2 e 3 é mostrado na Figura 10.5.

No ponto 1, apenas as instâncias de registro de ativação para as funções `main` e `fun1` estão na pilha. Quando `fun1` chama `fun2`, uma instância do registro de ativação de `fun2` é criada na pilha. Quando `fun2` chama `fun3`, uma instância de registro de ativação de `fun3` é criada na pilha. Quando a execução de `fun3` termina, a instância de seu registro de ativação é removida da pilha e o PE é usado para reiniciar o ponteiro do topo da pilha. Processos similares ocorrem quando as funções `fun2` e `fun1` terminam. Após o retorno da chamada a `fun1` a partir de `main`, a pilha tem apenas a instância de registro de ativação de `main`. Note que algumas implementações não usam uma instân-

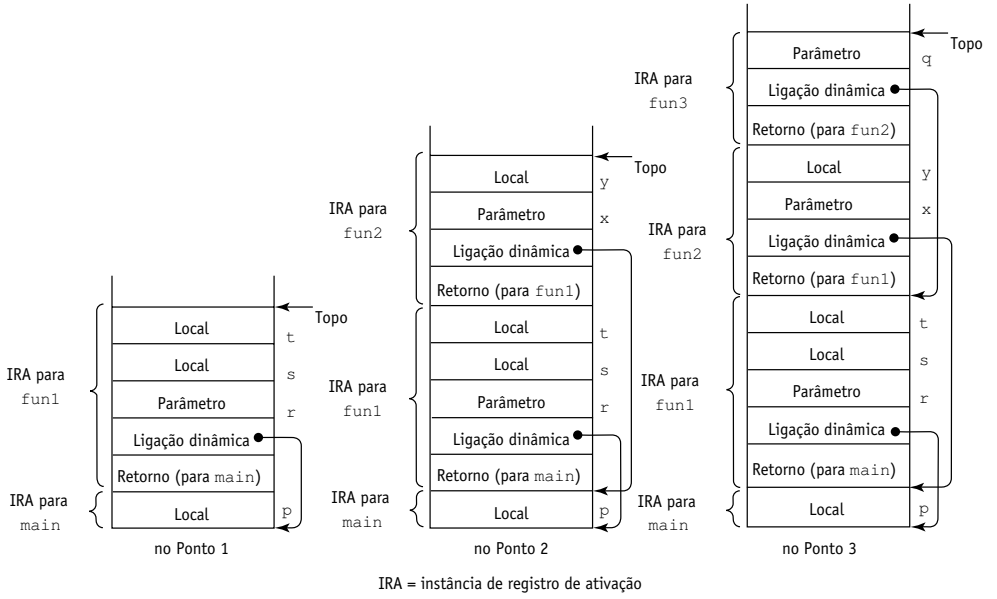


FIGURA 10.5
Conteúdo da pilha para três pontos em um programa.

cia de registro de ativação na pilha para funções `main`, como a mostrada na figura. Entretanto, isso pode ser feito dessa forma, o que simplifica tanto a implementação quanto nossa discussão. Nesse exemplo e em todos os outros deste capítulo, supomos que a pilha cresce do menor para o maior endereço, apesar de, em uma implementação em particular, a pilha poder crescer na direção oposta.

A coleção de ligações dinâmicas presentes na pilha em dado momento é chamada de **cadeia dinâmica** ou de **cadeia de chamadas**. Ela representa a história dinâmica de como a execução chegou à sua posição atual, que está sempre no código do subprograma cuja instância de registro de ativação está no topo da pilha. Referências a variáveis locais podem ser representadas no código como deslocamentos a partir do início do registro de ativação do escopo local, cujo endereço é armazenado no PE. Esse deslocamento é chamado de **deslocamento local** (*local_offset*).

O deslocamento local de uma variável em um registro de ativação pode ser determinado em tempo de compilação, usando a ordem, os tipos e os tamanhos de variáveis declaradas no subprograma associado ao registro. Para simplificar a discussão, supomos que todas as variáveis ocupam uma posição no registro de ativação. A primeira variável local declarada em um subprograma seria alocada no registro de ativação duas posições mais o número de parâmetros a partir da parte inferior (as primeiras duas posições são para o endereço de retorno e para a ligação dinâmica). A segunda variável local declarada estaria uma posição mais próxima ao topo da pilha e assim por diante. Por exemplo, considere o programa de exemplo anterior. Em `fun1`, o deslocamento local de `s` é 3;

para `t`, ele é 4. Da mesma forma, em `fun2`, o deslocamento local de `y` é 3. Para obter o endereço de qualquer variável local, o deslocamento local da variável é adicionado ao PE.

10.3.3 Recursão

Considere este exemplo de programa em C, que usa recursão para calcular a função fatorial:

```
int factorial(int n) {
    <---------- 1
  if (n <= 1)
    return 1;
  else return (n * factorial(n - 1));
    <---------- 2
 }
void main() {
  int value;
  value = factorial(3);
    <---------- 3
 }
```

O formato do registro de ativação para a função `factorial` é mostrado na Figura 10.6. Note que ele tem uma entrada adicional para o valor de retorno da função.

A Figura 10.7 mostra o conteúdo da pilha para as três vezes em que a execução alcança a posição 1 na função `factorial`. Cada uma delas mostra mais uma ativação da função, com seu valor funcional indefinido. A primeira instância do registro de ativação tem o endereço de retorno configurado para a função chamadora, `main`. As outras têm um endereço de retorno para a própria função; essas são para as chamadas recursivas.

A Figura 10.8 mostra o conteúdo da pilha para as três vezes em que a execução alcança a posição 2 na função `factorial`. A posição 2 seria o momento após o retorno (**`return)`** ser executado, mas antes de o registro de ativação ser removido da pilha. Lembre-se de que o código para a função multiplica o valor atual do parâmetro `n` pelo valor retornado pela chamada recursiva para a função. O primeiro retorno de `factorial`

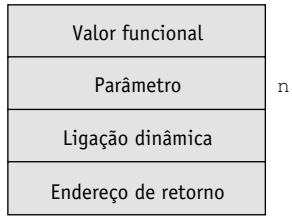


FIGURA 10.6
O registro de ativação para `factorial`.

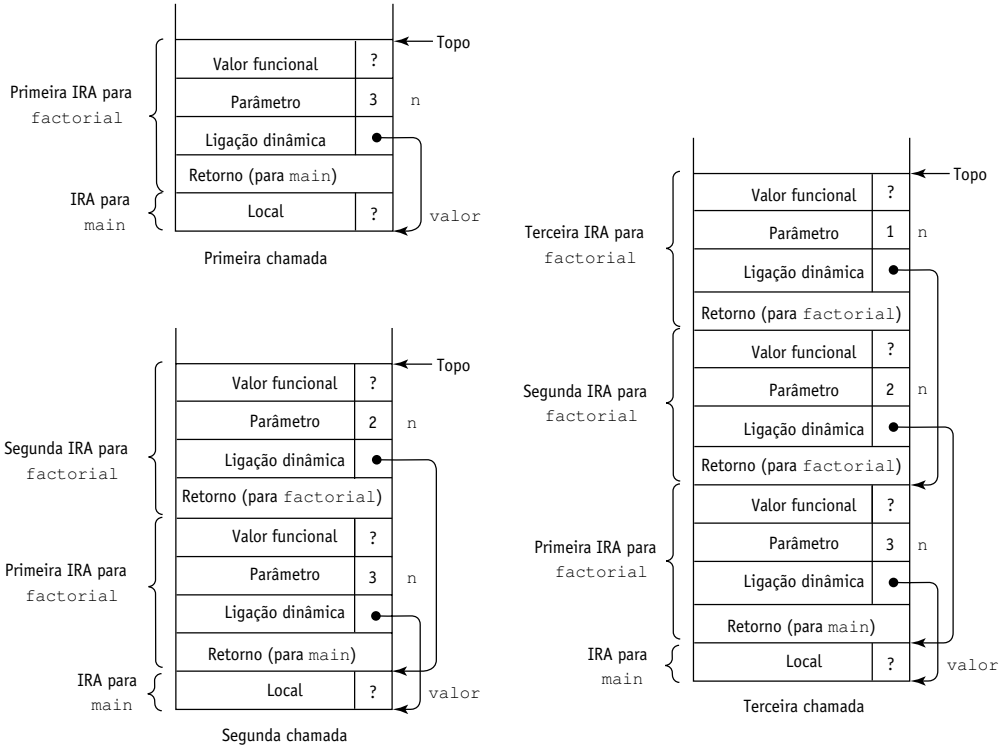


FIGURA 10.7
Conteúdo da pilha na posição 1 de `factorial`.

retorna o valor 1. A instância de registro de ativação para essa ativação tem o valor de 1 para sua versão do parâmetro `n`. O resultado dessa multiplicação, 1, é retornado para a segunda ativação de `factorial` para ser multiplicado por seu valor de parâmetro para `n`, que é 2. Esse passo retorna o valor 2 para a primeira ativação de factorial para ser multiplicado por seu valor de parâmetro para `n`, que é 3, levando ao valor final funcional 6, que é então retornado para a primeira chamada a `factorial` em `main`.

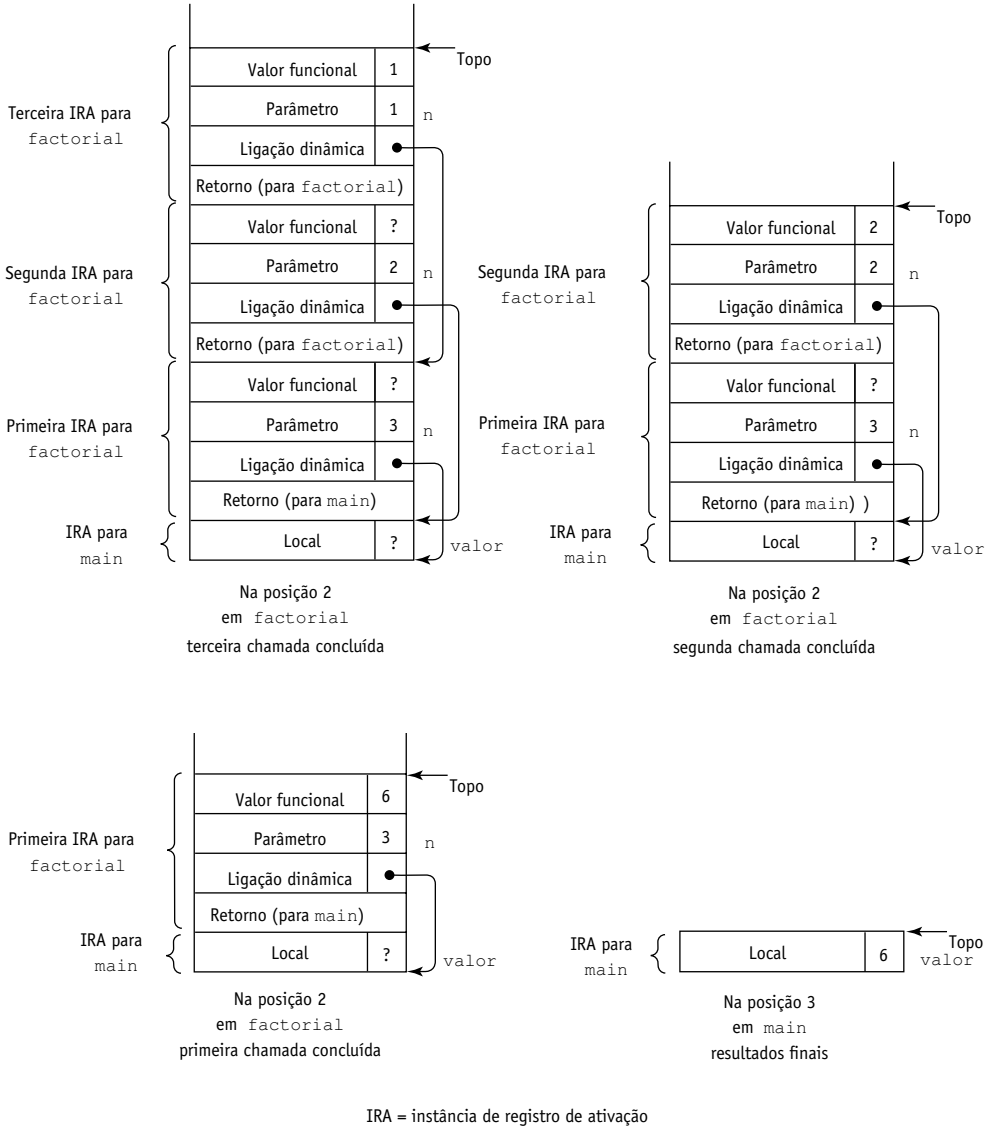


FIGURA 10.8
Conteúdo da pilha durante a execução de `main` e `factorial`.

10.4 SUBPROGRAMAS ANINHADOS

Algumas das linguagens de programação de escopo estático não baseadas em C usam variáveis locais dinâmicas da pilha e permitem que os subprogramas sejam aninhados. Entre elas estão Fortran 95+, Ada, Python, JavaScript, Ruby e Lua, bem como as linguagens funcionais. Nesta seção, examinamos a estratégia mais usada para implementar subprogramas que podem ser aninhados. Até o final desta seção, ignoramos os fechamentos.

10.4.1 Os fundamentos

Uma referência a uma variável não local em uma linguagem com escopo estático com subprogramas aninhados exige um processo de acesso de dois passos. Todas as variáveis não estáticas que podem ser acessadas não localmente estão em instâncias de registro de ativação existentes e, logo, em algum lugar na pilha. O primeiro passo do processo de acesso é encontrar a instância de registro de ativação na pilha em que a variável foi alocada. O segundo é usar o deslocamento local da variável (dentro da instância de registro de ativação) para acessá-la.

Encontrar a instância de registro de ativação correta é o passo mais interessante e mais difícil. Primeiro, note que, em dado subprograma, apenas as variáveis declaradas em escopos estáticos ancestrais são visíveis e podem ser acessadas. Além disso, instâncias de registro de ativação de todos os ancestrais estáticos estão sempre na pilha quando as variáveis contidas nelas são referenciadas por um subprograma aninhado. Isso é garantido pelas regras de semântica estática das linguagens de escopo estático: um subprograma só pode ser chamado quando todos os seus subprogramas ancestrais estáticos estão ativos.[1] Se um ancestral estático não estivesse ativo, suas variáveis locais não estariam vinculadas ao armazenamento, então não faria sentido permitir acesso a elas.

A semântica de referências não locais dita que a declaração correta é a primeira encontrada quando procuramos escopos que envolvem a referência, o mais proximamente aninhado primeiro. Para suportar referências não locais, deve ser possível encontrar na pilha todas as instâncias de registros de ativação que correspondam a esses ancestrais estáticos. Essa observação leva à estratégia de implementação descrita na próxima subseção.

Não tratamos das questões de blocos até a Seção 10.5. No restante desta seção, presume-se que todos os escopos são definidos pelos subprogramas. Como, nas linguagens baseadas em C, funções não podem ser aninhadas (os únicos escopos estáticos nessas linguagens são aqueles criados com blocos), as discussões desta seção não se aplicam diretamente a essas linguagens.

10.4.2 Encadeamentos estáticos

A maneira mais comum de implementar escopo estático em linguagens que permitem que os subprogramas sejam aninhados é pelo uso de encadeamentos estáticos. Nessa estratégia, um novo ponteiro, chamado de ligação estática (*static link*), é adicionado ao registro de ativação. A **ligação estática**, às vezes chamada de *ponteiro de escopo estático*, aponta para o final da instância de registro de ativação de uma ativação do ancestral estático. Ela é usada para acessos a variáveis não locais. Em geral, a ligação estática aparece no registro de ativação abaixo dos parâmetros. A adição da ligação estática ao registro de ativação requer que os deslocamentos locais sejam diferentes daqueles usados quando a ligação estática não é incluída. Em vez de haver dois elementos do registro de ativação antes dos parâmetros, agora existem três: o endereço de retorno, a ligação estática e a ligação dinâmica.

[1] Os fechamentos, evidentemente, contrariam essa regra.

Um **encadeamento estático** é uma cadeia de ligações estáticas que conectam certas instâncias de registro de ativação na pilha. Durante a execução de um subprograma `P`, a ligação estática de sua instância de registro de ativação aponta para uma instância de registro de ativação da unidade de programa do pai estático de `P`. Essa ligação estática da instância aponta, por sua vez, para a instância de registro de ativação da unidade de programa do avô estático de `P`, se existir um. Então, o encadeamento estático conecta todos os ancestrais estáticos de um subprograma executado, começando pelo seu pai estático. Esse encadeamento pode, obviamente, ser usado para implementar os acessos a variáveis não locais em linguagens de escopo estático.

Encontrar a instância de registro de ativação correta de uma variável não local usando ligações estáticas é relativamente simples. Quando é feita uma referência a uma variável não local, a instância de registro de ativação que contém a variável pode ser localizada por meio de uma busca no encadeamento estático, que ocorre até ser encontrada uma instância de registro de ativação de um ancestral estático que contenha a variável. Entretanto, esse processo pode ser muito mais simples. Como o encadeamento de escopos é conhecido em tempo de compilação, o compilador pode determinar não apenas que uma referência é não local, mas também o tamanho do encadeamento estático que deve ser seguido para alcançar a instância de registro de ativação que contém o objeto não local.

Considere a **profundidade estática** (*static_depth*) como um inteiro associado a um escopo estático que indica o quão profundamente ele está aninhado no escopo mais externo. Uma unidade de programa que não está aninhada dentro de qualquer outra unidade tem profundidade estática 0. Se o subprograma `A` é definido em uma unidade de programa não aninhada, sua profundidade estática é 1. Se ele contém a definição de um subprograma aninhado `B`, a profundidade estática de `B` é 2.

O tamanho do encadeamento estático necessário para alcançar a instância de registro de ativação correta para uma referência não local para uma variável `X` é exatamente a diferença entre a profundidade estática do subprograma que contém a referência a `X` e a profundidade estática do subprograma que contém a declaração de `X`. Essa diferença é chamada de **profundidade de aninhamento** (*nesting_depth*) ou **deslocamento de encadeamento** (*chain_offset*) da referência. A referência real pode ser representada por um par ordenado de inteiros (deslocamento de encadeamento, deslocamento local), no qual o deslocamento de encadeamento é o número de ligações para a instância de registro de ativação correta (o deslocamento local é descrito na Seção 10.3.2). Por exemplo, considere o seguinte esqueleto de programa em Python:

```
# Escopo global
. . .
def f1():
  def f2():
    def f3():
      . . .
    # fim de f3
    . . .
  # fim de f2
  . .
# fim de f1
```

As profundidades estáticas do escopo global, `f1`, `f2` e `f3`, são 0, 1, 2 e 3, respectivamente. Se o procedimento `f3` referenciasse uma variável declarada em `f1`, o deslocamento de encadeamento dessa referência seria 2 (profundidade estática de `f3` menos profundidade estática de `f1`). Se o procedimento `f3` referenciasse uma variável declarada em `f2`, o deslocamento de encadeamento dessa referência seria 1. Referências a variáveis locais podem ser manipuladas por meio do mesmo mecanismo, com um deslocamento de encadeamento 0; mas, em vez de usar o ponteiro estático para a instância de registro de ativação do subprograma onde a variável foi declarada como o endereço base, o PE é usado.

Para ilustrar o processo completo de acessos a variáveis não locais, considere o seguinte esqueleto de programa em JavaScript:

```
function main(){
  var x;
  function bigsub() {
    var a, b, c;
    function sub1 {
      var a, d;
      ...
      a = b + c; <----------------------------------1
      ...
    }  // fim de sub1
    function sub2(x) {
      var b, e;
      function sub3() {
        var c, e;
        ...
        sub1();
        ...
        e = b + a; <----------------------------------2
      } // fim de sub3
      ...
      sub3();
      ...
      a = d + e; <----------------------------------3
    } // fim de sub2
    ...
    sub2(7);
    ...
  } // fim de bigsub
  ...
  bigsub();
  ...
} // fim de main
```

A sequência de chamadas a procedimentos é

`main` chama `bigsub`
`bigsub` chama `sub2`
`sub2` chama `sub3`
`sub3` chama `sub1`

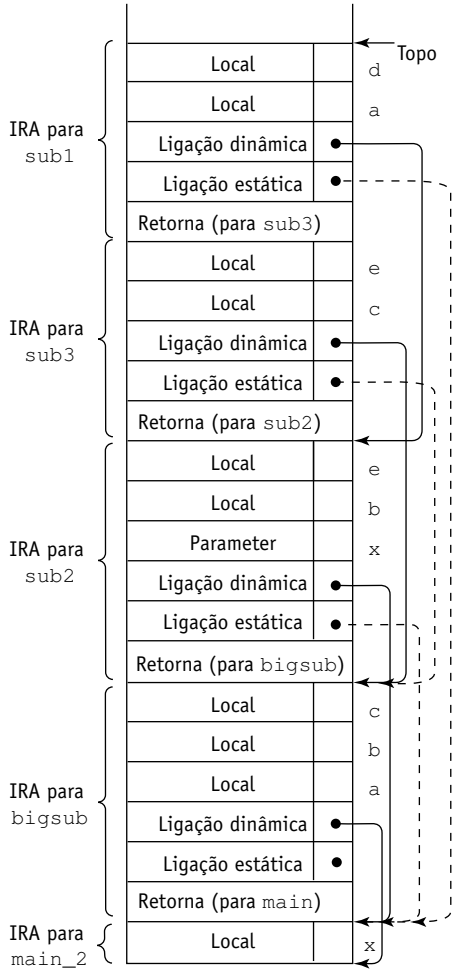


FIGURA 10.9
Conteúdo da pilha na posição 1 do programa `main`.

A situação da pilha quando a execução chega pela primeira vez ao ponto 1 desse programa é mostrada na Figura 10.9.

Na posição 1 do procedimento `sub1`, a referência é para a variável local `a`, não para a variável não local `a` de `bigsub`. Essa referência a `a` tem o par deslocamento de encadeamento/deslocamento local (0,3). A referência a `b` é para a variável não local `b` de `bigsub`. Ela pode ser representada pelo par (1,4). O deslocamento local é 4, porque um deslocamento 3 seria a primeira variável local (`bigsub` não tem parâmetros). Note que, se a ligação dinâmica fosse usada para realizar uma busca simples por uma instância de registro de ativação com uma declaração para a variável `b`, ela encontraria

a variável b declarada em sub2, o que seria incorreto. Se o par (1,4) fosse usado com o encadeamento dinâmico, a variável e de sub3 seria utilizada. A ligação estática, entretanto, aponta para o registro de ativação de bigsub, que tem a versão correta de b. A variável b em sub2 não está no ambiente de referenciamento nesse momento e está (corretamente) inacessível. A referência a c no ponto 1 é para o c definido em bigsub, representado pelo par (1,5).

Após sub1 completar sua execução, a instância de registro de ativação para sub1 é removida da pilha e o controle retorna para sub3. A referência para a variável e na posição 2 em sub3 é local e usa o par (0,4) para o acesso. A referência à variável b é para aquela declarada em sub2, porque ele é o ancestral estático mais próximo que contém tal declaração. Ela é acessada com o par (1,4). O deslocamento local é 4 porque b é a primeira variável declarada em sub1, e sub2 tem apenas um parâmetro. A referência à variável a é para o a declarado em bigsub, porque nem sub3 nem seu pai estático sub2 têm uma declaração para uma variável chamada a. Ela é referenciada pelo par (2,3).

Após sub3 completar sua execução, a instância de registro de ativação para sub3 é removida da pilha, deixando apenas instâncias de registro de ativação para main, bigsub e sub2. Na posição 3 em sub2, a referência à variável a é para o a em bigsub, o qual tem a única declaração de a entre as rotinas ativas. Esse acesso é feito com o par (1,3). Nessa posição, não existe um escopo visível contendo uma declaração para a variável d; então, essa referência a d é um erro de semântica estática. O erro seria detectado quando o compilador tentasse computar o par deslocamento de encadeamento/deslocamento local. A referência para e é para o e local em sub2; ela pode ser acessada com o par (0,5).

Em resumo, as referências para as variáveis a nos pontos 1, 2 e 3 seriam representadas pelos pontos:

- (0, 3) (local)
- (2, 3) (dois níveis de distância)
- (1, 3) (um nível de distância)

Neste ponto, é razoável perguntar como o encadeamento estático é mantido durante a execução de um programa. Se sua manutenção for muito complexa, o fato de ele ser simples e eficaz não será importante. Supomos aqui que os parâmetros que são subprogramas não são implementados.

A cadeia estática deve ser modificada para cada chamada e retorno a um subprograma. A parte do retorno é trivial: quando o subprograma termina, sua instância de registro de ativação é removida da pilha. Após essa remoção, a nova instância de registro de ativação no topo é da unidade que chamou o subprograma cuja execução recém terminou. Como o encadeamento estático para essa instância de registro de ativação nunca mudou, ele funciona corretamente, da mesma forma que faria antes da chamada ao outro subprograma. Logo, nenhuma outra ação é necessária.

A ação necessária na chamada a um subprograma é mais complexa. Apesar de o escopo pai correto ser facilmente determinado em tempo de compilação, sua instância de registro de ativação mais recente deve ser encontrada no momento da chamada. Isso pode ser feito buscando-se as instâncias de registro de ativação no encadeamento dinâmico até que a primeira do escopo pai seja encontrada. Entretanto, essa busca pode ser evitada se as declarações e referências a subprograma forem tratadas exatamente

como declarações e referências a variáveis. Quando o compilador encontra uma chamada a um subprograma, entre outras coisas, ele determina o subprograma que declarou o subprograma chamado, que deve ser um ancestral estático da rotina chamadora. Ele então calcula a profundidade de aninhamento, ou número de escopos que existem entre o chamador e o subprograma que declarou o subprograma chamado. Essa informação é armazenada e pode ser acessada pela chamada a subprograma durante a execução. No momento da chamada, a ligação estática da instância de registro de ativação do subprograma é determinada por uma descida no encadeamento estático do chamador. O número de ligações nessa descida é igual à profundidade de aninhamento calculada em tempo de compilação.

Considere novamente o programa `main` e a situação da pilha mostrada na Figura 10.9. Na chamada a `sub1` em `sub3`, o compilador define a profundidade de aninhamento de `sub3` (o chamador) como sendo dois níveis dentro do procedimento que declarou `sub1`, que é `bigsub`. Quando a chamada a `sub1` em `sub3` é executada, essa informação é usada para configurar a ligação estática da instância de registro de ativação para `sub1`. Essa ligação estática é configurada de modo a apontar para a instância de registro de ativação apontada pela segunda ligação estática no encadeamento estático, a partir da instância de registro de ativação do chamador. Nesse caso, o chamador é `sub3`, cuja ligação estática aponta para a instância de registro de ativação de seu pai (aquela de `sub2`). A ligação estática da instância de registro de ativação para `sub3` aponta para a instância de registro de ativação de `bigsub`. Então, a ligação estática para a nova instância de registro de ativação de `sub1` é configurada de modo a apontar para a instância de registro de ativação de `bigsub`.

Esse método funciona para todas as ligações de subprogramas, exceto quando estão envolvidos parâmetros que são subprogramas.

Uma crítica ao uso da estratégia de encadeamento estático para acessar variáveis não locais é que referências a variáveis em escopos além do pai estático são mais dispendiosas que referências a variáveis não locais. O encadeamento estático deve ser seguido: uma ligação para cada escopo envolvente a partir da referência até a declaração. Felizmente, na prática as referências a variáveis não locais distantes são raras, então esse não é um problema sério. Outra crítica à estratégia baseada em encadeamento estático é a dificuldade que um programador, trabalhando em um programa com limitação de tempo, tem para estimar os custos de referências não locais, porque o custo de cada referência depende da profundidade de aninhamento entre a referência e o escopo da declaração. Complicando ainda mais esse problema está o fato de que modificações de código subsequentes podem mudar as profundidades de aninhamento, modificando o tempo de algumas referências, tanto no código modificado quanto possivelmente no código distante das mudanças.

Algumas alternativas ao encadeamento estático foram desenvolvidas, em especial uma estratégia que usa uma estrutura de dados auxiliar, chamada **mostrador** (*display*). Entretanto, nenhuma das alternativas se mostrou superior ao método de encadeamento estático, ainda a estratégia mais usada. Logo, nenhuma delas é discutida aqui.

Os processos e as estruturas de dados descritos nesta seção implementam fechamentos corretamente em linguagens que não permitem que funções retornem funções e não deixam que funções sejam atribuídas a variáveis. Contudo, são inadequados para linguagens que permitem uma ou ambas dessas operações. Vários mecanismos novos são necessários para implementar o acesso a variáveis não locais nessas linguagens. Primeiro, se um subprograma acessa uma variável de um aninhamento, mas não do escopo

global, essa variável não pode ser armazenada apenas no registro de ativação de seu escopo inicial. Esse registro de ativação poderia ser liberado antes que o subprograma que precisasse dele fosse ativado. Tais variáveis também poderiam ser armazenadas no monte e receber extensão ilimitada (seus tempos de vida são o tempo de vida do programa inteiro). Segundo, os subprogramas devem ter mecanismos para acessar as variáveis não locais que são armazenadas no monte. Terceiro, as variáveis alocadas no monte que não são acessadas localmente devem ser atualizadas sempre que suas versões da pilha forem atualizadas. Claramente, essas são extensões não triviais para a implementação de escopo estático por meio de encadeamentos estáticos.

10.5 BLOCOS

Lembre-se, do Capítulo 5, de que diversas linguagens, incluindo as baseadas em C, fornecem escopos locais especificados pelo usuário para variáveis, chamados de **blocos**. Como exemplo de um bloco, considere o seguinte segmento de código:

```
{ int temp;
  temp = list[upper];
  list[upper] = list[lower];
  list[lower] = temp;
}
```

Um bloco é especificado nas linguagens baseadas em C como uma sentença composta que começa com uma ou mais definições de dados. O tempo de vida da variável `temp` no bloco anteriormente mostrado começa quando o controle entra no bloco e termina quando ele sai do bloco. A vantagem de usar tal variável local é que ela não pode interferir em outras variáveis com o mesmo nome declaradas em outros lugares do programa ou, mais especificamente, no ambiente de referenciamento do bloco.

Os blocos podem ser implementados pelo uso do processo de encadeamento estático, descrito na Seção 10.4 e usado para implementar subprogramas aninhados. Blocos são tratados como subprogramas sem parâmetros, sempre chamados a partir do mesmo local do programa. Logo, cada bloco tem um registro de ativação. Uma instância de seu registro de ativação é criada a cada vez que o bloco é executado.

Os blocos também podem ser implementados de uma maneira mais simples e eficiente. A quantidade máxima de armazenamento necessária para variáveis de um bloco em qualquer momento durante a execução de um programa pode ser determinada estaticamente, porque os blocos são acessados (entrada e saída) em ordem estritamente textual. Essa quantidade de espaço pode ser alocada após as variáveis locais no registro de ativação. Deslocamentos para todas as variáveis de bloco podem ser estaticamente computados; então, as variáveis de bloco podem ser endereçadas exatamente como se fossem variáveis locais.

Por exemplo, considere o esqueleto de programa:

```
void main() {
  int x, y, z;
  while ( . .  ) {
    int a, b, c;
    .
```

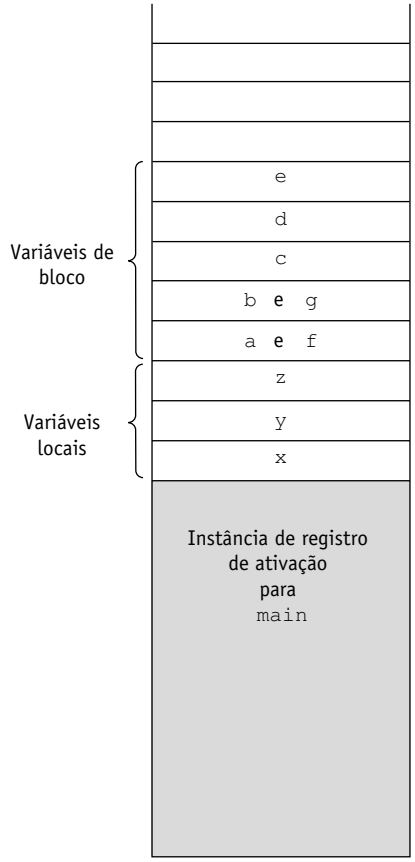


FIGURA 10.10
Armazenamento de variáveis de blocos quando os blocos não são tratados como procedimentos sem parâmetros.

```
    while ( . . . ) {
      int d, e;
      .
    }
  }
  while ( . . . ) {
    int f, g;
    . . .
  }
  . .
}
```

Para esse programa, o layout de memória estática mostrado na Figura 10.10 pode ser usado. Note que `f` e `g` ocupam as mesmas posições de memória que `a` e `b`, porque estas são retiradas da pilha quando o controle deixa seu bloco (antes de `f` e `g` serem alocadas).

10.6 IMPLEMENTAÇÃO DE ESCOPO DINÂMICO

Existem ao menos duas maneiras pelas quais variáveis locais e referências não locais a elas podem ser implementadas em uma linguagem de escopo dinâmico: acesso profundo (*deep access*) e acesso raso (*shallow access*). Note que os dois processos não são conceitos relacionados à vinculação profunda e rasa. Uma diferença importante entre a vinculação e o acesso é que as vinculações profundas e rasas resultam em semânticas diferentes, enquanto os acessos profundos e os rasos não.

10.6.1 Acesso profundo

Se as variáveis locais são dinâmicas da pilha e fazem parte dos registros de ativação de uma linguagem de escopo dinâmico, as referências a variáveis não locais podem ser resolvidas com buscas por meio das instâncias de registros de ativação dos outros subprogramas atualmente ativos, iniciando com aquele ativado mais recentemente. Esse conceito é similar àquele de acessar variáveis não locais em uma linguagem de escopo estático com subprogramas aninhados, exceto pelo fato de ser seguido o encadeamento dinâmico – em vez do estático. O encadeamento dinâmico liga todas as instâncias de registro de ativação na ordem inversa pela qual foram ativadas. Logo, ele é exatamente o necessário para referenciar variáveis não locais em uma linguagem de escopo dinâmico. Esse método é chamado de **acesso profundo**, porque pode exigir buscas profundas na pilha.

Considere o seguinte exemplo de esqueleto de programa:

```
void sub3() {
  int x, z;
  x = u + v;
  . . .
}
void sub2() {
  int w, x;
  . . .
}
void sub1() {
  int v, w;
  . . .
}
void main() {
  int v, u;
  . . .
}
```

Esse programa está escrito em uma sintaxe que dá a ele a aparência de uma linguagem baseada em C, mas não é de uma linguagem em particular. Suponha que a seguinte sequência de chamadas a funções ocorra:

```
main chama sub1
sub1 chama sub1
sub1 chama sub2
sub2 chama sub3
```

A Figura 10.11 mostra a pilha durante a execução da função `sub3` após essa sequência de chamadas. Note que as instâncias de registro de ativação não têm ligações estáticas, sem função em uma linguagem de escopo dinâmico.

Considere as referências às variáveis `x`, `u` e `v` na função `sub3`. A referência a `x` é encontrada na instância de registro de ativação de `sub3`. A referência a `u` é encontrada pela busca por *todas* as instâncias de registro de ativação na pilha, já que a única variável existente com esse nome está em `main`. Essa busca envolve seguir quatro ligações dinâmicas e examinar 10 nomes de variáveis. A referência a `v` é encontrada na instância de registro de ativação mais recente (mais próxima no encadeamento dinâmico) do subprograma `sub1`.

Existem duas diferenças importantes entre o método de acesso profundo para acesso a variáveis não locais em uma linguagem de escopo dinâmico e o método de encadeamento estático para linguagens de escopo estático. Primeiro, em uma linguagem de escopo dinâmico, não existe uma forma de determinar, em tempo de compilação, o tamanho da cadeia que precisará ser seguida. Cada instância de registro de ativação na cadeia precisa ser buscada até que a primeira instância da variável seja encontrada. Essa é uma das razões pelas quais as linguagens de escopo dinâmico têm velocidades

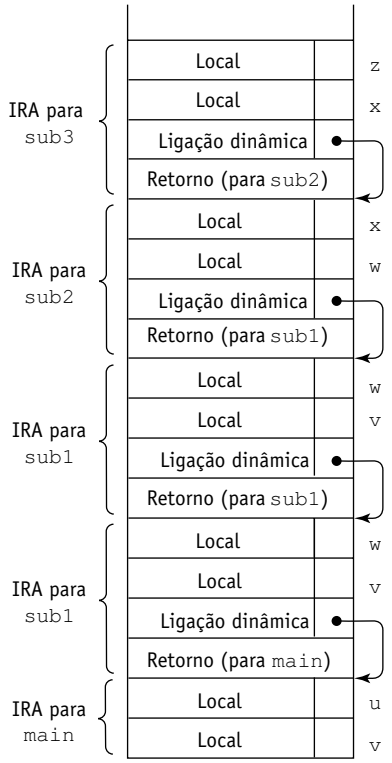


FIGURA 10.11
Conteúdo da pilha para um programa de escopo dinâmico.

de execução mais lentas que as linguagens de escopo estático. Segundo, os registros de ativação devem armazenar os nomes das variáveis para o processo de busca, enquanto nas implementações de linguagens com escopo estático apenas os valores são necessários. (Os nomes não são necessários para escopo estático, porque todas as variáveis são representadas pelos pares "deslocamento de encadeamento/deslocamento local".)

10.6.2 Acesso raso

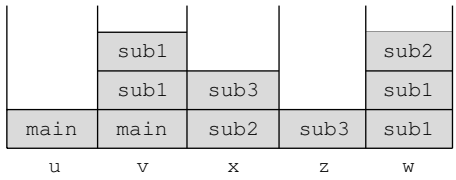
O acesso raso é um método de implementação alternativo, não uma semântica alternativa. Conforme dito anteriormente, a semântica do acesso profundo é idêntica à do acesso raso. No método de acesso raso, as variáveis declaradas em subprogramas não são armazenadas nos registros de ativação desses subprogramas. Como com o escopo dinâmico existe, no máximo, uma versão visível de uma variável de um nome específico em um momento do tempo, uma abordagem bastante diferente pode ser usada. Uma variação do acesso raso é ter uma pilha separada para cada nome de variável em um programa completo. Cada vez que uma variável com determinado nome é criada por uma declaração no início de um subprograma chamado, é dada uma célula no topo da pilha para o nome da variável. Cada referência ao nome é para a variável no topo da pilha associada a tal nome, porque a variável do topo é a mais recentemente criada. Quando um subprograma termina, o tempo de vida de suas variáveis locais termina e as pilhas para esses nomes de variáveis têm tais nomes removidos. Esse método permite referências rápidas a variáveis, mas manter as pilhas nas entradas e saídas de subprogramas é dispendioso.

A Figura 10.12 mostra as pilhas de variáveis para o programa de exemplo anteriormente apresentado, na mesma situação mostrada na pilha da Figura 10.11.

(Os nomes nas células da pilha indicam as unidades de programa da declaração da variável.)

Outra opção para implementar acesso raso é usar uma tabela central com uma posição para cada nome diferente de variável em um programa. Com cada entrada, é mantido um bit chamado **ativo** para indicar se o nome tem uma vinculação ou associação atual a uma variável. Qualquer acesso a qualquer variável pode então ser para um deslocamento na tabela central. O deslocamento é estático; portanto, o acesso pode ser rápido. Implementações de SNOBOL usam a técnica de implementação com tabela central.

A manutenção de uma tabela central é algo bastante simples. Uma chamada a um subprograma requer que todas as suas variáveis locais sejam colocadas logicamente na tabela central. Se a posição da nova variável na tabela central já está ativa – ou seja, se



(Os nomes nas células da pilha indicam as unidades de programa da declaração da variável.)

FIGURA 10.12
Um método de uso do acesso raso para implementar escopo dinâmico.

ela contém uma variável cujo tempo de vida ainda não acabou (o que é indicado pelo bit ativo) –, tal valor deve ser gravado em algum lugar durante o tempo de vida da nova variável. Sempre que uma variável começa seu tempo de vida, o bit de ativação em sua posição na tabela central deve ser ativado.

Existem diversas variações no projeto da tabela central e na maneira pela qual os valores são armazenados quando são temporariamente substituídos. Uma variação é ter uma pilha "oculta", na qual todos os objetos gravados são armazenados. Como as chamadas e retornos a subprogramas e, logo, os tempos de vida das variáveis locais são aninhados, isso funciona bem.

A segunda variação é, talvez, a mais clara e menos dispendiosa de implementar. Uma tabela central de células únicas é usada, armazenando apenas a versão atual de cada variável com um nome único. Variáveis substituídas são armazenadas no registro de ativação do subprograma que criou a variável de substituição. Esse é um mecanismo de pilha, mas ele usa a pilha já existente; portanto, a nova sobrecarga é mínima.

A escolha entre o acesso raso e o profundo para as variáveis não locais depende da frequência relativa das chamadas a subprogramas e referências não locais. O método de acesso profundo fornece ligação rápida de subprogramas, mas as referências às variáveis não locais, especialmente aquelas às variáveis não locais distantes (em termos do encadeamento de chamadas) são dispendiosas. O método de acesso raso fornece referências muito mais rápidas para variáveis não locais, principalmente para variáveis não locais distantes, mas é mais custoso em termos de ligação de subprogramas.

RESUMO

A semântica de ligação de subprogramas requer muitas ações por parte da implementação. No caso de subprogramas "simples", essas ações são relativamente descomplicadas. Na chamada, o estado da execução deve ser gravado, os parâmetros e o endereço de retorno devem ser passados para o subprograma chamado e o controle deve ser transferido. No retorno, os valores de parâmetros com passagem por resultado e passagem por valor-resultado devem ser devolvidos – assim como o valor de retorno (se for uma função) –, o estado de execução deve ser restaurado e o controle devolvido para o chamador. Em linguagens com variáveis locais dinâmicas da pilha e subprogramas aninhados, a ligação de subprogramas é mais complexa. Pode existir mais de uma instância de registro de ativação, essas instâncias devem ser armazenadas na pilha de tempo de execução e ligações estáticas e dinâmicas devem ser mantidas nas instâncias de registro de ativação. A ligação estática é usada para permitir referências a variáveis não locais em linguagens de escopo estático.

Subprogramas em linguagens com variáveis locais dinâmicas da pilha e subprogramas aninhados têm dois componentes: o código real (estático) e o registro de ativação (dinâmico da pilha). Instâncias de registro de ativação contêm os parâmetros formais e as variáveis locais, entre outras coisas.

O acesso às variáveis não locais em uma linguagem de escopo estático pode ser implementado pelo uso de encadeamento dinâmico ou por meio de algum método de tabela variável central. Encadeamentos dinâmicos fornecem acessos lentos, mas chamadas e retornos rápidos. Os métodos de tabela central fornecem acessos rápidos, mas chamadas e retornos lentos.

QUESTÕES DE REVISÃO

1. Qual é a definição usada neste capítulo para subprogramas "simples"?
2. Qual dos dois – chamador ou chamado – grava informações acerca do estado da execução?
3. O que deve ser armazenado para a ligação a um subprograma?
4. Qual é a tarefa de um ligador?
5. Quais são as duas razões pelas quais implementar subprogramas com variáveis locais dinâmicas da pilha é mais difícil do que implementar subprogramas simples?
6. Qual a diferença entre um registro de ativação e uma instância de registro de ativação?
7. Por que o endereço de retorno, a ligação dinâmica e os parâmetros são colocados na parte inferior do registro de ativação?
8. Que tipos de máquinas geralmente usam registradores para passagem de parâmetros?
9. Quais são os dois passos para localizarmos uma variável não local em uma linguagem de escopo estático com variáveis locais dinâmicas da pilha e subprogramas aninhados?
10. Defina *encadeamento estático*, *profundidade estática*, *profundidade de aninhamento* e *deslocamento de encadeamento*.
11. O que é um PE e qual o seu propósito?
12. Como as referências a variáveis são representadas no método de encadeamento estático?
13. Cite três linguagens de programação bastante usadas que não permitem subprogramas aninhados.
14. Cite dois problemas em potencial com o método de encadeamento estático.
15. Explique os dois métodos de implementar blocos.
16. Descreva o método de acesso profundo de implementação de escopo dinâmico.
17. Descreva o método de acesso raso de implementação de escopo dinâmico.
18. Quais são as duas diferenças entre o método de acesso profundo para acesso a variáveis não locais em linguagens de escopo dinâmico e o método de encadeamento estático para linguagens de escopo estático?
19. Compare a eficiência do método de acesso profundo com a do método de acesso raso, em termos de chamadas e de acesso a variáveis não locais.

PROBLEMAS

1. Mostre a pilha com todas as instâncias de registro de ativação, incluindo o encadeamento estático e o dinâmico, quando a execução alcança a posição 1 no seguinte esqueleto de programa. Suponha que `bigsub` está no nível 1.

```
function bigsub() {
   function a() {
      function b() {
         ... <--------------------------1
      } // fim de b
      function  c() {
        ...
        b();
        ...
      } // fim de c
      ...
      c();
      ...
  } // fim de a
  ...
  a();
  ...
} // fim de bigsub
```

2. Mostre a pilha com todas as instâncias de registro de ativação, incluindo o encadeamento estático e o dinâmico, quando a execução alcança a posição 1 no seguinte esqueleto de programa. Suponha que `bigsub` está no nível 1.

```
function bigsub() {
var mysum;
function a() {
 var x;
  function b(sum) {
   var y, z;
   ...
   c(z);
   ...
 } // fim de b
 ...
 b(x);
  ...
 } // fim de a
 function c(plums) {
  ... <-----------------------------1
 } // fim de c
 var  l;
 ...
 a();
 ...
} // fim de bigsub
```

3. Mostre a pilha com todas as instâncias de registro de ativação, incluindo o encadeamento estático e o dinâmico, quando a execução alcança a posição 1 no seguinte esqueleto de programa. Suponha que `bigsub` está no nível 1.

```
function bigsub() {
function a(flag) {
 function b() {
  ... a(false);
 ...
 } // fim de b
 ...
 if (flag)
  b();
 else c();
...
} // fim de a
function c() {
 function d() {
  ... <------------------------1
 } // fim de d
 ...
 d();
 ...
} // fim de c
...
a(true);
...
} // fim de bigsub
```

A sequência de chamadas para esse programa para que a execução alcance `d` é

`bigsub` chama `a`
`a` chama `b`
`b` chama `a`
`a` chama `c`
`c` chama `d`

4. Mostre a pilha com todas as instâncias de registro de ativação, incluindo o encadeamento estático e o dinâmico, quando a execução alcança a posição 1 no seguinte esqueleto de programa. Esse programa usa o método de acesso profundo para implementar o escopo dinâmico.

```
void fun1() {
  float a;
  . . .
}

void fun2() {
  int b, c;
  . . .
```

```
}

void fun3() {
  float d;
  . . . <--------- 1
}

void main() {
  char e, f, g;
   . . .
}
```

A sequência de chamadas para esse programa para que a execução alcance `fun3` é

`main` chama `fun2`
`fun2` chama `fun1`
`fun1` chama `fun1`
`fun1` chama `fun3`

5. Suponha que o programa do Problema 4 seja implementado por meio do método de acesso raso, com uma pilha para cada nome de variável. Mostre as pilhas no momento da execução de `fun3`, supondo que a execução ocorre até tal ponto por meio da sequência de chamadas mostrada no Problema 4.

6. Apesar de as variáveis locais em métodos Java serem alocadas dinamicamente no início de cada ativação, em quais circunstâncias o valor de uma variável local em determinada ativação em poderia reter o valor da ativação prévia?

7. É afirmado neste capítulo que, quando as variáveis não locais são acessadas em uma linguagem de escopo dinâmico por meio de encadeamento dinâmico, os nomes das variáveis devem ser armazenados nos registros de ativação com os valores. Se isso fosse feito na prática, cada acesso a uma variável não local exigiria uma sequência de dispendiosas comparações de cadeias representando nomes. Projete uma alternativa mais rápida a essas comparações de nomes.

8. Pascal permite gotos com alvos não locais. Como tais sentenças poderiam ser manipuladas se fossem usados encadeamentos estáticos para o acesso a variáveis não locais? *Dica*: considere a maneira pela qual a instância de registro de ativação correta do ancestral estático de um procedimento recentemente chamado é encontrada (consulte a Seção 10.4.2).

9. O método de encadeamento estático pode ser expandido ligeiramente com o uso de duas ligações estáticas em cada instância de registro de ativação, onde a segunda aponta para a instância de registro de ativação do avô estático. Como essa estratégia afetaria o tempo necessário para a ligação de subprogramas e referências a variáveis não locais?

10. Projete um esqueleto de programa e uma sequência de chamadas que resulte em uma instância de registro de ativação na qual as ligações estáticas e dinâmicas apontem para diferentes instâncias de registro de ativação na pilha de tempo de execução.

11. Se um compilador usa a estratégia de encadeamento estático para implementar blocos, quais das entradas nos registros de ativação para subprogramas são necessárias nos registros de ativação para blocos?

12. Examine as instruções de chamadas a subprogramas de três arquiteturas diferentes, incluindo ao menos uma máquina CISC e uma máquina RISC, e escreva uma breve comparação de suas capacidades. (O projeto dessas instruções normalmente determina, em parte, o projeto de ligação de subprogramas do escritor de compiladores.)

EXERCÍCIOS DE PROGRAMAÇÃO

1. Escreva um programa que inclua dois subprogramas, um que receba um único parâmetro e que realize alguma operação simples com esse parâmetro e outro que receba 20 parâmetros e use todos, mas apenas para uma operação simples. O programa principal deve chamar esses dois subprogramas muitas vezes. Inclua no programa o código para cronometrar e mostrar o tempo de execução das chamadas de cada um dos subprogramas. Execute o programa em uma máquina RISC e em uma máquina CISC e compare as taxas de tempo necessárias pos dois subprogramas. Com base nesses resultados, o que você pode dizer acerca da velocidade de passagem de parâmetros nas duas máquinas?

11

Tipos de dados abstratos e construções de encapsulamento

11.1 O conceito de abstração

11.2 Introdução à abstração de dados

11.3 Questões de projeto para tipos de dados abstratos

11.4 Exemplos de linguagem

11.5 Tipos de dados abstratos parametrizados

11.6 Construções de encapsulamento

11.7 Nomeação de encapsulamentos

Neste capítulo, exploramos as construções de linguagens de programação que suportam abstrações de dados. Entre as ideias surgidas nos últimos 50 anos nas metodologias e no projeto de linguagens de programação, a abstração de dados é uma das mais interessantes.

Começamos discutindo o conceito geral de abstração em programação e em linguagens de programação. A abstração de dados é então definida e ilustrada com um exemplo. Esse tópico é seguido por descrições do suporte para abstrações de dados em C++, Objective-C, Java, C# e Ruby. Para esclarecer as semelhanças e diferenças no projeto dos recursos da linguagem que suportam abstração de dados, são dadas implementações do mesmo exemplo de abstração em C++, Objective-C, Java e Ruby. A seguir, são discutidos os recursos de C++, Java 5.0 e C# 2005 para construir tipos de dados abstratos parametrizados.

Todas as linguagens utilizadas neste capítulo para ilustrar os conceitos e as construções de tipos de dados abstratos suportam programação orientada a objetos, pois praticamente todas as linguagens contemporâneas suportam programação orientada a objetos e quase todas as que não suportam e, ainda assim, aceitam tipos de dados abstratos, desapareceram.

Construções que suportam tipos de dados abstratos são encapsulamentos dos dados de operações em objetos do tipo. Encapsulamentos que contêm múltiplos tipos são necessários para a construção de programas maiores. Esses encapsulamentos e as questões associadas aos espaços de nomes também são discutidos neste capítulo.

Algumas linguagens de programação suportam encapsulamentos lógicos, não físicos, os quais são utilizados para encapsular nomes. Isso está discutido na Seção 11.7.

11.1 O CONCEITO DE ABSTRAÇÃO

Uma **abstração** é uma visão ou representação de uma entidade que inclui apenas os atributos mais significativos. De um modo geral, a abstração permite que alguém colete exemplares de entidades em grupos nos quais seus atributos comuns não precisam ser considerados. Por exemplo, suponha que definíssemos aves como criaturas com os seguintes atributos: duas asas, duas pernas, um rabo e penas. Então, se dissermos que um corvo é uma ave, uma descrição de um corvo não precisará incluir esses atributos. O mesmo ocorre para os pintarroxos, pardais e pica-paus de barriga amarela. Os atributos comuns nas descrições de espécies específicas de pássaros podem ser abstraídos, pois todas as espécies os têm. Dentro de uma espécie em particular, apenas os atributos que a diferenciam precisam ser considerados. Por exemplo, corvos têm os atributos de serem pretos, terem determinado tamanho e serem barulhentos. A descrição de um corvo precisa fornecer esses atributos, mas não os outros que são comuns a todas as aves. Isso resulta em uma simplificação significativa das descrições dos membros da espécie. Uma visão menos abstrata de uma espécie, aquela de um pássaro, pode ser considerada quando for necessário ver um alto nível de detalhes, em vez de apenas os atributos especiais.

No mundo das linguagens de programação, a abstração é uma arma contra a complexidade da programação; seu propósito é simplificar o processo de programação. É uma arma efetiva, pois permite que os programadores enfoquem atributos essenciais e ignorem os subordinados.

Os dois tipos fundamentais de abstração nas linguagens de programação contemporâneas são a abstração de processos e a abstração de dados.

O conceito de **abstração de processo** está entre os mais antigos no projeto de linguagens de programação (a Plankalkül suportava abstração de processo nos anos 1940). Todos os subprogramas são abstrações de processo, porque fornecem uma maneira pela qual um programa especifica um processo, sem fornecer os detalhes de como ele realiza sua tarefa (ao menos no programa chamador). Por exemplo, quando um programa precisa ordenar um vetor de dados numéricos de algum tipo, ele normalmente usa um subprograma para o processo de ordenação. No momento em que um processo de ordenação é necessário, uma sentença como

```
sortInt(list, listLen)
```

é colocada no programa. Essa chamada é uma abstração do processo de ordenação real, cujo algoritmo não é especificado. A chamada é independente do algoritmo implementado no subprograma chamado.

No caso do subprograma `sortInt`, os únicos atributos essenciais são o nome do vetor a ser ordenado, o tipo de seus elementos, o tamanho do vetor e o fato de que a chamada a `sortInt` resulta na ordenação do vetor. O algoritmo específico implementado por `sortInt` é um atributo que não é essencial para o usuário. O usuário só precisa ver o nome e o protocolo do subprograma de ordenação para poder usá-lo.

O amplo uso da abstração de dados necessariamente seguiu as abstrações de processo, visto que uma parte integral e essencial de todas as abstrações de dados são suas operações, definidas como abstrações de processos.

11.2 INTRODUÇÃO À ABSTRAÇÃO DE DADOS

A evolução da abstração de dados começou em 1960, com a primeira versão de COBOL, que incluía a estrutura de dados de registro.[1] As linguagens baseadas em C têm estruturas, as quais também são registros. Um tipo de dado abstrato é uma estrutura de dados na forma de um registro, mas que inclui subprogramas que manipulam seus dados.

Sintaticamente, um tipo de dados abstrato é um invólucro que inclui apenas a representação de dados de um tipo de dados específico e os subprogramas que fornecem as operações para esse tipo. Por meio de controles de acesso, detalhes desnecessários do tipo podem ser ocultados de unidades externas ao invólucro que o utilizam. Unidades de programa que usam um tipo de dados abstrato podem declarar variáveis de tal tipo, mesmo que a representação real seja ocultada deles. Um exemplar de um tipo de dados abstrato é chamado de **objeto**.

Um dos objetivos da abstração de dados é semelhante ao da abstração de processo: ela é uma arma contra a complexidade; um modo de tornar programas grandes e/ou complicados mais controláveis. Outros objetivos e as vantagens dos tipos de dados abstratos são discutidos posteriormente nesta seção.

[1] Lembre-se, do Capítulo 2, de que um registro é uma estrutura de dados que armazena campos, os quais têm nomes e podem ser de tipos diferentes.

A programação orientada a objetos, descrita no Capítulo 12, é uma melhoria do uso de abstração de dados em desenvolvimento de software, e a abstração de dados é um de seus componentes fundamentais.

11.2.1 Ponto flutuante como um tipo de dados abstrato

O conceito de um tipo de dados abstratos não é recente. Todos os tipos de dados predefinidos, mesmo os de Fortran I, são abstratos, apesar de raramente serem chamados assim. Por exemplo, considere um tipo de dados de ponto flutuante. A maioria das linguagens de programação inclui ao menos um desses tipos. Um tipo de ponto flutuante fornece a maneira de criar variáveis para armazenar dados de ponto flutuante e um conjunto de operações aritméticas para manipular objetos do tipo.

Tipos de ponto flutuante em linguagens de alto nível empregam um conceito-chave na abstração de dados: ocultamento de informação. O formato real do valor de dado de ponto flutuante em uma célula de memória é oculto do usuário, e as únicas operações disponíveis são as fornecidas pela linguagem. Não é permitido ao usuário criar novas operações em dados do tipo, exceto aquelas que puderem ser construídas com as operações predefinidas. O usuário não pode manipular diretamente as partes da representação real dos valores, porque essa representação é oculta. É esse o recurso que permite a portabilidade entre implementações de uma linguagem, mesmo que elas possam usar representações diferentes para determinados tipos de dados. Por exemplo, antes de o padrão de representações de ponto flutuante IEEE 754 aparecer em meados dos anos 1980, diversas representações eram usadas por diferentes arquiteturas de computadores. Entretanto, essa variação não impedia os programas que usavam tipos de ponto flutuante de serem portáveis entre as várias arquiteturas.

11.2.2 Tipos de dados abstratos definidos pelo usuário

Um tipo de dados abstrato deve fornecer as mesmas características fornecidas por tipos definidos na linguagem, como um tipo de ponto flutuante: (1) uma definição de tipo que permita às unidades de programa declararem variáveis do tipo, mas que oculte a representação de seus objetos; e (2) um conjunto de operações para manipular os objetos.

Agora, definimos formalmente um tipo de dados abstrato no contexto de tipos definidos pelo usuário. Um **tipo de dados abstrato** satisfaz as condições a seguir:

- A representação dos objetos do tipo é ocultada das unidades de programa que o utilizam; então, as únicas operações diretas possíveis nesses objetos são aquelas fornecidas na definição do tipo.
- As declarações do tipo e os protocolos das operações em objetos do tipo, que fornecem sua interface, são contidos em uma única unidade sintática. A interface do tipo não depende da representação dos objetos ou da implementação das operações. Além disso, outras unidades de programa podem criar variáveis do tipo definido.

Há várias vantagens na ocultação de informações. Uma delas é a maior confiabilidade. As unidades de programa que utilizam um tipo de dados abstrato específico são chamadas de **clientes** desse tipo. Clientes não podem manipular diretamente as representações subjacentes dos objetos, seja intencionalmente ou por acidente, aumentando

a integridade de tais objetos. Objetos podem ser modificados apenas por meio das operações fornecidas.

Outra vantagem da ocultação de informações é que ela reduz a faixa de código e o número de variáveis das quais um programador precisa estar ciente ao escrever ou ler uma parte do programa. O valor de determinada variável só pode ser alterado por código em uma faixa restrita, o que facilita a compreensão do código e torna menos desafiador encontrar fontes de alterações incorretas.

A ocultação de informações também torna os conflitos de nomes menos prováveis, pois os escopos das variáveis são menores.

Por fim, considere a seguinte vantagem da ocultação de informações: suponha que a implementação pilha original utilize uma representação de lista encadeada. Posteriormente, devido a problemas de gerenciamento de memória dessa representação, a abstração pilha é alterada para usar uma representação adjacente (uma que implementa uma pilha em um vetor). Como a abstração de dados foi usada, essa mudança pode ser feita no código que define o tipo pilha, mas nenhuma alteração será necessária em quaisquer dos clientes da abstração pilha. É claro, uma mudança no protocolo de qualquer operação pode exigir alterações nos clientes.

Apesar de a definição de tipo de dado abstrato especificar que membros de dados de objetos devem ser ocultos dos clientes, surgem muitas situações nas quais eles precisam acessar esses membros. A solução comum é fornecer métodos de acesso, às vezes denominados **leitores** e **escritores** (*getters* e *setters*), que permitem aos clientes acessar indiretamente os dados chamados ocultos – uma solução melhor do que simplesmente tornar os dados públicos, fornecendo acesso direto. Existem três razões pelas quais os métodos de acesso são melhores:

1. Apenas acesso de leitura pode ser fornecido; há um método de leitura, mas nenhum método de escrita correspondente.
2. Restrições podem ser incluídas nos escritores. Por exemplo, se o valor de dado deve ser restrito de forma a estar em determinada faixa, o escritor pode garantir isso.
3. A implementação real do membro de dados pode ser modificada sem afetar os clientes se os leitores e escritores forem as únicas formas de acesso.

Especificar dados em um tipo de dado abstrato como públicos e fornecer métodos de acesso para esses dados são violações dos princípios dos tipos de dados abstratos. Alguns profissionais acreditam que essas são simplesmente brechas que tornam um projeto imperfeito utilizável. Como veremos na Seção 11.4.5.2, Ruby proíbe tornar públicos dados de instância. Contudo, ela também facilita a criação de funções de acesso. É um desafio para os desenvolvedores projetar tipos de dados abstratos nos quais todos os dados são realmente ocultos.

A principal vantagem de empacotar as declarações do tipo e suas operações em uma única unidade sintática é que isso fornece um método para organizar um programa em unidades lógicas que podem ser compiladas separadamente. Em alguns casos, a implementação é incluída na declaração de tipo; em outros, ela fica em uma unidade sintática separada. A vantagem de ter a implementação do tipo e suas operações em unidades sintáticas diferentes é que isso aumenta a modularidade do programa e é uma clara separação entre projeto e implementação. Se tanto as declarações quanto as definições de tipos

e operações estão na mesma unidade sintática, deve existir alguma forma de ocultar do cliente as partes da unidade que especificam as definições.

11.2.3 Um exemplo

A pilha é uma estrutura de dados amplamente aplicável que armazena algum número de elementos de dados e só permite acesso ao elemento em uma de suas extremidades: o topo. Suponha um tipo de dados abstrato construído para uma pilha com as seguintes operações abstratas:

create(stack)	Cria e possivelmente inicializa um objeto pilha
destroy(stack)	Libera o armazenamento para a pilha
empty(stack)	Um predicado ou função booleana que retorna verdadeiro (*true*) se a pilha especificada está vazia e falso (*false*) caso contrário.
push(stack, element)	Insere o elemento especificado na pilha especificada
pop(stack)	Remove o elemento do topo da pilha especificada
top(stack)	Retorna uma cópia do elemento do topo da pilha especificada

Note que algumas implementações de tipos de dados abstratos não exigem as operações de criação e destruição. Por exemplo, simplesmente definir uma variável como de um tipo de dados abstrato pode implicitamente criar a estrutura de dados subjacente e inicializá-la. O armazenamento para essa variável pode ser implicitamente liberado no final do escopo da variável.

Um cliente do tipo pilha poderia ter uma sequência de código como:

```
. . .
create(stk1);
push(stk1, color1);
push(stk1, color2);
temp = top(stk1);
. . .
```

11.3 QUESTÕES DE PROJETO PARA TIPOS DE DADOS ABSTRATOS

Um recurso para definir tipos de dados abstratos em uma linguagem deve fornecer uma unidade sintática que envolva a declaração do tipo e os protótipos dos subprogramas que implementam as operações em objetos do tipo. Deve ser possível torná-los visíveis aos clientes da abstração, permitindo que declarem variáveis do tipo abstrato e manipulem seus valores. Apesar de o nome do tipo precisar ter visibilidade externa, a representação deve ser oculta. A representação do tipo e as definições dos subprogramas que implementam as operações podem aparecer dentro ou fora dessa unidade sintática.

Poucas, se é que existirão, operações gerais predefinidas devem ser fornecidas para objetos de tipos de dados abstratos além daquelas dadas com a definição de tipo. Simplesmente não existem muitas operações que se apliquem a uma ampla faixa de tipos de

dados abstratos. Entre essas estão operações de atribuição e comparações de igualdade e diferença. Se a linguagem não permite que os usuários sobrecarreguem a atribuição, a operação de atribuição deve ser incluída na abstração. Comparações de igualdade e diferença devem ser predefinidas na abstração em alguns casos, mas não em outros. Por exemplo, se o tipo é implementado como um ponteiro, a igualdade pode significar a igualdade de ponteiros, mas o projetista pode querer que seja a das estruturas referenciadas pelos ponteiros.

Algumas operações são necessárias para muitos tipos de dados abstratos, mas, como não são universais, frequentemente devem ser fornecidas pelo projetista do tipo. Entre elas estão os iteradores, métodos de acesso, construtores e destrutores. Iteradores foram discutidos no Capítulo 8. Os métodos de acesso fornecem um caminho para que dados ocultos sejam acessados diretamente pelos clientes. Os construtores são usados para a inicialização de partes de objetos recém-criados. Os destrutores servem para recuperar armazenamento no monte, que é usado por partes de objetos de tipos de dados abstratos em linguagens que não fazem recuperação implícita de armazenamento.

Conforme mencionado, o invólucro para um tipo de dados abstrato define um único tipo e suas operações. Muitas linguagens contemporâneas, incluindo C++, Objective-C, Java e C#, suportam tipos de dados abstratos diretamente.

A primeira questão de projeto é se os tipos de dados abstratos podem ser parametrizados. Por exemplo, se a linguagem suporta tipos de dados abstratos parametrizados, é possível projetar um tipo de dados abstrato para alguma estrutura que poderia armazenar elementos de qualquer tipo. Tipos de dados abstratos parametrizados são discutidos na Seção 11.5. A segunda questão de projeto é quais controles de acesso são fornecidos e como são especificados. Por último, o projetista da linguagem deve decidir se a especificação do tipo é fisicamente separada de sua implementação (ou se essa é uma escolha do projetista).

11.4 EXEMPLOS DE LINGUAGEM

O conceito de abstração de dados tem suas origens em SIMULA 67, apesar de essa linguagem não fornecer suporte completo para tipos de dados abstratos, pois não inclui um modo de ocultar detalhes da implementação. Nesta seção, descrevemos o suporte para abstração de dados fornecido por C++, Objective-C, Java, C# e Ruby.

11.4.1 Tipos de dados abstratos em C++

C++, lançado pela primeira vez em 1985, foi criado com a adição de recursos à linguagem C. As primeiras adições importantes foram as que suportam a programação orientada a objetos. Como um dos principais componentes da programação orientada a objetos são os tipos de dados abstratos, C++ obviamente é obrigado a suportá-los.

C++ fornece duas construções muito parecidas, a classe e a estrutura, as quais suportam tipos de dados abstratos mais diretamente. Como essas estruturas são mais comumente usadas quando apenas dados são incluídos, não as discutimos aqui.

As classes C++ são tipos. Uma unidade de programa C++ que declara uma instância de uma classe também pode acessar qualquer uma das entidades públicas nessa classe, mas apenas por meio de uma instância da classe.

C++: nascimento, onipresença e críticas comuns

BJARNE STROUSTRUP

Bjarne Stroustrup é o projetista e implementador original de C++ e autor dos livros *A Tour of C++*, *Programming – Principles and Practice using C++*, *The C++ Programming Language* (*A Linguagem de Programação C++*, *no Brasil*), *The Design and Evolution of C++* e muitas outras publicações. Suas áreas de pesquisa incluem sistemas distribuídos, projeto, técnicas de programação, ferramentas de desenvolvimento de software e linguagens de programação. Ativamente envolvido na padronização ANSI/ISO de C++, Dr. Stroustrup é diretor-gerente da divisão de tecnologia da Morgan Stanley, em Nova York, EUA, professor convidado de Ciência da Computação na Universidade de Columbia e professor de pesquisa emérito de Ciência da Computação na Universidade do Texas A&M. Além disso, é membro da National Academy of Engineering, da ACM e do IEEE. Em 1993, recebeu o Prêmio Grace Murray Hopper da ACM "por seu trabalho pioneiro que levou às bases da linguagem de programação C++. Por meio dessas bases e de esforços contínuos do Dr. Stroustrup, C++ se tornou uma das linguagens de programação mais influentes na história da computação".
(ano da entrevista: 2002)

UM BREVE HISTÓRICO SOBRE A RELAÇÃO DE STROUSTRUP COM A COMPUTAÇÃO

No que você estava trabalhando, e quando, antes de se juntar ao Bell Labs no início dos anos 1980? No Bell Labs, estava pesquisando na área geral de sistemas distribuídos. Entrei lá em 1979. Antes disso, estava terminando meu doutorado nessa área na Universidade de Cambridge.

Você imediatamente começou a trabalhar em "C com Classes" (que mais tarde se tornou C++)? Trabalhei em alguns projetos relacionados à computação distribuída antes de iniciar C com Classes e durante o desenvolvimento dele e de C++. Por exemplo, estava tentando encontrar uma maneira de distribuir o núcleo do UNIX entre vários computadores e ajudei diversos projetos a construírem simuladores.

Foi o interesse em matemática que o levou a essa profissão? Me inscrevi para um bacharelado em "matemática com ciência da computação" e meu mestrado é oficialmente em matemática. Eu – erroneamente – pensei que a computação fosse algum tipo de matemática aplicada. Fiz alguns anos de matemática e me considero fraco no assunto, mas isso ainda é muito melhor que não saber nada de matemática. Na época em que me inscrevi, nunca tinha visto um computador. O que amo na computação é a programação e não as áreas mais matemáticas.

DISSECANDO UMA LINGUAGEM BEM-SUCEDIDA

Gostaria de começar de trás para frente, listando alguns itens que eu acredito terem feito C++ onipresente, e ver sua reação. Primeiro: essa linguagem tem "código aberto", não proprietário e padronizado pela ANSI/ISO. O padrão ISO C++ é importante. Existem muitas implementações de C++ desenvolvidas e mantidas de forma independente. Sem um padrão para elas se adequarem e um processo de padronização para ajudar a coordenar a evolução de C++, surgiria um caos de dialetos.

Também é importante o fato de existirem implementações de código aberto e comerciais disponíveis. Além disso, para muitos usuários, é crucial que o padrão forneça uma medida de proteção contra a manipulação por parte dos fornecedores de implementações.

O processo de padronização da ISO é aberto e democrático. O comitê de C++ raramente se encontra com menos de 50 pessoas presentes e em geral mais de oito nações estão representadas em cada reunião. Não é apenas um fórum de vendedores.

C++ é ideal para a programação de sistemas (a qual, no momento do nascimento de C++, era o maior setor do mercado de desenvolvimento de código).

Sim, C++ é um forte concorrente para qualquer projeto de programação de sistemas. Também é eficiente para programação de sistemas embarcados, que atual-

mente é o setor de crescimento mais rápido. Outra área de crescimento para C++ é a programação numérica/de engenharia/científica de alto desempenho.

Sua natureza orientada a objetos e a inclusão de classes e bibliotecas tornam a programação mais eficiente e transparente. C++ é uma linguagem de programação multiparadigma. Ou seja, ela suporta vários estilos fundamentais de programação (incluindo a orientada a objetos) e combinações desses estilos. Quando bem utilizada, leva a bibliotecas mais limpas, mais flexíveis e mais eficientes do que as que podem ser fornecidas com apenas um paradigma. Um exemplo são os contêineres e algoritmos da biblioteca padrão C++, os quais são basicamente um framework de programação genérico. Quando usados junto com hierarquias de classes (orientadas a objetos), o resultado é uma combinação excelente de segurança de tipos, eficiência e flexibilidade.

Sua incubação no ambiente de desenvolvimento no AT&T O AT&T Bell Labs forneceu um ambiente crucial para o desenvolvimento de C++. Os laboratórios eram uma fonte excepcionalmente rica de problemas desafiadores e um ambiente singularmente colaborativo para pesquisas práticas. C++ emergiu do mesmo laboratório de pesquisa que C e se beneficiou da mesma tradição intelectual, experiência e pessoas excepcionais. O tempo todo, o AT&T deu suporte à padronização de C++. Entretanto, C++ não teve uma campanha de marketing pesada, como muitas linguagens modernas. O Bell Labs simplesmente não trabalha dessa maneira.

Esqueci algo de sua lista principal de itens? Sem dúvida.

Agora, deixe-me parafrasear algumas das críticas a C++ e ver como você reage: C++ é uma linguagem imensa/desajeitada. O problema "hello world" é 10 vezes maior em C++ que em C. C++ certamente não é uma linguagem pequena, mas poucas linguagens modernas o são. Se uma linguagem é pequena, você tende a precisar de bibliotecas gigantescas para fazer o trabalho e normalmente depende de convenções e extensões. Prefiro ter as partes-chave da complexidade inevitável na linguagem, em que elas podem ser vistas, ensinadas e efetivamente padronizadas, em vez de ocultas em algum lugar de um sistema. Para a maioria das necessidades, não considero C++ "desajeitada". O programa "hello world" em C++ não é maior que seu equivalente em C na minha máquina, e não deve ser na sua também.

Na verdade, o código objeto para a versão de C++ do programa "hello world" é menor que a versão de C na minha máquina. Não existe uma razão linguística pela qual uma versão deveria ser maior que a outra. Tudo é uma questão de como o implementador organizou as bibliotecas. Se uma versão é significativamente maior que a outra, relate o problema para o implementador da versão maior.

Os críticos dizem que é mais difícil programar em C++ (comparado com C). Você mesmo comentou algo sobre atirar em seu próprio pé, em relação a C *versus* C++. Sim, eu realmente disse algo como "é fácil atirar em seu próprio pé com C; com C++ é mais difícil, mas, quando acontece, C++ explode sua perna inteira". No entanto, o que eu disse sobre C++ é, em um grau variável, verdadeiro para todas as linguagens poderosas. À medida que você protege as pessoas de perigos simples, elas começam a enfrentar problemas novos e menos óbvios. Alguém que evita os problemas simples pode simplesmente estar se direcionando para um problema não tão simples. Uma das dificuldades de ambientes muito protetores e do excesso de suporte é que os problemas difíceis podem ser descobertos tarde demais. Além disso, um problema raro é mais difícil de encontrar do que um frequente, porque você não suspeita dele.

C++ é adequado para sistemas embarcados de hoje, mas não para os sistemas de software para Internet atuais. Ele também é adequado a – e amplamente usado em – "software para a Internet" hoje. Por exemplo, dê uma olhada na minha página chamada "aplicações C++". Você notará que alguns dos maiores provedores de serviços Web, como Amazon, Adobe, Google, Quicken e Microsoft, dependem criticamente de C++. Na área relacionada de jogos se usa muito C++.

Esqueci de alguma outra pergunta que você ouve com frequência? Com certeza.

11.4.1.1 Encapsulamento

Os dados definidos em uma classe C++ são chamados de **membros de dados**; as funções (métodos) definidas em uma classe são chamadas de **funções membro**. Membros de dados e funções membro aparecem em duas categorias: classes e instâncias. Os membros de classe são associados à classe; os membros de instância são associados às instâncias da classe. Neste capítulo são discutidos apenas os membros de instância de uma classe. Todas as instâncias de uma classe compartilham um conjunto único de funções membro, mas cada uma tem seu próprio conjunto dos membros de dados da classe. Instâncias de classe podem ser estáticas, dinâmicas da pilha ou dinâmicas do monte. Se estáticas ou dinâmicas da pilha, são referenciadas diretamente com variáveis de valores. Se dinâmicas do monte, são referenciadas por ponteiros. Instâncias de classe dinâmicas da pilha são sempre criadas pela elaboração de uma declaração de objeto. Além disso, seu tempo de vida termina quando o final do escopo de sua declaração é alcançado. Instâncias de classe dinâmicas do monte são criadas com o operador **`new`** e destruídas com **`delete`**. Tanto classes da pilha quanto dinâmicas do monte podem ter ponteiros como membros de dados que referenciam dados dinâmicos do monte – então, mesmo que uma instância seja dinâmica da pilha, ela pode incluir membros de dados que referenciam dados dinâmicos do monte.

Uma função membro de uma classe pode ser definida de duas maneiras distintas: a definição completa pode aparecer na classe ou apenas em seu cabeçalho. Quando tanto o cabeçalho quanto o corpo de uma função membro aparecem na definição da classe, a função é implicitamente internalizada. Isso significa que seu código é colocado no código do chamador, em vez de exigir a ligação usual de chamada e retorno. Se apenas o cabeçalho de uma função membro aparece na definição da classe, sua definição completa aparece fora da classe e é compilada separadamente. A razão para permitir que as funções membro fossem internalizadas era economizar sobrecarga de chamada de função em aplicações de tempo real, nas quais a eficiência em tempo de execução é de fundamental importância. A desvantagem de internalizar funções membro é que isso polui a interface da definição da classe, o que resulta em uma redução na legibilidade.

Colocar as definições das funções membro fora da definição da classe separa a especificação da implementação, um objetivo comum da programação moderna.

11.4.1.2 Ocultação de informação

Uma classe C++ pode conter tanto entidades ocultas quanto visíveis (ocultas dos clientes ou visíveis para os clientes da classe). Entidades ocultas são colocadas em uma cláusula **`private`** e entidades visíveis, ou públicas, aparecem em uma cláusula **`public`**. Logo, a cláusula **`public`** descreve a interface para instâncias da classe. Existe ainda uma terceira categoria de visibilidade, **`protected`**, que torna um membro visível para subclasses, mas não para clientes.

11.4.1.3 Construtores e destrutores

C++ permite que o usuário inclua funções chamadas de **construtores** em definições de classe, usadas para inicializar os membros de dados de novos objetos criados. Um cons-

trutor também pode alocar os dados dinâmicos do monte referenciados pelos membros ponteiros do novo objeto. Construtores são implicitamente chamados quando um objeto do tipo da classe é criado. Um construtor tem o mesmo nome da classe cujos objetos inicializa. Os construtores podem ser sobrecarregados, mas é claro que cada construtor de uma classe deve ter um perfil de parâmetros único.

Uma classe C++ também pode incluir uma função chamada de **destrutor**, implicitamente chamada quando o tempo de vida de uma instância da classe termina. Conforme mencionado, instâncias de classe dinâmicas da pilha podem conter membros ponteiros que referenciam dados dinâmicos do monte. A função destrutora para essa instância pode incluir um operador **delete** nos membros ponteiros para liberar o espaço no monte que referenciam. Os destrutores são usados como uma ajuda à depuração, mostrando ou imprimindo os valores de algum ou de todos os membros de dados do objeto antes de esses membros serem liberados. O nome de um destrutor é o nome da classe, precedido por til (~).

Nem construtores nem destrutores têm tipos de retorno, e nenhum deles usa sentenças **return**. Os dois podem ser chamados explicitamente.

11.4.1.4 Um exemplo

Nosso exemplo de tipo de dados abstrato em C++ é uma pilha:

```
#include <iostream.h>
class Stack {
  private:  //** Esses membros são visíveis apenas para outros
            //** membros e amigos (consulte a Seção 11.6.4)
    int *stackPtr;
    int maxLen;
    int topSub;
  public:  //** Esses membros são visíveis para os clientes
    Stack() {   //** Um construtor
      stackPtr = new int [100];
      maxLen = 99;
      topSub = -1;
    }
    ~Stack() {delete  [] stackPtr;};  //** Um destrutor
    void push(int number) {
      if (topSub == maxLen)
        cerr << "Error in push--stack is full\n";
      else stackPtr[++topSub] = number;
    }
    void pop() {
      if (empty())
        cerr << "Error in pop--stack is empty\n";
      else topSub--;
    }
    int  top() {
      if  (empty())
```

```
        cerr << "Error in top--stack is empty\n";
      else
        return  (stackPtr[topSub]);
    }
    int  empty() {return  (topSub == -1);}
}
```

Discutimos poucos aspectos dessa definição de classe, porque isso não é necessário para se entender todos os detalhes do código. Objetos da classe `Stack` são dinâmicos da pilha, mas incluem um ponteiro que referencia dados dinâmicos do monte. A classe `Stack` tem três membros de dados – `stackPtr`, `maxLen` e `topSub` – todos privados. O membro `stackPtr` é usado para referenciar os dados dinâmicos do monte, a matriz que implementa a pilha. A classe tem também quatro funções membro públicas – `push`, `pop`, `top` e `empty` –, assim como um construtor e um destrutor. Todas as definições de funções membro são incluídas nessa classe, apesar de poderem ter sido definidas externamente. Como os corpos das funções são incluídos, são todos implicitamente internalizados. O construtor usa o operador **`new`** para alocar um vetor com 100 elementos inteiros (**`int`**) do monte. Ele também inicializa `maxLen` e `topSub`.

A seguir está um exemplo de programa que usa o tipo de dado abstrato `Stack`:

```
void main() {
  int  topOne;
  Stack stk;  //** Cria uma instância da classe Stack
  stk.push(42);
  stk.push(17);
  topOne = stk.top();
  stk.pop();
  . . .
}
```

A seguir, temos uma definição da classe `Stack` apenas com os protótipos das funções membro. Esse código é armazenado em um arquivo de cabeçalho com a extensão de nome de arquivo `.h`. As definições das funções membro seguem a definição da classe. Elas usam o operador de resolução de escopo, `::`, para indicar a classe à qual pertencem. Essas definições são armazenadas em um arquivo de código com a extensão `.cpp`.

```
// Stack.h - o arquivo de cabeçalho para a classe Stack
#include <iostream.h>
class Stack {
  private:  //** Esses membros são visíveis apenas para outros
            //** membros e amigos (consulte a Seção 11.6.3)
    int *stackPtr;
    int maxLen;
    int topSub;
  public:    //** Esses membros são visíveis para os clientes
    Stack();  //** Um construtor
    ~Stack();  //** Um destrutor
    void push(int);
    void pop();
```

```
    int top();
    int empty();
}
// Stack.cpp - o arquivo de implementação para a classe Stack
#include <iostream.h>
#include "Stack.h"
using std::cout;
Stack::Stack() {  //** Um construtor
  stackPtr =  new int  [100];
  maxLen = 99;
  topSub = -1;
}
Stack::~Stack() {delete  [] stackPtr;};  //** Um destrutor
void Stack::push(int  number) {
  if (topSub == maxLen)
    cerr << "Error in push--stack is full\n";
  else stackPtr[++topSub] = number;
}
void Stack::pop() {
  if (topSub == -1)
    cerr << "Error in pop--stack is empty\n";
  else  topSub--;
}
int top() {
      if  (topSub == -1)
        cerr << "Error in top--stack is empty\n";
      else
        return  (stackPtr[topSub]);
    }
int Stack::empty() {return  (topSub == -1);}
```

11.4.2 Tipos de dados abstratos em Objective-C

Conforme mencionado, Objective-C é semelhante a C++, pois seu projeto inicial foi a linguagem C com extensões para suportar programação orientada a objetos. Uma das diferenças fundamentais entre elas é que Objective-C usa a sintaxe de Smalltalk para suas chamadas de método.

11.4.2.1 Encapsulamento

A parte da interface de uma classe Objective-C é definida em um contêiner chamado de **interface**, com a seguinte sintaxe geral:

```
@interface nome-classe: classe-pai {
 declarações de variáveis de instância
}
 protótipos de método
@end
```

A primeira e a última linha, que começam com arroba (`@`), são diretivas.

A implementação de uma classe é empacotada em um contêiner, naturalmente chamado de *implementação*, o qual tem a seguinte sintaxe:

```
@implementation nome-classe
 definições de método
@end
```

Como em C++, as classes são tipos em Objective-C.

Os protótipos de método têm a seguinte sintaxe:

```
(+ | -) (tipo-retorno) nome-método [: (parâmetros-formais)];
```

Quando presente, o sinal de adição indica que o método é de classe; um sinal de subtração indica um método de instância. Os colchetes envolvendo os parâmetros formais indicam que eles são opcionais. Nem os parênteses nem os dois pontos estão presentes quando não há parâmetros. Como na maioria das outras linguagens que suportam programação orientada a objetos, todas as instâncias de uma classe Objective-C compartilham uma única cópia de seus métodos de instância, mas cada instância tem sua própria cópia dos dados de instância.

A forma sintática da lista de parâmetros formal é diferente das linguagens mais comuns, C, C++, Java e C#. Se existe um parâmetro, seu tipo é especificado entre parênteses antes do nome do parâmetro, como no seguinte protótipo de método:

```
-(void) meth1: (int) x;
```

O nome desse método é `meth1:` (observe os dois pontos). Um método com dois parâmetros poderia aparecer como no seguinte exemplo de protótipo de método:

```
-(int) meth2: (int) x second: (float) y;
```

Nesse caso, o nome do método é `meth2:second:`, embora evidentemente seja um nome mal escolhido. A última parte do nome (`second`) poderia ser omitida, como no seguinte:

```
-(int) meth2: (int) x: (float) y;
```

Nesse caso, o nome do método é `meth2::`.

As definições de método são como os protótipos de método, exceto que têm uma sequência de sentenças delimitada por colchetes no lugar do ponto e vírgula.

A sintaxe de uma chamada a um método sem parâmetros é a seguinte:

```
[nome-objeto nome-método];
```

Se um método recebe um parâmetro, dois pontos são anexados ao seu nome e o parâmetro vem a seguir. Não há nenhuma outra pontuação entre o nome do método e o

parâmetro. Por exemplo, uma chamada a um método denominado `add1` no objeto referenciado por `myAdder`, que recebe um parâmetro, o literal `7`, apareceria como segue:

```
[myAdder add1: 7];
```

Se um método recebe dois parâmetros e tem apenas uma parte em seu nome, dois pontos vêm após o primeiro parâmetro e o segundo parâmetro aparece depois deste. Nenhuma outra pontuação é usada entre os dois parâmetros. Se houver mais parâmetros, esse padrão se repete. Por exemplo, se `add1` recebesse três parâmetros e não tivesse outras partes em seu nome, poderia ser chamado com o seguinte:

```
[myAdder add1: 7: 5: 3];
```

Um método pode ter vários parâmetros e várias partes em seu nome, como no exemplo anterior:

```
-(int) meth2: (int) x second: (float) y;
```

Um exemplo de chamada a esse método seria:

```
[myObject meth2: 7 second: 3,2];
```

Em Objective-C, os construtores são denominados **inicializadores**; eles fornecem apenas valores iniciais. Além disso, podem receber qualquer nome e, como resultado, devem ser chamados explicitamente. Os construtores retornam uma referência para o novo objeto, de modo que seu tipo é sempre um ponteiro para o nome da classe. Eles usam uma sentença **`return`** que retorna **`self`**, uma referência para o objeto atual.

Em Objective-C, um objeto é criado pela chamada a **`alloc`**. Normalmente, depois dessa chamada, o construtor da classe é chamado explicitamente. Essas duas chamadas podem ser (e normalmente são) colocadas em cascata, como na sentença a seguir, que cria um objeto da classe `Adder` com **`alloc`**, então chama seu construtor, `init`, no novo objeto e coloca o endereço do novo objeto em `myAdder`:

```
Adder *myAdder = [[Adder alloc]init];
```

Todas as instâncias de classe são dinâmicas do monte e referenciadas por meio de variáveis de referência.

Os programas C quase sempre importam um arquivo de cabeçalho para funções de entrada e saída, o `stdio.h`. Em Objective-C, um arquivo de cabeçalho normalmente é importado e tem os protótipos de uma variedade de funções frequentemente exigidas, incluindo aquelas para entrada e saída, assim como alguns dados necessários. Isso é feito com o seguinte:

```
#import <Foundation/Foundation.h>
```

Praticamente todo código na função `main` de um programa normalmente é colocado em uma sentença composta que vem após `@autoreleasepool`. Qualquer objeto criado na sentença composta é automaticamente liberado no final dessa sentença.

A sentença composta `@autoreleasepool` em uma função `main` é seguida por uma sentença `return 0`, indicando que a execução chegou ao fim do código de `main`.

11.4.2.2 Ocultação de informação

Objective-C usa as diretivas `@private` e `@public` para especificar os níveis de acesso das variáveis de instância em uma definição de classe. Elas são usadas da mesma forma como as palavras reservadas **`public`** e **`private`** são usadas em C++. A diferença é que o padrão na Objective-C é `protected`, enquanto em C++ é `private`. Ao contrário da maioria das linguagens de programação que suportam orientação a objetos, em Objective-C não há maneira de restringir o acesso a um método.

Em Objective-C, a convenção é o nome de um método de leitura para uma variável de instância ser o nome da variável. O nome do método de escrita é a palavra `set` com o nome da variável com inicial maiúscula anexado. Assim, para uma variável chamada `sum`, o método de leitura se chamaria `sum` e o método de escrita se chamaria `setSum`. Supondo que `sum` seja uma variável **`int`**, esses métodos poderiam ser definidos como:

```
// O leitor para sum
-(int) sum {
  return sum;
}
// O escritor para sum
-(void) setSum: (int) s {
  sum = s;
}
```

Se os métodos de leitura e escrita para determinada variável não impõem quaisquer restrições sobre suas ações, eles podem ser gerados automaticamente pelo compilador de Objective-C. Isso é feito listando-se, na diretiva `property`, na seção de interface, as variáveis de instância para as quais leitores e escritores devem ser gerados, como no seguinte:

```
@property int sum;
```

Na seção de implementação, as variáveis são listadas em uma diretiva `synthesize`, como no seguinte:

```
@synthesize sum;
```

As variáveis para as quais são gerados leitores e escritores pelo compilador frequentemente são denominadas **propriedades** e diz-se que os métodos de acesso são **sintetizados**.

Os leitores e escritores de variáveis de instância podem ser usados de duas maneiras, ou em chamadas de método ou na notação por pontos, como se fossem publicamente acessíveis. Por exemplo, se tivéssemos definido um leitor e um escritor para a variável `sum`, eles poderiam ser usados como no seguinte:

```
[myObject setSum: 100];
newSum = [myObject sum];
```

ou como se fossem publicamente acessíveis, como no seguinte:

```
myObject.sum = 100;
newSum = myObject.sum;
```

11.4.2.3 Um exemplo

A seguir estão as definições da interface e da implementação da classe `stack` em Objective-C:

```
// stack.m - interface e implementação de uma pilha simples

#import <Foundation/Foundation.h>

// Seção de interface

@interface Stack: NSObject {
  int stackArray [100];
  int stackPtr;
  int maxLen;
  int topSub;
}
  -(void) push: (int) number;
  -(void) pop;
  -(int) top;
  -(int) empty;
@end
// Seção de implementação

@implementation Stack
  -(Stack *) initWith {
    maxLen = 100;
    topSub = -1;
    stackPtr = stackArray;
    return self;
  }

  -(void) push: (int) number {
    if (topSub == maxLen)
       NSLog(@"Error in push--stack is full");
     else
       stackPtr[++topSub] = number;
  }

  -(void) pop {
```

```
    if (topSub == -1)
       NSLog(@"Error in pop--stack is empty");
     else
       topSub--;
  }
  -(int) top {
     if (topSub >= 0)
       return  stackPtr[topSub]);
     else
       NSLog(@"Error in top--stack is empty");
  }

  -(int) empty {
     return topSub == -1);
  }

  int main (int  argc,  char  *argv[]) {
    int  temp;
    @autoreleasepool{
    Stack *myStack = [[Stack  alloc]initWith];
    [myStack push: 5];
    [myStack push: 3];
    temp = [myStack top];
    NSLog(@"Top element is:%i", temp);
    [myStack pop];
    temp = [myStack top];
    NSLog(@"Top element is:%i", temp);
    temp = [myStack top];
    [myStack pop];
    }
    return 0;
  }
@end
```

A saída desse programa é:

```
Top element is: 3
Top element is: 5
Error in top--stack is empty
Error in pop--stack is empty
```

A saída na tela de um programa Objective-C é criada com uma chamada a um método de nome incomum, `NSLog`, o qual recebe uma cadeia literal como parâmetro. As cadeias literais são criadas com uma arroba (`@`) seguida por uma cadeia entre aspas. Se uma cadeia de saída inclui os valores de variáveis, os nomes das variáveis são incluídos como parâmetros na chamada a `NSLog`. As posições dos valores na cadeia literal são marcadas com códigos de formatação; por exemplo, `%i` para um inteiro e `%f` para um valor de ponto flutuante em notação científica, semelhante à função `printf` de C.

11.4.2.4 Avaliação

O suporte para tipos de dados abstratos em Objective-C é adequado. Algumas pessoas acham estranho o uso de formas sintáticas de duas linguagens muito diferentes, Smalltalk (para suas chamadas de método) e C (para quase tudo mais). Além disso, o uso de diretivas em vez de construções da linguagem para indicar seções de interfaces e de implementação da classe também difere daquele da maioria das outras linguagens de programação. Uma pequena deficiência é a falta de um modo de restringir o acesso a métodos. Assim, mesmo métodos destinados a serem usados apenas dentro de uma classe são acessíveis para os clientes. Outra deficiência secundária é que os construtores devem ser chamados explicitamente, o que exige que os programadores se lembrem de chamá-los e leva a uma maior desordem no programa cliente.

11.4.3 Tipos de dados abstratos em Java

O suporte de Java para tipos de dados abstratos é similar ao de C++. Existem, no entanto, algumas diferenças importantes. Todos os objetos são alocados do monte e acessados por variáveis de referência. Os métodos em Java devem ser completamente definidos em uma classe. Um corpo de método deve aparecer com seu cabeçalho de método correspondente.[2] Logo, um tipo de dados abstrato é declarado e definido em uma única unidade sintática. Um compilador Java pode internalizar qualquer método não sobrescrito. As definições são ocultas dos clientes por serem declaradas como privadas.

Uma vantagem importante das classes Java em relação às classes de C++ e Objective-C é o uso de coleta de lixo implícita de todos os objetos. Isso permite que o programador ignore o problema da liberação de objetos e do congestionamento do código de liberação nas implementações de tipos de dados abstratos.

Em vez de ter cláusulas privadas e públicas em suas definições de classe, em Java os modificadores de acesso podem ser anexados às definições de métodos e de variáveis. Se uma variável ou método de instância não tem um modificador de acesso, tem acesso ao pacote, o qual está discutido na Seção 11.7.2.

11.4.3.1 Um exemplo

A seguir, temos uma definição de classe em Java para nosso exemplo da pilha:

```
class StackClass {
  private int [] stackRef;
  private int maxLen,
              topIndex;
  public StackClass() {  // Um construtor
    stackRef = new int [100];
    maxLen = 99;
    topIndex = -1;
```

[2] As interfaces Java são uma exceção a isso – uma interface tem cabeçalhos de método, mas não pode incluir seus corpos.

```
  }
  public void push(int number) {
    if (topIndex == maxLen)
      System.out.println("Error in push-stack is full");
    else stackRef[++topIndex] = number;
  }
  public void pop() {
    if (empty())
      System.out.println("Error in pop-stack is empty");
    else --topIndex;
  }
  public int top() {
    if (empty()) {
      System.out.println("Error in top-stack is empty");
      return 9999;
    }
    else
      return (stackRef[topIndex]);
  }
  public boolean empty() {return (topIndex == -1);}
}
```

Uma classe de exemplo que usa `StackClass` é mostrada a seguir:

```
public class TstStack {
  public static void main(String[] args) {
    StackClass myStack = new StackClass();
    myStack.push(42);
    myStack.push(29);
    System.out.println("29 is: " + myStack.top());
    myStack.pop();
    System.out.println("42 is: " + myStack.top());
    myStack.pop();
    myStack.pop();  // Produz uma mensagem de erro
  }
}
```

Uma diferença óbvia é a falta de um destrutor na versão Java, desnecessário devido à coleção de lixo implícita da linguagem.[3]

11.4.3.2 Avaliação

Embora seja diferente de algumas maneiras principalmente estéticas, o suporte de Java para tipos de dados abstratos é semelhante ao de C++. Java claramente fornece o que é necessário para projetar tipos de dados abstratos.

[3]Em Java, o método **`finalize`** serve como um tipo de destrutor.

11.4.4 Tipos de dados abstratos em C#

Lembre-se de que C# é baseado tanto em C++ quanto em Java e inclui algumas construções novas. Como em Java, todas as instâncias de classe em C# são dinâmicas do monte. Construtores padrão, que fornecem valores iniciais para dados de instância, são predefinidos para todas as classes. Esses construtores fornecem valores iniciais típicos, como `0` para tipos **`int`** e **`false`** para tipos booleanos (**`boolean`**). Um usuário pode fornecer um ou mais construtores para qualquer classe que definir. Esses construtores podem atribuir valores iniciais a alguns ou a todos os dados de instâncias da classe. Qualquer variável de instância não inicializada em um construtor definido pelo usuário recebe um valor do construtor padrão.

Embora C# permita definir destrutores, como utiliza coleta de lixo para a maioria de seus objetos do monte, eles raramente são usados.

11.4.4.1 Encapsulamento

Como mencionado na Seção 11.4.2, C++ contém classes e estruturas, as quais são construções praticamente idênticas. A única diferença é que o modificador de acesso padrão para classes é **`private`**, enquanto para estruturas é **`public`**. C# também tem estruturas, mas elas são bem diferentes das de C++. Em C#, estruturas são, em certo sentido, classes leves. Elas podem ter construtores, propriedades, métodos e atributos de dados e implementar interfaces, mas não suportam herança. Outra diferença importante entre estruturas e classes em C# é que estruturas são tipos de valores, em oposição a tipos de referência. Elas são alocadas na pilha de tempo de execução, em vez de no monte. Por padrão, se são passadas como parâmetros, a exemplo de outros tipos de valor, são passadas por valor. Todos os tipos de valores em C#, incluindo seus tipos primitivos, são estruturas. As estruturas podem ser criadas declarando-as, como outros tipos de valor predefinidos, como **`int`** ou **`float`**. Também podem ser criadas com o operador **`new`**, que chama um construtor para inicializá-las.

Estruturas são usadas em C# principalmente para implementar tipos relativamente simples e pequenos que nunca precisarão ser tipos base para herança. Elas também são usadas quando é conveniente para os objetos de um tipo serem alocados na pilha, em vez de no monte.

11.4.4.2 Ocultação de informação

C# usa os modificadores de acesso **`private`** e **`protected`** exatamente como eles são usados em Java.

C# fornece propriedades, herdadas de Delphi, como uma maneira de implementar leitores e escritores sem exigir chamadas a métodos explícitas pelo cliente. Como em Objective-C, propriedades fornecem acesso implícito a dados de instância privados específicos. Por exemplo, considere a seguinte classe simples e o código cliente:

```
public class Weather {
  public int DegreeDays {  //** DegreeDays é uma propriedade
    get {
       return degreeDays;
    }
    set {
```

```
      if(value < 0 || value > 30)
        Console.WriteLine(
             "Value is out of range: {0}",  value);
      else
        degreeDays =  value;
    }
  }
  private int  degreeDays;
   . . .
  }
. . .
Weather w =  new Weather();
int  degreeDaysToday, oldDegreeDays;
. . .
w.DegreeDays = degreeDaysToday;
. . .
oldDegreeDays = w.DegreeDays;
```

Na classe `Weather`, é definida a propriedade `DegreeDays`. Essa propriedade fornece um método de leitura e um método de escrita para acesso ao membro de dados privado `degreeDays`. No código cliente, logo após a definição da classe, `degreeDays` é tratado como uma variável membro pública, apesar de o acesso a ele estar disponível apenas por meio da propriedade. Note o uso da variável implícita **`value`** no método de escrita. Esse é o mecanismo por meio do qual o novo valor da propriedade é referenciado.

O exemplo da pilha em C# não é mostrado aqui. A única diferença entre a versão em Java da Seção 11.4.4.1 e a versão em C# são as chamadas de método de saída e o uso de **`bool`**, em vez de **`boolean`**, para o tipo de retorno do método `empty`.

11.4.5 Tipos de dados abstratos em Ruby

Ruby fornece suporte para tipos de dados abstratos por meio de suas classes. Em termos de capacidades, as classes de Ruby são similares às de C++ e de Java.

11.4.5.1 Encapsulamento

Em Ruby, uma classe é definida em uma sentença composta aberta com a palavra reservada **`class`** e fechada com **`end`**. Os nomes das variáveis de instância têm uma forma sintática especial: eles devem começar com arroba (`@`). Os métodos de instância têm a mesma sintaxe das funções em Ruby: começam com a palavra reservada **`def`** e terminam com **`end`**. Métodos de classe são distinguidos de métodos de instância por meio da inserção do nome da classe no início do nome do método com um ponto como separador. Por exemplo, em uma classe chamada `Stack`, um nome de método de classe teria `Stack` antes do nome. Construtores em Ruby são chamados `initialize`. Como o construtor não pode ser sobrecarregado, só pode haver um por classe.

As classes em Ruby são dinâmicas, no sentido de que membros podem ser adicionados a qualquer momento. Isso é feito simplesmente incluindo-se mais definições de classe que especifiquem novos membros. Além disso, mesmo as classes predefinidas da linguagem, como `String`, podem ser estendidas. Por exemplo, considere a seguinte definição de classe:

```
class MyClass
  def meth1
    . . .
  end
end
```

Essa classe poderia ser estendida pela adição de um segundo método, `meth2`, com uma segunda definição de classe:

```
class MyClass
  def meth7
    . . .
  end
end
```

Métodos também podem ser removidos de uma classe. Isso é feito ao se fornecer outra definição de classe na qual o método a ser removido é enviado como parâmetro ao método `remove_method`. As classes dinâmicas de Ruby são outro exemplo de um projetista de linguagem trocando legibilidade (e confiabilidade) por flexibilidade. Permitir alterações dinâmicas em classes claramente aumenta a flexibilidade da linguagem, ao passo que prejudica a legibilidade. Para determinar o comportamento de uma classe em determinado ponto em um programa, deve-se encontrar e considerar todas as suas definições no programa.

11.4.5.2 Ocultação de informação

O controle de acesso para métodos em Ruby é dinâmico; então, violações de acesso são detectadas apenas durante a execução. O método de acesso padrão é público, mas também pode ser protegido ou privado. Existem duas maneiras de especificar o controle de acesso, ambas as quais usam funções com os mesmos nomes dos níveis de acesso: **`private`** e **`public`**. Uma maneira é chamar a função apropriada sem parâmetros. Isso restaura o acesso padrão para métodos definidos na sequência na classe. Por exemplo,

```
class MyClass
  def meth1
  . . .
  end
  . . .
private
  def meth7
```

```
  . . .
  end
  . . .
end  # da classe MyClass
```

A alternativa é chamar as funções de controle de acesso com os nomes dos métodos específicos como parâmetros. Por exemplo, o seguinte é semanticamente equivalente à definição de classe anterior:

```
class MyClass
  def meth1
  . . .
  end
  . . .
  def meth7
  . . .
  end
  private :meth7, . . .
   end  # da classe MyClass
```

Em Ruby, todos os membros de dados de uma classe são privados e isso não pode ser mudado. Assim, os membros de dados só podem ser acessados pelos métodos da classe, alguns dos quais podem ser métodos de acesso. Em Ruby, dados de instância acessíveis por meio de métodos de acesso são denominados **atributos**.

Para uma variável de instância chamada `@sum`, os métodos de leitura e escrita seriam os seguintes:

```
def sum
  @sum
end
def sum=(new_sum)
  @sum = new_sum
end
```

Observe que os leitores recebem o nome da variável de instância, menos o `@`. Os nomes dos métodos de escrita são iguais aos dos leitores correspondentes, exceto que têm um sinal de igualdade (=) anexado.

Leitores e escritores podem ser gerados implicitamente pelo sistema Ruby, incluindo-se chamadas a `attr_reader` e a `attr_writer`, respectivamente, na definição de classe. Os parâmetros para eles são os símbolos dos nomes dos atributos, como ilustrado a seguir:

```
attr_reader :sum, :total
attr_writer :sum
```

11.4.5.3 Um exemplo

A seguir está o exemplo da pilha escrito em Ruby:

```
# Stack.rb – define e testa uma pilha de tamanho máximo
#             igual a 100, implementada como uma matriz
```

```
class StackClass
# Construtor
  def initialize
    @stackRef = Array.new(100)
    @maxLen = 100
    @topIndex = -1
  end

# push method
  def push(number)
    if @topIndex == @maxLen
      puts "Error in push - stack is full"
    else
      @topIndex = @topIndex + 1
      @stackRef[@topIndex] = number
     end
     end

# pop method
  def pop
     if empty
       puts "Error in pop - stack is empty"
     else
       @topIndex = @topIndex - 1
     end
   end

# top method
   def top
     if empty
       puts "Error in top - stack is empty"
     else
       @stackRef[@topIndex]
     end
   end

# empty method
  def empty
    @topIndex == -1
  end
end  # da classe Stack

# Testa código para StackClass
myStack = StackClass.new
myStack.push(42)
myStack.push(29)
puts "Top element is (should be 29): #{myStack.top}"
myStack.pop
puts "Top element is (should be 42): #{myStack.top}"
myStack.pop
# O pop a seguir deve produzir uma
```

```
#  mensagem de erro - a pilha está vazia
myStack.pop
```

Lembre-se de que a notação #{variável} converte o valor da variável em uma cadeia, a qual é então inserida na cadeia na qual aparece. Essa classe define uma estrutura de pilha que pode armazenar objetos de qualquer tipo.

11.4.5.4 Avaliação

Lembre-se de que, em Ruby, tudo é objeto e os vetores são na realidade vetores de referências a objetos. Isso claramente torna essa pilha mais flexível que os exemplos similares em C++ e Java. Além disso, simplesmente passando o tamanho máximo desejado para o construtor, os objetos dessa classe podem ter qualquer tamanho máximo. Visto que as matrizes em Ruby têm tamanho dinâmico, a classe poderia ser modificada para implementar objetos da pilha que não são restritos a tamanho algum, exceto aquele imposto pela capacidade de memória da máquina. Como os nomes de classe e variáveis de instância têm formas diferentes, Ruby apresenta uma ligeira vantagem, em termos de legibilidade, em relação às outras linguagens discutidas nesta seção.

11.5 TIPOS DE DADOS ABSTRATOS PARAMETRIZADOS

Em geral, é conveniente ser capaz de parametrizar tipos de dados abstratos. Por exemplo, devemos ser capazes de projetar um tipo de dados abstrato pilha que possa armazenar quaisquer tipos escalares, em vez de ter de escrever uma abstração de pilha separada para cada tipo escalar diferente. Note que isso é relevante apenas para linguagens estaticamente tipadas. Em uma linguagem dinamicamente tipada, como Ruby, qualquer pilha pode armazenar implicitamente elementos de qualquer tipo. De fato, diferentes elementos da pilha poderiam ser de tipos distintos. Nas três seções seguintes, são discutidas as capacidades de C++, Java 5.0 e C# 2005 para construir tipos de dados abstratos parametrizados.

11.5.1 C++

Para tornar o exemplo da classe pilha em C++ da Seção 11.4.1 genérico no tamanho da pilha, apenas a função construtora precisa ser modificada, como no código:

```
Stack(int size) {
  stackPtr = new int  [size];
  maxLen = size - 1;
  topSub = -1;
}
```

A declaração para um objeto pilha agora pode aparecer como a seguir:

```
Stack stk(150);
```

A definição de classe para `Stack` pode incluir ambos os construtores, de forma que os usuários podem usar a pilha de tamanho padrão ou especificar outro.

O tipo do elemento da pilha pode ser genérico, tornando a classe uma classe **template**. Então, o tipo do elemento pode ser um parâmetro de *template*. A definição da classe *template* para um tipo pilha é:

```
#include <iostream.h>
template <typename Type>  // Type é o parâmetro de template
class Stack {
  private:
    Type *stackPtr;
    int maxLen;
    int topSub;
  public:
// Um construtor para pilhas de 100 elementos
    Stack() {
      stackPtr = new Type [100];
      maxLen = 99;
      topSub = -1;
    }
// Um construtor para determinado número de elementos
    Stack(int size) {
      stackPtr = new Type [size];
      maxLen = size - 1;
      topSub = -1;
    }
    ~Stack() {delete stackPtr;};  // Um destrutor
    void push(Type number) {
      if (topSub == maxLen)
        cout << "Error in push-stack is full\n";
      else stackPtr[++ topSub] = number;
    }
    void pop() {
      if (empty())
        cout << "Error in pop-stack is empty\n";
      else topSub --;
    }
    Type top() {
      if (empty())
        cerr << "Error in top--stack is empty\n";
      else
        return  (stackPtr[topSub]);
    }
    int  empty() {return  (topSub == -1);}
}
```

As classes *template* C++ são instanciadas para se tornarem classes tipadas em tempo de compilação. Por exemplo, uma instância da classe *template* `Stack`, assim como uma instância da classe tipada, podem ser criadas com a seguinte declaração:

```
Stack<int> myIntStack;
```

Contudo, se uma instância da classe *template* `Stack` já tiver sido criada para o tipo **`int`**, a classe tipada não precisará ser criada.

11.5.2 Java 5.0

Java 5.0 suporta uma forma de tipos de dados abstratos parametrizados na qual os parâmetros genéricos devem ser classes. Lembre-se de que tais parâmetros foram brevemente discutidos no Capítulo 9.

Os tipos genéricos mais comuns são de coleção, como `LinkedList` e `ArrayList`, que estavam na biblioteca de classes Java antes de o suporte para tipos genéricos ser adicionado. Os tipos de coleção originais armazenavam instâncias da classe `Object`; portanto, poderiam armazenar quaisquer objetos (mas não tipos primitivos). Logo, os tipos de coleção sempre foram capazes de armazenar múltiplos tipos (desde que fossem classes). Existiam três problemas relacionados a isso. Primeiro, sempre que um objeto era removido da coleção, ele precisava ser convertido no tipo apropriado. Segundo, não existia verificação de erros quando elementos eram adicionados à coleção. Isso significava que, uma vez criada a coleção, objetos de qualquer classe poderiam ser adicionados a ela, mesmo que a coleção fosse destinada a armazenar apenas objetos `Integer`. Terceiro, os tipos de coleção não podiam armazenar tipos primitivos. Então, para armazenar valores **`int`** em um `ArrayList`, o valor primeiro tinha de ser colocado em uma instância da classe `Integer`. Por exemplo, considere o código:

```
//* Cria um objeto da classe ArrayList
ArrayList myArray = new ArrayList();
//* Cria um elemento
myArray.add(0, new Integer(47));
//* Obtém o primeiro elemento
Integer myInt = (Integer)myArray.get(0);
```

No Java 5.0, as classes de coleção, cuja mais usada é `ArrayList`, tornaram-se classes genéricas. Essas classes são instanciadas chamando-se **`new`** no construtor da classe e passando a ele o parâmetro genérico entre os sinais de menor e maior (<>). Por exemplo, a classe `ArrayList` pode ser instanciada para armazenar objetos `Integer` com a sentença:

```
ArrayList <Integer> myArray = new ArrayList <Integer>();
```

Essa nova classe resolve dois dos problemas das coleções anteriores à Java 5.0. Apenas objetos da classe `Integer` podem ser colocados na coleção `myArray`. Além disso, não existe a necessidade de converter um objeto que esteja sendo removido da coleção.

Java 5.0 ainda inclui coleções genéricas para listas encadeadas, filas e conjuntos.

Os usuários também podem definir classes genéricas em Java 5.0. Por exemplo, poderíamos ter o seguinte:

```
public class MyClass<T> {
  . . .
}
```

Essa classe poderia ser instanciada com o seguinte:

```
MyClass<String> myString;
```

Existem alguns inconvenientes nessas classes genéricas definidas pelo usuário. Por exemplo, elas não podem armazenar primitivos. Segundo, os elementos não podem ser indexados. Eles devem ser adicionados às coleções genéricas definidas pelo usuário com o método `add`. A seguir, implementamos o exemplo de pilha genérica usando um `ArrayList`. Observe que o último elemento de um `ArrayList` é encontrado por meio do método `size`, o qual retorna o número de elementos na estrutura. Os elementos são excluídos da estrutura com o método `remove`. A seguir está a classe genérica:

```
import  java.util.*;
public class  Stack2<T> {
  private ArrayList<T> stackRef;
  private int  maxLen;
  public Stack2() { // Um construtor
    stackRef = new ArrayList<T> ();
    maxLen = 99;
  }
  public void push(T newValue) {
    if (stackRef.size() == maxLen)
      System.out.println("Error in push-stack is full");
    else
      stackRef.add(newValue);
  }
  public void pop() {
    if (empty())
      System.out.println("Error in pop-stack is empty");
    else
     stackRef.remove(stackRef.size() - 1);
  }
  public T top() {
    if (empty()) {
      System.out.println("Error in top-stack is empty");
      return null; }
    else
      return (stackRef.get(stackRef.size() - 1));
  }
  public boolean empty() {return (stackRef.isEmpty());}
```

Essa classe poderia ser instanciada para o tipo `String` com o seguinte:

```
Stack2<String> myStack = new Stack2<String>();
```

Lembre-se de que, no Capítulo 9, descrevemos que Java 5.0 suporta classes curinga. Por exemplo, `Collection<?>` é uma classe curinga para todas as classes de coleção. Isso permite que um método seja escrito de forma a aceitar qualquer tipo de coleção como parâmetro. Como uma coleção pode ser genérica, a classe `Collection<?>` é, em certo sentido, um tipo genérico de uma classe genérica.

Algum cuidado deve ser tomado com objetos do tipo curinga. Por exemplo, como os componentes de determinado objeto desse tipo têm um tipo, outros objetos de outros tipos não podem ser adicionados à coleção. Por exemplo, considere

```
Collection<?> c = new ArrayList<String>();
```

Seria inválido usar o método `add` para colocar algo nessa coleção, a menos que seu tipo fosse `String`.

Em Java 5.0, é fácil definir uma classe genérica que funcione apenas para um conjunto restrito de tipos. Por exemplo, uma classe pode declarar uma variável para o tipo genérico e chamar um método, como `compareTo`, por meio dessa variável. Se a classe é instanciada para um tipo que não inclui um método `compareTo`, não pode ser usada. Para evitar que uma classe genérica seja instanciada para um tipo que não suporta `compareTo`, ela poderia ser definida com o seguinte parâmetro genérico:

```
<T extends Comparable>
```

`Comparable` é a interface na qual `compareTo` é declarado. Se esse tipo genérico é usado em uma definição de classe, a classe não pode ser instanciada para qualquer tipo que não implemente `Comparable`. A escolha da palavra reservada **`extends`** parece estranha aqui, mas seu uso está relacionado ao conceito de subtipo. Aparentemente, os projetistas de Java não quiseram acrescentar outra palavra reservada mais conotativa à linguagem.

11.5.3 C# 2005

Como no caso de Java, a primeira versão de C# definia classes de coleção que armazenavam objetos de qualquer classe. Eram elas `ArrayList`, `Stack` e `Queue`. Essas classes tinham os mesmos problemas que as classes de coleção de Java antes da versão 5.0.

As classes genéricas foram adicionadas a C# em sua versão 2005. As cinco coleções genéricas predefinidas são `Array`, `List`, `Stack`, `Queue` e `Dictionary` (que implementa dispersões). Exatamente como em Java 5.0, essas classes eliminam os problemas de permitir tipos mistos em coleções e exigir conversões quando os objetos são removidos das coleções.

Como em Java 5.0, os usuários podem definir classes genéricas em C# 2005. Uma capacidade das coleções genéricas definidas pelo usuário de C# é que qualquer uma delas permite que seus elementos sejam indexados (acessados por meio de índices). Apesar de os índices normalmente serem inteiros, uma alternativa é usar cadeias como índices.

Uma capacidade que Java 5.0 fornece e que C# 2005 não fornece são as classes curinga.

11.6 CONSTRUÇÕES DE ENCAPSULAMENTO

As primeiras cinco seções deste capítulo discutiram tipos de dados abstratos, que são encapsulamentos mínimos. Esta seção descreve os encapsulamentos de múltiplos tipos, necessários para programas maiores.

11.6.1 Introdução

Quando o tamanho de um programa ultrapassa a fronteira de algumas poucas mil linhas de código, dois problemas práticos se tornam evidentes. Do ponto de vista do programador, ter tal programa aparecendo como uma única coleção de subprogramas ou definições de tipos de dados abstratos não impõe nele um nível adequado de organização para mantê-lo intelectualmente gerenciável. O segundo problema prático para programas grandes é a recompilação. Para os relativamente pequenos, recompilar o programa inteiro após cada modificação não é dispendioso. Para programas grandes, o custo da recompilação é significativo. Então, existe uma necessidade óbvia de encontrar maneiras de evitar a recompilação das partes de um programa não afetadas por uma mudança. A solução para esses problemas é organizar os programas em coleções de códigos e dados logicamente relacionados, de modo que cada uma possa ser compilada sem a recompilação do restante do programa. Um **encapsulamento** é tal coleção.

Encapsulamentos são colocados em bibliotecas e disponibilizados para reuso em programas que não são aqueles para os quais eles foram escritos. As pessoas têm escrito programas com mais de alguns milhares de linhas nos últimos 50 anos, então as técnicas para fornecer encapsulamentos têm evoluído.

Em linguagens que permitem subprogramas aninhados, os programas podem ser organizados por definições de subprogramas aninhadas dentro de subprogramas maiores que os utilizam. Isso pode ser feito em Python e Ruby. Como discutido no Capítulo 5, entretanto, esse método de organizar programas, que usa escopo estático, está longe de ser o ideal. Logo, mesmo em linguagens que permitem subprogramas aninhados, eles não são usados como construção de organização de encapsulamento principal.

11.6.2 Encapsulamento em C

C não fornece suporte completo para tipos de dados abstratos, apesar de tanto tipos de dados abstratos quanto encapsulamentos de múltiplos tipos poderem ser simulados. Em C, uma coleção de funções e definições de dados relacionadas pode ser colocada em um arquivo, que pode ser compilado independentemente. Esse arquivo, que age como uma biblioteca, tem uma implementação de suas entidades. A interface para ele, incluindo os dados, tipos e declarações de funções, é colocada em um arquivo separado chamado de **arquivo de cabeçalho**. Representações de tipo podem ser ocultas ao serem declaradas no arquivo de cabeçalho como ponteiros para tipos struct. As definições completas para tais tipos struct precisam aparecer apenas no arquivo de implementação.

O arquivo de cabeçalho, em forma de fonte, e a versão compilada do arquivo de implementação são repassados para os clientes. Quando tal biblioteca é usada, o arquivo de cabeçalho é incluído no código cliente por meio de uma especificação de pré-processador `#include`, de forma que as referências às funções e aos dados no código cliente possam ser verificadas em relação aos seus tipos. A especificação `#include` também documenta o fato de que o programa cliente depende do arquivo de implementação da biblioteca. Essa estratégia separa a especificação e a implementação de um encapsulamento.

Embora esses encapsulamentos funcionem, eles criam algumas inseguranças. Por exemplo, um usuário poderia simplesmente recortar e colar as definições do arquivo

de cabeçalho no programa cliente, em vez de usar `#include`. Isso funcionaria, porque `#include` copia o conteúdo de seu arquivo operando para o arquivo no qual `#include` aparece. Entretanto, existem dois problemas nessa estratégia. Primeiro, a documentação de dependência entre o programa cliente e a biblioteca (e seu arquivo de cabeçalho) é perdida. Segundo, suponha que um usuário copie o arquivo de cabeçalho para seu programa. Então, o autor da biblioteca altera os arquivos de cabeçalho e de implementação. Depois disso, o usuário utiliza o novo arquivo de implementação com o cabeçalho antigo. Por exemplo, uma variável `x` poderia ter sido definida como do tipo **`int`** no arquivo de cabeçalho antigo, que o código cliente ainda usa, apesar de o código de implementação ter sido recompilado com o novo arquivo de cabeçalho, que define `x` como **`float`**. Então, o código de implementação foi compilado com `x` como um **`int`**, mas o código cliente foi compilado com `x` como um **`float`**. O ligador não detecta esse erro.

Então, é responsabilidade do usuário garantir que tanto o arquivo de cabeçalho quanto o de implementação estejam atualizados. Em geral, isso é feito com um utilitário `make`.

Outro inconveniente dessa estratégia são os problemas inerentes aos ponteiros e a possível confusão com a atribuição e as comparações de ponteiros.

11.6.3 Encapsulamento em C++

C++ fornece dois tipos de encapsulamento: arquivos de cabeçalho e de implementação podem ser definidos como em C, ou podem ser estabelecidos cabeçalhos de classes e definições. Em função da complexa interação dos *templates* C++ e da compilação separada, os arquivos de cabeçalho das bibliotecas de *template* de C++ geralmente incluem definições completas de recursos, em vez de apenas declarações de dados e protocolos de subprogramas; isso ocorre em parte devido ao uso do ligador C para programas C++.

Quando classes que não contêm *templates* são usadas para encapsulamentos, o arquivo de cabeçalho de classe tem apenas os protótipos das funções membros, com as definições das funções fornecidas fora da classe, em um arquivo de código, como no último exemplo na Seção 11.4.1.4. Isso separa claramente a interface da implementação.

Um problema de projeto de linguagem que resulta do fato de ter classes, mas nenhuma construção de encapsulamento generalizada, é que, às vezes, quando são definidas operações que utilizam duas classes de objetos diferentes, as operações não pertencem naturalmente a nenhuma das classes. Por exemplo, suponha que tivéssemos um tipo de dados abstrato para matrizes e outro para vetores e precisássemos de uma operação de multiplicação entre um vetor e uma matriz. O código de multiplicação deve ter acesso aos membros de dados tanto da classe do vetor quanto da classe da matriz, mas nenhuma das classes é a casa natural para o código. Além disso, independentemente de qual for escolhido, o acesso aos membros do outro é um problema. Em C++, essas situações podem ser manipuladas permitindo-se que funções não membro sejam "amigas" (*friends*) de uma classe. Funções amigas têm acesso às entidades privadas da classe em que são declaradas amigas. Para a operação de multiplicação entre matriz e vetor, uma solução em C++ é definir a operação fora das classes da matriz e do vetor, mas defini-la como amiga de ambas. O seguinte código esqueleto ilustra esse cenário:

```
class Matrix;  //** Uma declaração de classe
class Vector {
  friend Vector multiply(const Matrix&, const Vector&);
  . . .
};
class Matrix {  //** A definição de classe
  friend Vector multiply(const Matrix&, const Vector&);
  . . .
};
//** A função que usa tanto objetos do tipo Matrix quanto do tipo
Vector
Vector multiply(const Matrix& m1, const Vector& v1) {
  . . .
}
```

Além de funções, classes inteiras podem ser definidas como amigas de uma classe; então, todos os membros privados da classe são visíveis para todos os membros da classe amiga.

11.6.4 Montagens em C#

C# inclui uma construção de encapsulamento maior que uma classe. A construção é aquela usada por todas as linguagens de programação .NET: a montagem. Montagens são construídas por compiladores .NET. Uma aplicação .NET consiste em uma ou mais montagens. Uma **montagem** é um arquivo[4] que aparece para programas aplicativos como uma única biblioteca de ligação dinâmica (`.dll`)[5] ou um executável (`.exe`). Uma montagem define um módulo, o qual pode ser desenvolvido separadamente. Ela inclui vários componentes diferentes. Um dos principais é seu código de programação, que está em uma linguagem intermediária, compilado a partir de sua linguagem-fonte. No .NET, a linguagem intermediária é chamada Common Intermediate Language (CIL). Ela é usada por todas as linguagens .NET. Como seu código está em CIL, uma montagem pode ser usada em qualquer arquitetura, dispositivo ou sistema operacional. Quando executada, a CIL tem compilação *just-in-time* para código nativo da arquitetura na qual reside.

Além do código CIL, uma montagem .NET contém metadados que descrevem toda classe que define, assim como todas as classes externas que utiliza. Uma montagem também inclui uma lista de todas as montagens nela referenciadas e um número de versão.

[4]Uma montagem pode consistir em qualquer número de arquivos.

[5]Uma **biblioteca de ligação dinâmica (DLL)** é uma coleção de classes e métodos individualmente ligados a um programa que executa quando é necessário durante a execução. Logo, apesar de um programa ter acesso a todos os recursos em determinada DLL, apenas as partes realmente usadas são carregadas e ligadas ao programa. As DLLs fazem parte do ambiente de programação do Windows desde a primeira aparição do seu sistema operacional. Contudo, as DLLs do .NET são muito diferentes das dos sistemas Windows anteriores.

No mundo .NET, a montagem é a unidade básica de implantação de software. Montagens podem ser privadas – disponíveis para apenas uma aplicação – ou públicas – disponíveis para qualquer aplicação.

Conforme mencionado, C# tem um modificador de acesso, **`internal`**. Um membro interno (**`internal`**) de uma classe é visível para todas as classes na montagem em que aparece.

Java tem uma estrutura de arquivos semelhante a uma montagem, chamada Java Archive (JAR). Ela também é usada para implantação de sistemas de software Java. As JARs são construídas com o utilitário Java `jar`, não por um compilador.

11.7 NOMEAÇÃO DE ENCAPSULAMENTOS

Temos considerado os encapsulamentos como contêineres sintáticos para recursos de software logicamente relacionados – em particular, tipos de dados abstratos. A finalidade desses encapsulamentos é oferecer um modo de organizar os programas em unidades lógicas para compilação. Isso permite que partes dos programas sejam recompiladas após alterações isoladas. Existe outro tipo de encapsulamento necessário para construir programas grandes: um encapsulamento de nomeação.

Um programa grande normalmente é escrito por muitos desenvolvedores, que trabalham de maneira independente, talvez até em localizações geográficas diferentes. Isso requer que as unidades lógicas do programa sejam independentes, mas que ainda seja possível trabalhar em conjunto. Além disso, cria um problema de nomeação: como desenvolvedores que trabalham independentemente criam nomes para suas variáveis, métodos e classes sem acidentalmente usar nomes já utilizados por outro programador em uma parte diferente do mesmo sistema de software?

As bibliotecas são a origem do mesmo tipo de problemas de nomeação. Nas últimas duas décadas, grandes sistemas de software se tornaram mais dependentes de bibliotecas de software de suporte. Praticamente todo software escrito em linguagens de programação contemporâneas requer o uso de bibliotecas padrão grandes e complexas e de bibliotecas específicas para a aplicação em questão. Esse uso disseminado de múltiplas bibliotecas tem exigido novos mecanismos para gerenciar nomes. Por exemplo, quando um desenvolvedor adiciona novos nomes a uma biblioteca existente ou cria uma nova biblioteca, ele não pode usar um novo nome que entre em conflito com um já definido em um programa aplicativo do cliente ou em alguma outra biblioteca. Sem a ajuda de algum processador de linguagem, isso é praticamente impossível, porque não existe uma maneira de o autor da biblioteca saber quais nomes um programa cliente usa ou quais são definidos por outras bibliotecas que o programa cliente possa usar.

Encapsulamentos de nomeação definem escopos de nome que ajudam a evitar conflitos. Cada biblioteca pode criar seu próprio encapsulamento para evitar que seus nomes entrem em conflito com aqueles definidos em outras bibliotecas ou em código cliente. Cada parte lógica de um sistema de software pode criar um encapsulamento com o mesmo propósito.

Encapsulamentos de nomeação são encapsulamentos lógicos, pois não precisam ser fisicamente adjacentes. Várias coleções de código diferentes podem ser colocadas no mesmo espaço de nomes, mesmo sendo armazenadas em diferentes lugares. Nas seções

a seguir, descrevemos brevemente os usos dos encapsulamentos de nomeação em C++, Java e Ruby.

11.7.1 Espaços de nome em C++

C++ inclui uma especificação, **namespace**, que ajuda os programas a gerenciarem o problema de espaços de nomes globais. É possível colocar cada biblioteca em seu próprio espaço de nomes e qualificá-los no programa, com o nome do espaço de nomes, quando eles são usados fora desse espaço. Por exemplo, suponha que existe um arquivo de cabeçalho de um tipo de dados abstrato que implementa pilhas. Se existe a preocupação de que algum outro arquivo de biblioteca possa definir um nome usado no tipo de dados abstrato pilha, o arquivo que define a pilha pode ser colocado em seu próprio espaço de nomes. Isso é feito colocando-se todas as declarações para a pilha em um bloco de espaço de nomes, como a seguir:

```
namespace myStackSpace {
  // Declarações de pilha
}
```

O arquivo de implementação para o tipo de dados abstrato pode referenciar os nomes declarados no arquivo de cabeçalho com o operador de resolução de escopo, `::`, como em

```
myStackSpace::topSub
```

O arquivo de implementação também pode aparecer em uma especificação de bloco de espaço de nomes idêntica àquela usada no arquivo de cabeçalho, tornando diretamente visíveis todos os nomes declarados no arquivo de cabeçalho. Isso é definitivamente mais simples, mas um pouco menos legível, porque é menos óbvio onde um nome específico no arquivo de implementação é declarado.

O código cliente pode acessar os nomes no espaço de nomes do arquivo de cabeçalho de uma biblioteca de três maneiras. Uma é qualificá-los a partir da biblioteca com o nome do espaço de nomes. Por exemplo, uma referência à variável topSub poderia aparecer como segue:

```
myStackSpace::topSub
```

Esse é exatamente o modo como o código de implementação poderia referenciá-la se o arquivo de implementação não estivesse no mesmo espaço de nomes.

As outras duas estratégias usam a diretiva **using**. Ela pode ser utilizada para qualificar nomes individuais de um espaço de nomes, como em

```
using myStackSpace::topSub;
```

que torna `topSub` visível, mas não quaisquer outros nomes do espaço de nomes `MyStackSpace`.

A diretiva **using** pode também ser usada para qualificar todos os nomes de um espaço de nomes, como em:

```
using namespace myStackSpace;
```

O código que inclui essa diretiva pode acessar diretamente os nomes definidos no espaço de nomes, como em

```
p = topSub;
```

Os espaços de nomes são um recurso complicado de C++; aqui, apresentamos apenas os aspectos mais simples.

C# inclui espaços de nomes bastante parecidos com aqueles de C++.

11.7.2 Pacotes em Java

Java inclui uma construção de encapsulamento de nomeação: o pacote. Pacotes podem conter mais de uma definição de tipo[6], e os tipos em um pacote são amigos parciais uns dos outros. *Parcial* aqui significa que as entidades públicas, protegidas (consulte o Capítulo 12) ou que não têm um especificador de acesso definidas em um tipo em um pacote são visíveis para todos os outros tipos no pacote.

Diz-se que as entidades sem modificadores de acesso têm **escopo de pacote**, porque são visíveis por todo o pacote. Java, dessa forma, tem menos necessidade para declarações amigas explícitas e não inclui funções amigas ou classes amigas de C++.

Os recursos definidos em um arquivo são especificados como em um pacote em particular, com uma declaração de pacote, como em

```
package stkpkg;
```

A declaração de pacote deve aparecer como a primeira linha do arquivo. Os recursos de cada arquivo que não incluem uma declaração de pacote são implicitamente colocados no mesmo pacote não nomeado.

Os clientes de um pacote podem referenciar os tipos definidos no pacote com nomes completamente qualificados. Por exemplo, se o pacote `stkpkg` tem uma classe chamada `myStack`, essa classe pode ser referenciada em um cliente de `stkpkg` como `stkpkg.myStack`. Do mesmo modo, uma variável no objeto `myStack` chamada `topSub` poderia ser referenciada como `stkpkg.myStack.topSub`. Como essa estratégia pode rapidamente se tornar desajeitada quando os pacotes são aninhados, Java fornece a declaração **`import`**, a qual permite referências mais curtas aos nomes de tipo definidos em um pacote. Por exemplo, suponha que o cliente inclua o seguinte:

```
import stkpkg.myStack;
```

Agora, a classe `myStack` pode ser referenciada apenas por seu nome. Para poder acessar todos os nomes de tipo no pacote, um asterisco pode ser usado na sentença **`import`**, no lugar do nome de tipo. Por exemplo, se quiséssemos importar todos os tipos em `stkpkg`, poderíamos usar o seguinte:

```
import stkpkg.*;
```

[6] Aqui, tipo significa uma classe ou uma interface.

Note que o **import** de Java é apenas um mecanismo de abreviação. Nenhum outro recurso externo oculto se torna disponível com **import**. Na verdade, em Java nada é implicitamente oculto se puder ser encontrado pelo compilador ou pelo carregador de classes (usando o nome do pacote e a variável de ambiente `CLASSPATH`).

O **import** de Java documenta as dependências do pacote no qual ele aparece com os pacotes nomeados no **import**. Essas dependências são menos óbvias quando **import** não é usado.

11.7.3 Módulos em Ruby

As classes de Ruby servem como encapsulamentos de nomeação, como o fazem as classes de outras linguagens que suportam programação orientada a objetos. Ruby tem um encapsulamento de nomeação adicional, chamado **módulo**. Normalmente, os módulos definem coleções de métodos e constantes. Então, eles são convenientes para encapsular bibliotecas de métodos e constantes, cujos nomes estão em um espaço de nomes separado, de forma que não existem conflitos de nomes com outros nomes em um programa que usa o módulo. Módulos são diferentes das classes, pois não podem ser instanciados ou estendidos por herança nem definem variáveis. Métodos definidos em um módulo incluem o nome do módulo em seus nomes. Por exemplo, considere o seguinte esqueleto de definição de módulo:

```
module MyStuff
  PI = 3.14159265
  def MyStuff.mymethod1(p1)
  . . .
  end
  def MyStuff.mymethod2(p2)
  . . .
  end
end
```

Supondo que o módulo `MyStuff` seja armazenado em seu próprio arquivo, um programa que queira usar a constante e os métodos de `MyStuff` deve primeiro obter acesso ao módulo. Isso é feito com o método `require`, que recebe como parâmetro o nome do arquivo na forma de um literal do tipo cadeia. Então, as constantes e métodos do módulo podem ser acessados pelo nome dele. Considere o seguinte código, que usa nosso módulo de exemplo, `MyStuff`, armazenado no arquivo chamado `MyStuffMod`:

```
require 'myStuffMod'
. . .
MyStuff.mymethod1(x)
. . .
```

Os módulos são discutidos mais detalhadamente no Capítulo 12.

RESUMO

O conceito de tipos de dados abstratos, e seu uso em projeto de programas, foi um marco no desenvolvimento da programação como uma disciplina de engenharia. Apesar de o conceito ser relativamente simples, seu uso não se tornou conveniente e seguro até que linguagens foram projetadas para suportá-lo.

Os dois recursos principais de tipos de dados abstratos são o empacotamento de objetos de dados, com suas operações associadas, e a ocultação de informações. Uma linguagem pode suportar tipos de dados abstratos ou simulá-los com encapsulamentos mais gerais.

A abstração de dados em C++ é fornecida pelas classes. As classes são tipos, e as instâncias podem ser dinâmicas da pilha ou dinâmicas do monte. Uma função membro (método) pode ter sua definição completa aparecendo na classe, ou ter apenas o protocolo dado na classe e a definição colocada em outro arquivo, que pode ser compilado separadamente. Classes em C++ têm duas cláusulas, cada uma prefixada com um modificador de acesso: `private` ou `public`. Tanto construtores quanto destrutores podem ser dados em definições de classes. Objetos alocados do monte devem ser explicitamente liberados com **`delete`**.

Como em C++, as abstrações de dados de Objective-C são classes. Classes são tipos e todos são dinâmicos do monte. As declarações de métodos devem aparecer em seções de interface das classes e as definições de método devem aparecer em seções de implementação. Os construtores são chamados de **inicializadores**; eles devem ser chamados explicitamente. Variáveis de instância podem ser privadas ou públicas. O acesso aos métodos não pode ser restrito. As chamadas de método usam sintaxe semelhante à usada por Smalltalk. Objective-C suporta propriedades, e métodos de acesso para propriedades podem ser fornecidos pelo compilador.

Abstrações de dados em Java são similares às de C++, exceto que todos os objetos em Java são alocados do monte e acessados por meio de variáveis de referência. Além disso, todos os objetos são coletados quando viram lixo. Em vez de ter modificadores de acesso anexados às cláusulas, em Java os modificadores aparecem em declarações individuais (ou definições).

C# suporta tipos de dados abstratos tanto com classes quanto com estruturas. Suas estruturas são tipos de valores e não suportam herança. As classes C# são similares àquelas de Java.

Ruby suporta tipos de dados abstratos com suas classes. As classes de Ruby diferem daquelas da maioria das outras linguagens, pois são dinâmicas – membros podem ser adicionados, apagados ou modificados durante a execução.

C++, Java 5.0 e C# 2005 permitem que seus tipos de dados abstratos sejam parametrizados – Ada por meio de seus pacotes genéricos, C++ por suas classes *template* e Java 5.0 e C# 2005 com suas classes e interfaces de coleção e suas classes genéricas definidas pelo usuário.

Para suportar a construção de programas grandes, algumas linguagens contemporâneas incluem construções de encapsulamento de múltiplos tipos, que podem conter uma coleção de tipos relacionados logicamente. Um encapsulamento também pode fornecer controle de acesso às suas entidades. Os encapsulamentos fornecem ao programador um método para organizar programas, facilitando a recompilação.

C++, C#, Java e Ruby fornecem encapsulamentos de nomeação. Para Ada e Java, eles são chamados de pacotes; para C++ e C#, são espaços de nomes; para Ruby, são módulos. Parcialmente devido à disponibilidade dos pacotes, Java não tem funções nem classes amigas.

QUESTÕES DE REVISÃO

1. Quais são os dois tipos de abstrações em linguagens de programação?
2. Defina *tipo de dados abstrato*.
3. Quais são as vantagens das duas partes da definição de *tipo de dados abstrato*.
4. Quais são os requisitos de projeto de linguagem para uma linguagem que suporte tipos de dados abstratos?
5. Quais são as questões de projeto de linguagem para tipos de dados abstratos?
6. De onde os objetos em C++ são alocados?
7. Em que diferentes locais a definição de uma função membro em C++ pode aparecer?
8. Qual é o propósito de um construtor em C++?
9. Quais são os tipos de retorno válidos de um construtor?
10. Onde todos os métodos Java são definidos?
11. Como as instâncias de classe em C++ são criadas?
12. Como são especificadas as seções de interface e de implementação de uma classe Objective-C?
13. As classes Objective-C são tipos?
14. Qual é o nível de acesso de métodos Objective-C?
15. Qual é a origem da sintaxe de chamadas de método em Objective-C?
16. Quando os construtores são chamados implicitamente em Objective-C?
17. Por que propriedades são melhores que especificar uma variável de instância como pública?
18. De onde são alocadas as instâncias de classe Java?
19. Por que Java não tem destrutores?
20. Onde todos os métodos Java são definidos?
21. Onde as classes Java são alocadas?
22. Por que destrutores não são necessários com tanta frequência em Java como acontece em C++?
23. O que é uma função amiga? O que é uma classe amiga?
24. Cite uma razão pela qual Java não tem funções amigas ou classes amigas.

25. Descreva as diferenças fundamentais entre structs e classes em C#.
26. Como um objeto struct é criado em C#?
27. Explique as três razões pelas quais os métodos de acesso para tipos privados são melhores que tornar os tipos públicos.
28. Quais são as diferenças entre structs em C++ e structs em C#?
29. Qual é o nome de todos os construtores em Ruby?
30. Qual é a diferença fundamental entre as classes de Ruby e aquelas de C++ e de Java?
31. Como as instâncias das classes *template* de C++ são criadas?
32. Descreva os dois problemas que aparecem na construção de programas grandes e que levaram ao desenvolvimento de construções de encapsulamento.
33. Que problemas podem ocorrer ao se usar C para definir tipos de dados abstratos?
34. O que é um espaço de nomes em C++ e qual é o seu propósito?
35. O que é um pacote Java e qual é o seu propósito?
36. Descreva uma montagem no .NET.
37. Que elementos podem aparecer em um módulo Ruby?

PROBLEMAS

1. Alguns engenheiros de software acreditam que todas as entidades importadas devem ser qualificadas pelo nome da unidade de programa exportadora. Você concorda? Justifique sua resposta.
2. Suponha que alguém projetou um tipo de dados abstrato pilha no qual a função `top` retorna um caminho de acesso (ou ponteiro), em vez de retornar uma cópia do elemento do topo. Essa não é uma abstração de dados verdadeira. Por quê? Dê um exemplo que ilustre o problema.
3. Escreva uma análise das similaridades e das diferenças entre pacotes em Java e espaços de nomes em C++.
4. Discuta as vantagens das propriedades em C#, em relação a escrever métodos de acesso em C++ ou Java.
5. Explique os perigos da estratégia de C para encapsulamento.
6. Por que C++ não eliminou os problemas discutidos no Problema 5?
7. Quais são as vantagens e desvantagens da estratégia Objective-C de distinguir métodos de classe de métodos de instância sintaticamente?
8. De que maneiras as chamadas de método em C++ são mais ou menos legíveis que as de Objective-C?

9. Quais são os argumentos contra e a favor do projeto de Objective-C que define que o acesso a método não pode ser restrito?
10. Por que os destrutores não são necessários em Java, mas são essenciais em C++?
11. Quais são os argumentos a favor e contra a política de internalizar métodos?
12. Descreva uma situação na qual uma estrutura C# seja preferível a uma classe C#.
13. Explique por que encapsulamentos de nomeação são importantes para desenvolver programas grandes.
14. Descreva as três maneiras pelas quais um cliente pode referenciar um nome a partir de um espaço de nomes C++.
15. O espaço de nomes da biblioteca padrão de C#, `System`, não é implicitamente disponível para os programas em C#. Você acha que essa é uma boa ideia? Argumente.
16. Quais são as vantagens e desvantagens da capacidade de modificar objetos em Ruby?
17. Compare os pacotes de Java com os módulos de Ruby.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Projete um tipo de dados abstrato para uma matriz com elementos inteiros em uma linguagem que você conheça, incluindo operações para adição, subtração e multiplicação de matrizes.
2. Projete um tipo de dados abstrato fila para elementos de ponto flutuante em uma linguagem que você conheça, incluindo operações para inserir, remover e esvaziar a fila. A operação de remoção remove o elemento e retorna o seu valor.
3. Modifique a classe C++ para o tipo abstrato mostrado na Seção 11.4.2 para usar uma representação de lista encadeada e teste-a com o mesmo código que aparece neste capítulo.
4. Modifique a classe Java para o tipo abstrato mostrado na Seção 11.4.4 para usar uma representação de lista encadeada e teste-a com o mesmo código que aparece neste capítulo.
5. Escreva um tipo de dados abstrato para números complexos, incluindo operações para adição, subtração, multiplicação, divisão, extração de cada uma das partes de um número complexo e construção de um número complexo a partir de duas constantes, variáveis ou expressões de ponto flutuante. Use C++, Java, C# ou Ruby.
6. Escreva um tipo de dados abstrato para filas cujos elementos armazenem nomes de 10 caracteres. Os elementos da fila devem ser dinamicamente alocados do monte. As operações da fila são a inserção, a remoção e o esvaziamento. Use C++, Java, C# ou Ruby.
7. Escreva um tipo de dados abstrato para uma fila cujos elementos possam ser de qualquer tipo primitivo. Use Java 5.0, C# 2005 ou C++.

8. Escreva um tipo de dados abstrato para uma fila cujos elementos incluam tanto uma cadeia de 20 caracteres quanto uma prioridade inteira. Essa fila deve ter os seguintes métodos: inserir, que recebe uma cadeia e um inteiro como parâmetros; remover, que retorna a cadeia da fila que tem a prioridade mais alta; e esvaziar. A fila não deve ser mantida na ordem de prioridade dos elementos; então, operações de remoção devem sempre buscar toda a fila.
9. Um deque é uma fila de duas extremidades, com operações de adição e remoção de elementos em ambas. Modifique a solução do Exercício de programação 7 para implementar um deque.
10. Escreva um tipo de dados abstrato para números racionais (um numerador e um denominador). Inclua um construtor e métodos para obter o numerador e o denominador, soma, subtração, multiplicação, divisão, teste de igualdade e exibição. Use Java, C#, C++ ou Ruby.

12

Suporte para programação orientada a objetos

12.1 Introdução

12.2 Programação orientada a objetos

12.3 Questões de projeto para linguagens orientadas a objetos

12.4 Suporte para programação orientada a objetos em linguagens específicas

12.5 Implementação de construções orientadas a objetos

12.6 Reflexão

Este capítulo começa com uma breve introdução à programação orientada a objetos, seguida por uma discussão sobre as principais questões de projeto para herança e vinculação dinâmica. A seguir são discutidos o suporte para programação orientada a objetos em Smalltalk, C++, Objective-C, Java, C# e Ruby. Depois disso há um panorama da implementação de vinculações dinâmicas de chamadas de métodos em linguagens orientadas a objetos. A última seção discute a reflexão.

12.1 INTRODUÇÃO

Linguagens que suportam programação orientada a objetos são bastante usadas atualmente. De COBOL até LISP, incluindo praticamente todas as linguagens entre elas, apareceram dialetos que suportam programação orientada a objetos. C++ e Objective-C suportam programação procedural e orientada a dados, além da programação orientada a objetos. CLOS, uma versão orientada a objetos de LISP (Paepeke, 1993), também suporta programação funcional. Algumas das linguagens mais novas projetadas para programação orientada a objetos não suportam outros paradigmas, mas ainda assim empregam algumas das estruturas imperativas básicas e têm a aparência das linguagens imperativas mais antigas. Entre essas estão Java e C#. É desafiador classificar Ruby: é uma linguagem orientada a objetos no sentido de que todos os dados são objetos, mas é uma linguagem híbrida, porque é possível usá-la para programação procedural. Por fim, existe a linguagem orientada a objetos pura, mas um tanto anticonvencional: Smalltalk. Ela foi a primeira linguagem a oferecer suporte completo para programação orientada a objetos. Os detalhes do suporte para programação orientada a objetos variam muito entre as linguagens, e esse é o principal assunto deste capítulo.

Este capítulo é como uma continuação do anterior. Esse relacionamento reflete o fato de que programação orientada a objetos é, basicamente, uma aplicação do princípio da abstração para tipos de dados abstratos. Especificamente, em programação orientada a objetos, as partes comuns de uma coleção de tipos de dados abstratos similares são fatoradas e colocadas em um novo tipo. Os membros da coleção herdam essas partes comuns do novo tipo. Esse recurso é a herança, que está no centro da programação orientada a objetos e das linguagens que a suportam.

A outra característica da programação orientada a objetos, a vinculação dinâmica de chamadas a métodos, também é discutida extensivamente neste capítulo.

Embora a programação orientada a objetos seja suportada por algumas das linguagens funcionais, por exemplo, CLOS, OCaml e F#, essas linguagens não são discutidas aqui.

12.2 PROGRAMAÇÃO ORIENTADA A OBJETOS

12.2.1 Introdução

O conceito de **programação orientada a objetos** tem suas raízes em SIMULA 67, mas não foi completamente desenvolvido[1] até a evolução de Smalltalk que resultou em Smalltalk 80 (em 1980, é claro). De fato, algumas pessoas consideram Smalltalk o modelo base para uma linguagem de programação puramente orientada a objetos. Uma linguagem orientada a objetos deve fornecer suporte para três recursos-chave de linguagem: tipos de dados abstratos, herança e vinculação dinâmica de chamadas a métodos. Tipos abstratos de dados foram discutidos em detalhes no Capítulo 11; então, este capítulo foca em herança e vinculação dinâmica.

12.2.2 Herança

Há muito tempo existe uma pressão sobre os desenvolvedores de software para aumentarem sua produtividade. Essa pressão é intensificada pela contínua redução no custo do hardware de computadores. No final dos anos 1980, ficou claro para muitos desenvolvedores que uma das oportunidades mais promissoras para o aumento da produtividade em sua profissão era o reuso de software. Tipos abstratos de dados, com encapsulamento e controles de acesso, eram candidatos óbvios para reúso. O problema do reúso de tipos de dados abstratos é que, em praticamente todos os casos, os recursos e capacidades do tipo existente não são exatamente os necessários para o novo uso. O tipo antigo exige pelo menos algumas pequenas modificações, que podem ser difíceis, pois exigem que a pessoa responsável por elas entenda parte, se não todo, do código existente. Em muitos casos, quem faz a modificação não é o autor original do programa. Além disso, as modificações frequentemente exigem mudanças em todos os programas clientes.

Um segundo problema da programação com tipos de dados abstratos é que as definições de tipos são independentes e no mesmo nível.[2] Esse projeto normalmente impossibilita a organização de um programa para combinar com o espaço do problema tratado por ele. Em muitos casos, o problema subjacente tem categorias de objetos relacionados, tanto como irmãos (similares uns aos outros) quanto em um relacionamento pais e filhos (com um relacionamento de descendência).

A herança oferece uma solução tanto para o problema de modificação oriundo do reuso de tipos abstratos de dados quanto para o da organização de programas. Se um novo tipo abstrato de dados pode herdar os dados e as funcionalidades de algum tipo existente, e se também é permitido que ele modifique algumas das entidades e adicione novas, o reuso é amplamente facilitado, sem exigir mudanças no tipo abstrato de dados reutilizado. Os programadores podem começar com um tipo abstrato de dados já existente e projetar um descendente modificado dele para atender a um novo requisito do problema. Além disso, a herança fornece um framework para a definição de hierarquias de classes relacionadas que pode refletir os relacionamentos de descendência no espaço do problema.

[1] Embora SIMULA 67 tivesse classes, os membros definidos dentro delas não ficavam ocultos para código de fora.

[2] Isso é semelhante às funções de um programa em C, as quais também são independentes e em um único nível.

Os tipos abstratos de dados em linguagens orientadas a objetos, seguindo a nomenclatura de SIMULA 67, são normalmente chamados de **classes**. Assim como as instâncias de tipos abstratos de dados, as instâncias de classes são chamadas de **objetos**. Uma classe definida por meio de herança de outra classe é chamada de **classe derivada**, **subclasse** ou **classe filho**. Uma classe da qual a nova é derivada é sua **classe base**, **superclasse** ou **classe pai**. Os subprogramas que definem as operações em objetos de uma classe são chamados de **métodos**. As chamadas a métodos são algumas vezes denominadas **mensagens**. A coleção completa de métodos de uma classe é chamada de **protocolo de mensagens** ou **interface de mensagens**. As computações em um programa orientado a objetos são especificadas por mensagens enviadas de objetos para outros objetos ou, em alguns casos, para classes.

Métodos são semelhantes aos subprogramas. Ambos são coleções de código que realiza alguma computação. Ambos podem receber parâmetros e retornam resultados.

Passar uma mensagem é diferente de chamar um subprograma. Um subprograma normalmente processa os dados passados a ele como parâmetro por seu chamador, ou é acessado de forma não local ou globalmente. Uma mensagem enviada para um objeto é um pedido para executar um de seus métodos. Pelo menos alguns dos dados nos quais o método deve operar fazem parte do próprio objeto. Os objetos têm métodos que definem processos que eles podem executar em si mesmos. Como os objetos são de tipos de dados abstratos, essas devem ser as únicas maneiras de manipular seus dados. Um subprograma define um processo que pode executar em quaisquer dados enviados a ele (ou disponibilizados de forma não local ou globalmente).

Como um exemplo simples de herança, considere o seguinte: suponha que temos uma classe chamada `Vehicles`, a qual tem variáveis para ano, cor e marca. Uma especialização, ou subclasse, natural disso seria `Truck`, que poderia herdar as variáveis de `Vehicle`, mas adicionaria variáveis para capacidade de reboque e número de rodas. A Figura 12.1 mostra um diagrama simples para indicar o relacionamento entre a classe `Vehicle` e a classe `Truck`, no qual a seta aponta para a classe pai.

Uma classe derivada pode diferir de seu pai de várias maneiras.[3] A seguir estão as diferenças mais comuns entre uma classe pai e suas subclasses:

1. A subclasse pode adicionar variáveis e/ou métodos àqueles herdados da classe pai.
2. A subclasse pode modificar o comportamento de um ou mais métodos herdados. Um método modificado tem o mesmo nome, e geralmente o mesmo protocolo, daquele que está sendo modificado.

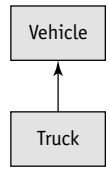


FIGURA 12.1
Um exemplo simples de herança.

[3]Se uma subclasse não difere de sua classe pai, ela obviamente não serve a propósito algum.

3. A classe pai pode definir que algumas de suas variáveis ou métodos têm acesso privado, o que significa que não serão visíveis na subclasse.

Diz-se que o novo método **sobrescreve** o método herdado, chamado então de **método sobrescrito**. A finalidade de um método sobrescrito é fornecer uma operação na subclasse que seja semelhante à da classe pai, mas personalizada para objetos da subclasse. Por exemplo, uma classe pai, `Bird`, poderia ter um método `draw` que extraísse uma ave (bird) genérica. Uma subclasse de `Bird`, chamada `Waterfowl`, poderia sobrescrever o método `draw` herdado de `Bird` para extrair uma ave aquática (waterfowl) genérica, talvez um pato.

As classes podem ter dois tipos de métodos e dois de variáveis. Os mais usados são chamados de **métodos de instância** e **variáveis de instância**. Cada objeto de uma classe tem seu próprio conjunto de variáveis de instância, que armazenam o estado do objeto. A única diferença entre dois objetos da mesma classe é o estado de suas variáveis de instância.[4] Por exemplo, uma classe para carros poderia ter variáveis de instância para cor, marca, modelo e ano. Métodos de instância operam apenas nos objetos da classe. **Variáveis de classe** pertencem à classe, em vez de ao seu objeto; então, existe apenas uma cópia para a classe. Por exemplo, se quiséssemos contar o número de instâncias de uma classe, o contador não poderia ser uma variável de instância – precisaria ser uma variável de classe. **Métodos de classe** podem realizar operações na classe e também em objetos da classe. Eles podem ser chamados prefixando-se seus nomes com o nome da classe ou com uma variável que referencie uma de suas instâncias. Se uma classe define um método de classe, esse método pode ser chamado mesmo que não exista nenhuma instância da classe. Um método de classe poderia ser usado para criar uma instância da classe.

Se uma nova classe é uma subclasse de uma única classe pai, o processo de derivação é chamado de **herança simples**. Se uma classe tem mais de uma classe pai, o processo é chamado de **herança múltipla**. Quando algumas classes estão relacionadas por herança simples, seus relacionamentos umas com as outras podem ser mostrados em uma árvore de derivação. Os relacionamentos de classes em uma herança múltipla podem ser mostrados em um grafo de derivação. Isso é apresentado na Figura 12.5, na Seção 12.4.2.2.

Uma desvantagem da herança como forma de aumentar a possibilidade de reuso é que ela cria dependências entre classes em uma hierarquia. Isso contraria uma das vantagens dos tipos abstratos de dados, a independência de um tipo em relação aos outros. É claro, nem todos os tipos abstratos de dados podem ser completamente independentes. Mas, em geral, essa é uma de suas características mais positivas. Entretanto, pode ser difícil, se não impossível, aumentar a reutilização de tipos abstratos de dados sem criar dependências entre alguns deles. Além disso, em muitos casos, as dependências naturalmente espelham dependências no espaço do problema subjacente.

No Capítulo 11 são discutidos os controles de acesso para variáveis e métodos (juntos, frequentemente denominados membros) em uma classe. Os membros privados são visíveis dentro da classe, enquanto os membros públicos são visíveis também para clientes da classe. A herança apresenta uma nova categoria de visibilidade possível, as

[4] Isso não é verdade em Ruby, que permite que diferentes objetos da mesma classe sejam diferentes de outras maneiras.

subclasses. Os membros privados de uma classe base não são visíveis para as subclasses, mas os membros públicos são. O terceiro nível de acessibilidade, protegido, permite que os membros de uma classe base sejam visíveis para as subclasses, mas não para clientes.

12.2.3 Vinculação dinâmica

A terceira característica essencial (além dos tipos de dados abstratos e da herança) das linguagens de programação orientadas a objetos é um tipo de polimorfismo[5] fornecido pela vinculação dinâmica de mensagens às definições de métodos. Isso às vezes é chamado de **despacho dinâmico**. Considere a seguinte situação: há uma classe base, `A`, que define um método `draw` que desenha alguma figura associada à classe base. Uma segunda classe, `B`, é definida como uma subclasse de `A`. Objetos dessa nova classe também precisam de um método `draw` parecido com o que é fornecido por `A`, mas um pouco distinto, porque eles são ligeiramente diferentes. Então, a subclasse sobrescreve o método `draw` herdado. Se um cliente de `A` e de `B` tivesse uma variável que fosse uma referência para objetos da classe `A`, essa referência também poderia apontar para objetos da classe `B`, transformando-se em uma referência **polimórfica**. Se o método `draw`, definido em ambas as classes, é chamado por meio da referência polimórfica, o sistema de tempo de execução deve determinar, durante a execução, qual método deve ser chamado, o de `A` ou o de `B` (determinando qual é o tipo do objeto atualmente referenciado pela referência).[6] A Figura 12.2 mostra essa situação.

O polimorfismo é uma parte natural de qualquer linguagem orientada a objetos estaticamente tipada. De certo modo, ele torna uma linguagem estaticamente tipada um pouco tipada dinamicamente; esse "pouco" reside em algumas vinculações de chamadas de método a métodos. O tipo de uma variável polimórfica é efetivamente dinâmico.

A estratégia que acabamos de descrever não é a única maneira de projetar referências polimórficas. Uma alternativa, utilizada em Objective-C, é descrita na Seção 12.4.3.

Um propósito da vinculação dinâmica é permitir que os sistemas de software sejam estendidos mais facilmente durante o desenvolvimento e a manutenção. Suponha

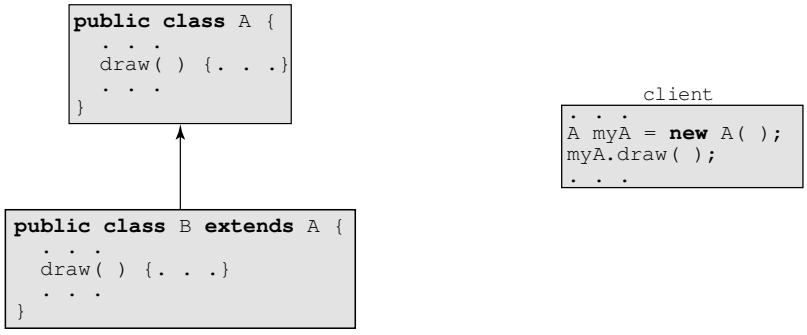


FIGURA 12.2
Vinculação dinâmica.

[5] O polimorfismo é definido no Capítulo 9.

[6] A vinculação dinâmica de chamadas de método a métodos às vezes é denominada *polimorfismo dinâmico*.

que temos um catálogo de carros usados implementado como uma classe `Car` e uma subclasse para cada carro no catálogo. As subclasses contêm uma imagem do carro e informações específicas sobre o veículo. Os usuários podem fazer pesquisas com um programa que mostra as imagens e as informações de cada carro pesquisado. A imagem de cada veículo (e suas informações) inclui um botão em que o usuário pode clicar se estiver interessado nele. Depois que o usuário acaba de examinar o catálogo, o sistema imprime as imagens e as informações sobre os carros que o interessam. Um modo de implementar esse sistema é colocar uma referência para cada carro (subclasse de `Car`) de interesse em uma lista que possa armazenar referências para a classe base, `Car`. Assim, quando o usuário terminasse sua pesquisa, informações sobre seus carros de interesse poderiam ser impressas para que ele pudesse estudar e comparar as alternativas da lista. Evidentemente, o catálogo de carros mudaria frequentemente, o que exigiria alterações correspondentes nas subclasses de `Car`. Contudo, alterações na coleção de subclasses não exigiriam quaisquer outras mudanças no sistema.

Em alguns casos, o projeto de uma hierarquia de herança resulta em uma ou mais classes tão altas na hierarquia que uma instanciação delas não faria sentido. Por exemplo, suponha que um programa definisse uma classe de construção (chamada `Building`) e uma coleção de subclasses para tipos específicos de construções, como catedrais góticas francesas – `French_Gothic_Cathedrals`. Provavelmente, não faria sentido ter um método de desenho (`draw`) implementado em `Building`. Mas, como todas as suas classes descendentes devem ter tais métodos, o protocolo (mas não o corpo) dele é incluído em `Building`. Tal método é chamado de **método abstrato** (*método puramente virtual* em C++). Uma classe que inclua pelo menos um método abstrato é chamada de **classe abstrata** (*classe base abstrata* em C++). Normalmente, tal classe não pode ser instanciada, porque alguns de seus métodos são declarados, mas não são definidos (eles não têm corpos). Qualquer subclasse de uma classe abstrata a ser instanciada deve fornecer implementações (definições) para todos os métodos abstratos herdados.

12.3 QUESTÕES DE PROJETO PARA LINGUAGENS ORIENTADAS A OBJETOS

Diversas questões devem ser consideradas quando projetamos os recursos de linguagem de programação para suportar herança e vinculação dinâmica. Aquelas que consideramos mais importantes são discutidos nesta seção.

12.3.1 A exclusividade dos objetos

Um projetista de linguagem totalmente comprometido com o modelo de objetos de computação projeta um sistema de objetos que agrupa todos os outros conceitos de tipo. Para ele, tudo, desde um simples inteiro escalar até um sistema de software completo, é um objeto. As vantagens dessa escolha são a elegância e a uniformidade pura da linguagem e de seu uso. A principal desvantagem é que operações simples devem ser feitas pelo processo de passagem de mensagens, o que em geral as torna mais lentas que operações similares em um modelo imperativo, no qual instruções únicas de máquina implementam tais operações simples. Nesse modelo mais puro de computação orientada a objetos,

todos os tipos são classes. Não existe distinção entre classes predefinidas e classes definidas pelo usuário. Na verdade, todas as classes são tratadas da mesma forma e toda a computação é realizada por meio de passagem de mensagens.

Uma alternativa ao uso exclusivo de objetos, comum em linguagens imperativas nas quais o suporte à programação orientada a objetos foi adicionado, é a seguinte: manter a coleção completa de tipos da linguagem imperativa de base e adicionar o modelo de objetos. Essa estratégia resulta em uma linguagem maior, cuja estrutura de tipos pode ser confusa para novos usuários da linguagem.

Outra alternativa ao uso exclusivo de objetos é ter uma estrutura de tipos em um estilo imperativo para tipos primitivos escalares, mas implementar todos os estruturados como objetos. Essa escolha fornece uma velocidade de operações em valores primitivos comparável àquelas esperadas no modelo imperativo.

12.3.2 As subclasses são subtipos?

Se uma linguagem permite programas nos quais uma variável de uma classe pode ser substituída por uma variável de uma de suas classes ancestrais em qualquer situação, sem causar erros de tipo e sem alterar o comportamento do programa, essa linguagem suporta o **princípio da substituição**. Em tal linguagem, se a classe `B` é derivada da classe `A`, então `B` tem tudo de `A`, e o comportamento de um objeto da classe `B`, quando usado no lugar de um objeto da classe `A`, é idêntico ao de um objeto da classe `A`.[7] Quando isso é verdade, `B` é um **subtipo** de `A`. Embora uma subclasse que seja um subtipo de sua classe pai deva expor todos os membros expostos por sua classe pai, ela pode ter membros que não estão na classe pai e ainda ser um subtipo.

Os subtipos de Ada são exemplos de subtipos predefinidos. Por exemplo:

```
subtype Small_Int is Integer range -100..100;
```

Variáveis do tipo `Small_Int` têm todas as operações das variáveis do tipo `Integer`, mas podem armazenar apenas um subconjunto dos valores possíveis em `Integer`. Além disso, toda variável `Small_Int` pode ser utilizada em qualquer lugar onde uma variável `Integer` pode ser usada. Ou seja, toda variável `Small_Int` é, em certo sentido, uma variável `Integer`.

A definição de subtipo claramente proíbe haver entidades públicas na classe pai que não são públicas na subclasse. Então, o processo de derivação para subtipos deve exigir que entidades públicas da classe pai sejam herdadas como entidades públicas na subclasse.

Nem todas as subclasses são subtipos, nem todos os subtipos são subclasses. Por exemplo, uma subclasse não pode ser um subtipo se mudar o comportamento de um de seus métodos sobrescrevedores. Além disso, uma classe que não é subclasse de outra pode ser um subtipo dessa classe, definindo-se os mesmos membros, tanto em termos de tipos como de comportamento. Um subtipo herda interfaces e comportamento, enquanto uma subclasse herda a implementação, principalmente para promover o reúso de código.

[7] Há um problema teórico fundamental nesse último requisito: em geral, não é possível determinar se os comportamentos de dois programas são idênticos.

A maioria das linguagens de tipo estático que suportam programação orientada a objetos é projetada de modo que as subclasses sejam subtipos, a menos que o programador projete especificamente uma subclasse que tenha comportamento diferente de sua classe pai.

Uma questão óbvia é: subclasses são subtipos teóricos ou práticos? Provavelmente não é comum definir uma subclasse cujos métodos sobrescrevedores preservam os protocolos de tipo de seus métodos sobrescritos correspondentes, mas não seus efeitos. Portanto, esse não é um problema prático frequente. Contudo, exigir que todas as subclasses sejam subtipos, se houvesse uma maneira razoavelmente simples de impor isso, colocaria a herança em uma base teórica mais sólida.

12.3.3 Herança simples e múltipla

Outra questão de projeto simples para linguagens orientadas a objetos é: a linguagem permite herança múltipla (além da simples)? Talvez a resposta não seja tão óbvia. O propósito da herança múltipla é permitir que uma nova classe herde de duas ou mais classes.

Como a herança múltipla pode ser muito útil, por que um projetista de linguagem não a incluiria? As razões estão em duas categorias: complexidade e eficiência. A complexidade adicional é ilustrada em diversos problemas. Primeiro, note que, se uma classe tem duas classes pais não relacionadas e nenhuma delas define um nome que é definido na outra, não há problema. Entretanto, suponha que uma subclasse chamada `C` herde tanto da classe `A` quanto da classe `B` e que ambas definam um método que pode ser herdado, chamado `display`. Se `C` precisar referenciar ambas as versões de `display`, como pode fazê-lo? Esse problema de ambiguidade é ainda mais complicado quando as duas classes pais definem métodos nomeados de forma idêntica e um deles, ou ambos, precisa ser sobrescrito na subclasse.

Outra questão surge se `A` e `B` são derivadas de um pai comum, `Z`, e `C` tem tanto `A` quanto `B` como classes pais. Essa situação é chamada de herança **diamante** ou herança **compartilhada**. Nesse caso, tanto `A` quanto `B` devem incluir as variáveis herdadas de `Z`. Suponha que `Z` inclua uma variável que pode ser herdada, chamada `sum`. A questão é se `C` deve herdar ambas as versões de `sum` ou apenas uma – e, se for apenas uma, qual delas? Podem existir situações de programação nas quais apenas uma das duas deve ser herdada, e outras nas quais ambas devem ser. Um problema similar ocorre quanto `A` e `B` herdam um método de `Z` e ambas sobrescrevem esse método. Se um cliente de `C`, que herda os dois métodos sobrescrevedores, chama o método, qual deles é chamado (ou ambos devem ser chamados)? A herança diamante é mostrada na Figura 12.3.

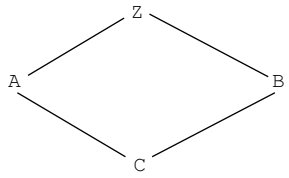


FIGURA 12.3
Um exemplo de herança diamante.

A questão da eficiência pode ser mais aparente do que real. Em C++, por exemplo, suportar herança múltipla requer apenas um acesso à matriz adicional e uma operação de adição extra para cada chamada a método dinamicamente vinculada, ao menos em algumas arquiteturas de máquina (Stroustrup, 1994, p. 270). Apesar de essa operação ser necessária mesmo se o programa não usa herança múltipla, é um pequeno custo adicional.

O uso de herança múltipla pode facilmente levar a organizações de programa complexas. Muitos daqueles que tentaram usar herança múltipla descobriram que projetar classes para serem usadas como múltiplos pais é difícil. E as dificuldades não estão restritas às criadas pelo desenvolvedor inicial. Em algum momento posterior, uma classe pode ser usada, por outro desenvolvedor, como um dos pais de uma nova classe. A manutenção de sistemas que usam herança múltipla pode ser um problema mais sério, já que essa herança leva a dependências mais complexas entre as classes. Nem todos os profissionais acreditam que os benefícios da herança múltipla valem o esforço adicional de projetar e manter um sistema que a utiliza.

Uma interface é um pouco parecida com uma classe abstrata; seus métodos são declarados, mas não definidos. Interfaces não podem ser instanciadas. Elas são usadas como uma alternativa à herança múltipla.[8] As interfaces oferecem algumas das vantagens da herança múltipla, mas têm menos desvantagens. Por exemplo, os problemas da herança diamante são evitados quando são usadas interfaces em vez de herança múltipla.

12.3.4 Alocação e liberação de objetos

Existem duas questões de projeto relacionadas à alocação e à liberação de objetos. A primeira é o local de onde os objetos são alocados. Se eles se comportam como tipos de dados abstratos, então podem ser alocados de qualquer lugar. Isso significa que podem ser alocados da pilha de tempo de execução ou explicitamente criados no monte com um operador ou função, como **`new`**. Se eles são todos dinâmicos da pilha, existe a vantagem de haver um método de criação e acesso uniforme por meio de ponteiros ou variáveis de referência. Esse projeto simplifica a operação de atribuição para objetos, tornando-a, em todos os casos, apenas uma mudança em um valor de ponteiro ou de referência. Isso também permite que as referências a objetos sejam desreferenciadas implicitamente, simplificando a sintaxe de acesso.

Se os objetos são dinâmicos da pilha, existe um problema em potencial relacionado aos subtipos. Se a classe `B` é filha da classe `A` e `B` é um subtipo de `A`, um objeto do tipo `B` pode ser atribuído a uma variável do tipo `A`. Por exemplo, se `b1` é uma variável do tipo `B` e `a1` é uma variável do tipo `A`, então

```
a1 = b1;
```

é uma sentença permitida. Se `a1` e `b1` são referências a objetos dinâmicos do monte, não há problema – a atribuição é uma simples atribuição de ponteiros. Entretanto, se `a1` e `b1` são dinâmicas da pilha, então são variáveis de valor e, se for atribuído o valor do objeto,

[8] As interfaces apareceram inicialmente em Java, cujos projetistas reconheceram as dificuldades surgidas com o uso de herança múltipla.

deve ser copiado para o espaço do objeto alvo. Se `B` adicionar um campo de dados aos herdados de `A`, então `a1` não terá espaço suficiente na pilha para todos os membros de `b1`. O excesso será simplesmente truncado, podendo ser confuso para programadores que escrevem ou usam o código. Esse truncamento é chamado de **fatiamento de objetos** (*object slicing*). O exemplo a seguir e a Figura 12.4 ilustram o problema.

```
class A {
  int x;
  . . .
};
class B : A {
  int y;
  . . .
}
```

A segunda questão é relacionada aos casos nos quais os objetos são alocados do monte: a liberação é implícita, explícita ou de ambos os tipos? Se a liberação é implícita, é necessário algum método implícito de recuperação de armazenamento. Se a liberação pode ser explícita, surge a questão acerca da possibilidade da criação de ponteiros soltos ou referências soltas.

12.3.5 Vinculação estática e dinâmica

Como discutido na Seção 12.2.3, a vinculação dinâmica de mensagens a métodos é parte essencial da programação orientada a objetos. A questão é se todas as vinculações de mensagens a métodos são dinâmicas. A alternativa é permitir ao usuário especificar se uma vinculação deve ser estática ou dinâmica. A vantagem é que vinculações estáticas são mais rápidas. Então, se uma vinculação não precisa ser dinâmica, por que pagar o preço?

12.3.6 Classes aninhadas

Um dos principais objetivos das classes aninhadas é a ocultação de informação. Se uma nova classe é necessária para apenas uma classe, não existe razão para defini-la de for-

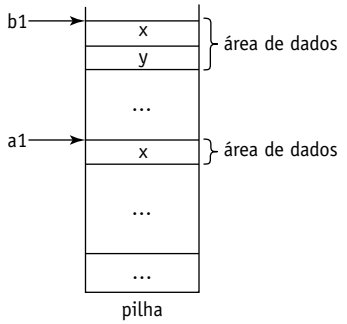


FIGURA 12.4
Um exemplo de fatiamento de objetos.

ma que seja vista por outras. Nessa situação, a nova classe pode ser aninhada dentro da classe que a utiliza. Em alguns casos, a nova está aninhada dentro de um subprograma, em vez de diretamente em outra classe.

A classe na qual a nova está aninhada é chamada de **classe aninhadora**. As questões de projeto mais óbvias associadas ao aninhamento de classes são relacionadas à visibilidade. Especificamente, uma delas é: quais dos membros da classe aninhadora são visíveis para a classe aninhada? A outra questão importante é a oposta: quais dos membros da classe aninhada são visíveis para a classe aninhadora?

12.3.7 Inicialização de objetos

A questão da inicialização diz respeito a se, e como, os objetos são inicializados com valores ao serem criados. Isso é mais complicado do que pode parecer. Uma questão é se os objetos devem ser inicializados manualmente ou por meio de algum mecanismo implícito. Quando um objeto de uma subclasse é criado, a inicialização associada do membro herdado da classe pai é implícita ou o programador deve lidar explicitamente com ela?

12.4 SUPORTE PARA PROGRAMAÇÃO ORIENTADA A OBJETOS EM LINGUAGENS ESPECÍFICAS

12.4.1 Smalltalk

Muitos acreditam que Smalltalk é a linguagem de programação orientada a objetos definitiva. Ela foi a primeira a incluir suporte completo para esse paradigma. Logo, é natural começar por ela um levantamento do suporte linguístico para programação orientada a objetos.

12.4.1.1 Características gerais

Em Smalltalk, a noção de objeto é verdadeiramente universal. Praticamente tudo é um objeto, desde coisas tão simples como a constante inteira `2` até um sistema de tratamento de arquivos complexo. Como objetos, eles são tratados uniformemente. Todos têm memória local, habilidade de processamento inerente, capacidade de comunicar com outros objetos e possibilidade de herdar métodos e variáveis de instância dos ancestrais. Classes não podem ser aninhadas em Smalltalk.

Toda computação se dá por meio de mensagens, mesmo uma simples operação aritmética. Por exemplo, a expressão `x + 7` é implementada como o envio da mensagem `+` para `x` (a fim de executar o método `+`), enviando `7` como parâmetro. Essa operação retorna um novo objeto numérico com o resultado da adição.

Respostas às mensagens têm a forma de objetos e são usadas para retornar informações solicitadas ou computadas, ou apenas para confirmar que o serviço solicitado foi concluído.

Todos os objetos Smalltalk são alocados do monte e referenciados por variáveis de referência, as quais são implicitamente desreferenciadas. Não existe uma sentença ou

operação de liberação explícita. Assim, toda liberação é implícita, usando um processo de coleta de lixo para recuperação de armazenamento.

Em Smalltalk, os construtores devem ser explicitamente chamados quando um objeto é criado. Uma classe pode ter múltiplos construtores, mas cada um deles deve ter um nome único.

Classes Smalltalk não podem ser aninhadas em outras classes.

Diferentemente das linguagens híbridas, como C++ e Objective-C, Smalltalk foi projetada para apenas um paradigma de desenvolvimento de software – orientado a objetos. Além disso, ela não adota nenhuma das aparências das linguagens imperativas. Sua pureza de propósito é refletida em sua elegância simples e uniformidade de projeto.

Há um exemplo de programa em Smalltalk no Capítulo 2.

12.4.1.2 Herança

Uma subclasse Smalltalk herda todos os membros de sua superclasse. A subclasse também pode ter suas próprias variáveis de instância, com nomes distintos dos das variáveis nas classes ancestrais. Por fim, a subclasse pode definir novos métodos e redefinir métodos já existentes em uma classe ancestral. Quando uma subclasse tem um método cujo nome e protocolo são os mesmos da classe ancestral, o da subclasse oculta o da classe ancestral. O acesso ao método oculto é fornecido por um prefixo à mensagem com a pseudovariável **`super`**. O prefixo faz o método iniciar a busca a partir da superclasse, em vez de localmente.

Como os membros em uma classe pai não podem ser ocultados a partir das subclasses, as subclasses podem ser (e normalmente são) subtipos.

Smalltalk não suporta herança múltipla.

12.4.1.3 Vinculação dinâmica

A vinculação dinâmica de mensagens a métodos em Smalltalk funciona assim: uma mensagem para um objeto gera uma busca por um método correspondente na classe à qual o objeto pertence. Se a busca falha, ela continua na superclasse dessa classe e assim por diante, até a classe de sistema, `Object`, que não tem uma superclasse. `Object` é a raiz da árvore de derivação de classes na qual toda classe é um nó. Se nenhum método é encontrado em algum lugar nessa cadeia, ocorre um erro. É importante lembrar que essa busca de método é dinâmica – ela ocorre quando a mensagem é enviada. Smalltalk não vincula mensagens a métodos estaticamente, independentemente das circunstâncias.

A única verificação de tipos em Smalltalk é dinâmica, e o único erro de tipo ocorre quando uma mensagem é enviada para um objeto que não tem um método correspondente, seja localmente, seja por herança. Esse é um conceito de verificação de tipos diferente daqueles da maioria das outras linguagens. A verificação de tipos de Smalltalk tem o simples objetivo de garantir que uma mensagem case com algum método.

Variáveis em Smalltalk não são tipadas; qualquer nome pode ser vinculado a qualquer objeto. Como resultado direto disso, Smalltalk suporta polimorfismo dinâmico. Todo código Smalltalk é genérico, no sentido de que os tipos das variáveis são irrelevantes, desde que sejam consistentes. O significado de uma operação (método ou operador) em uma variável é determinado pela classe do objeto ao qual a variável está vinculada.

O principal ponto desta discussão é que, desde que os objetos referenciados em uma expressão tenham métodos para as mensagens da expressão, os tipos dos objetos são irrelevantes. Isso significa que nenhum código está amarrado a um tipo em particular.

12.4.1.4 Avaliação de Smalltalk

Smalltalk é uma linguagem pequena, apesar de o sistema Smalltalk ser grande. A sintaxe da linguagem é simples e altamente regular. É um bom exemplo do poder que pode ser fornecido por uma linguagem pequena, se for construída em torno de um conceito simples, mas poderoso. No caso de Smalltalk, o conceito é que toda a programação pode ser feita com apenas uma hierarquia de classes construída por meio de herança, objetos e passagem de mensagens.

Em comparação com os programas compilados de linguagens imperativas convencionais, os programas Smalltalk equivalentes são significativamente mais lentos. Apesar de ser teoricamente interessante que a indexação de matrizes e laços possa ser fornecida dentro do modelo de passagem de mensagens, a eficiência é um fator importante na avaliação de linguagens de programação. Logo, ela será uma questão importante na maioria das discussões acerca da aplicabilidade prática de Smalltalk.

A vinculação dinâmica de Smalltalk permite que erros de tipo não sejam detectados até a execução. Pode ser escrito um programa que inclua mensagens a métodos inexistentes, e isso não será detectado até que as mensagens sejam enviadas, causando, posteriormente, no desenvolvimento, muito mais erros e reparos do que os que ocorreriam em uma linguagem estaticamente tipada. Contudo, na prática os erros de tipo não são um problema sério nos programas Smalltalk.

De um modo geral, o projeto de Smalltalk piora ao optar pela elegância da linguagem e pela aderência estrita aos princípios do suporte à programação orientada a objetos, geralmente sem levar em consideração questões práticas, em particular a eficiência de execução. Isso é mais evidente no uso exclusivo de objetos e variáveis sem tipos.

A interface com o usuário de Smalltalk teve um impacto importante na computação: o uso integrado de janelas, mouse e menus pop-up e pull-down, todos os quais apareceram primeiro em Smalltalk, dominam os sistemas de software contemporâneos.

Talvez o maior impacto de Smalltalk seja o avanço da programação orientada a objetos, que hoje é a metodologia de projeto e codificação mais amplamente usada.

12.4.2 C++

O Capítulo 2 descreve como C++ evoluiu a partir de C e SIMULA 67, com o objetivo de projeto de suportar programação orientada a objetos, mantendo compatibilidade quase completa com as versões anteriores de C. As classes C++, conforme utilizadas para suportar tipos de dados abstratos, são discutidas no Capítulo 11. O suporte à C++ para as outras questões essenciais da programação orientada a objetos é explorado nesta seção. A coleção completa de detalhes de classes, de herança e de vinculação dinâmica em C++ é extensa e complexa. Esta seção se concentra apenas nos tópicos

mais importantes, especificamente naqueles relacionados às questões de projeto descritas na Seção 12.3.

C++ foi a primeira linguagem de programação orientada a objetos bastante usada e ainda está entre as mais populares. Então, naturalmente, é aquela com a qual as outras linguagens são geralmente comparadas. Por essas razões, nossa abordagem de C++ é mais detalhada que a das outras linguagens exemplificadas neste capítulo.

12.4.2.1 Características gerais

Para manter a compatibilidade com versões anteriores de C, C++ mantém seu sistema de tipos e adiciona classes a ele. Logo, C++ tem tanto os tipos tradicionais de linguagens imperativas quanto as estruturas de classes de uma linguagem orientada a objetos. Além disso, suporta métodos e funções não relacionadas a classes específicas. Isso a torna uma linguagem híbrida, que suporta programação procedural e orientada a objetos.

Os objetos de C++ podem ser estáticos, dinâmicos da pilha ou dinâmicos do monte. A liberação explícita por meio do operador **`delete`** é necessária para objetos dinâmicos do monte, já que C++ não inclui recuperação de armazenamento implícita.

Muitas definições de classe incluem um método destrutor, implicitamente chamado quando um objeto da classe deixa de existir. O destrutor é usado para liberar memória alocada do monte referenciada pelos membros de dados. Ele também pode ser usado para gravar parte ou todo o estado antes de morrer, em geral para propósitos de depuração.

12.4.2.2 Herança

Uma classe C++ pode ser derivada de uma classe existente – sua classe pai ou classe base. Ao contrário de Smalltalk e da maioria das outras linguagens que suportam programação orientada a objetos, uma classe C++ também pode ser independente, sem uma superclasse. Na definição de uma classe derivada, o nome da classe tem o nome da classe base anexado com dois pontos (`:`), como na seguinte forma sintática:

```
class nome_classe_derivada : nome_classe_base { ... }
```

Os dados em uma definição de classe são chamados de *membros de dados*, e as funções são chamadas de *funções membro* (funções membro em outras linguagens são geralmente chamadas de métodos). Alguns ou todos os membros da classe base podem ser herdados pela classe derivada, a qual também pode adicionar novos membros e modificar funções membro herdadas.

Todos os objetos C++ devem ser inicializados antes de serem usados. Logo, todas as classes C++ incluem ao menos um método construtor que inicializa os membros de dados do novo objeto. Membros construtores são implicitamente chamados quando um objeto é criado. Se quaisquer membros de dados são ponteiros para dados alocados no monte, o construtor aloca o armazenamento.

Se uma classe é derivada de outra, os membros de dados herdados devem ser inicializados quando o objeto da classe derivada é criado. Para isso, o construtor da classe base é chamado implicitamente. Quando os dados de inicialização devem ser fornecidos

Sobre paradigmas e uma programação melhor

BJARNE STROUSTRUP

Bjarne Stroustrup é o projetista e implementador original de C++ e autor de *A Tour of C++*, *Programming: Principles and Practice Using C++*, *The C++ Programming Language*, *The Design and Evolution of C++* e muitas outras publicações. Suas áreas de pesquisa incluem sistemas distribuídos, projeto, técnicas de programação, ferramentas de desenvolvimento de software e linguagens de programação. Ativamente envolvido na padronização ANSI/ISO de C++, Dr. Stroustrup é Diretor Gerente da divisão de tecnologia da Morgan Stanley, em Nova York, EUA, professor convidado de Ciência da Computação na Universidade de Columbia e professor de pesquisa emérito de Ciência da Computação na Universidade do Texas A&M. Além disso, é membro da National Academy of Engineering, da ACM e do IEEE. Em 1993, recebeu o Prêmio Grace Murray Hopper da ACM "por seu trabalho pioneiro que levou às bases da linguagem de programação C++. Por meio dessas bases e de esforços contínuos do Dr. Stroustrup, C++ se tornou uma das linguagens de programação mais influentes na história da computação". (*ano da entrevista: 2002*)

PARADIGMAS DE PROGRAMAÇÃO

Quais as vantagens e desvantagens do paradigma orientado a objetos (POO)? Primeiro, deixe-me explicar o que quero dizer com POO – muitos pensam que "orientado a objetos" é simplesmente um sinônimo de "bom". Se fosse, não seriam necessários outros paradigmas. O segredo do OO é o uso de hierarquia de classes que fornecem comportamento polimórfico por meio de algum equivalente aproximado às funções virtuais. Para se ter uma orientação a objetos adequada, é importante evitar o acesso direto aos dados em tal hierarquia e usar apenas uma interface funcional bem projetada.

Além dessas qualidades bem documentadas, a programação orientada a objetos apresenta algumas fraquezas óbvias. Especificamente, nem todo o conceito se encaixa de modo natural em uma hierarquia de classes, e os mecanismos que suportam a programação orientada a objetos podem impor sobrecargas significativas se comparados com outras alternativas. Para muitas abstrações simples, classes que não dependem de hierarquia e vinculações em tempo de execução fornecem uma alternativa mais simples e eficiente. Além disso, quando nenhuma resolução em tempo de execução é necessária, a programação genérica dependente de polimorfismo paramétrico (em tempo de compilação) é uma estratégia que se comporta melhor e é mais eficiente.

Então, C++ é OO ou outra coisa? C++ suporta diversos paradigmas, incluindo POO, programação genérica e programação procedural. A combinação desses paradigmas define a programação multiparadigma, que tem como característica o suporte para mais de um estilo de programação ("paradigma") e suas combinações.

Você tem um miniexemplo de programação multiparadigma? Considere esta variante do exemplo clássico "coleção de formas" (originado nos primeiros dias da primeira linguagem a suportar programação orientada a objetos: Simula 67):

```
void draw_all(const vector<Shape*>& vs)
{
    for (int i = 0; i<vs.size(); ++i)
        vs[i]->draw();
}
```

Aqui, uso o contêiner genérico `vector` com o tipo polimórfico `Shape`. Ele fornece a segurança da tipagem estática e ótimo desempenho em tempo de execução. O tipo `Shape` fornece a capacidade de manipular uma forma (ou seja, qualquer objeto de uma classe derivada de `Shape`) sem recompilação.

Podemos facilmente generalizar esse procedimento para qualquer contêiner que case com os requisitos da biblioteca padrão de C++:

```
template<class C>
        void draw_all(const  C& c)
{
    typedef typename C::
        const_iterator CI;
    for (CI p = c.begin();
        p!=c.end(); ++p)
        (*p)->draw();
}
```

O uso de iteradores nos permite aplicar `draw_all()` para contêineres que não suportam índices, como a lista da biblioteca padrão.

```
vector<Shape*> vs;
list<Shape*> ls;
// . . .
draw_all(vs);
draw_all(ls);
```

Podemos generalizar isso ainda mais, para manipular qualquer sequência de elementos definidos por um par de iteradores:

```
template<class Iterator> void
draw_all(Iterator b, Iterator e)
{
    for_each(b,e,mem_fun(&Shape::draw));
}
```

Para simplificar a implementação, usei o algoritmo da biblioteca padrão `for_each`.

Poderíamos chamar essa última versão de `draw_all()` para uma lista da biblioteca padrão e um vetor:

```
list<Shape*> ls;
Shape* as[100];
// . . .
draw_all(ls.begin(),ls.end());
draw_all(as,as+100);
```

A LINGUAGEM "CERTA" PARA A TAREFA

É útil ter experiência com vários paradigmas ou seria melhor investir tempo em conhecer profundamente apenas as linguagens OO? É essencial para todos que desejam ser profissionais nas áreas relacionadas ao desenvolvimento de software conhecer diversas linguagens e paradigmas de programação. Atualmente, C++ é a melhor linguagem para programação multiparadigma e uma opção para aprender várias formas de programação. Contudo, não é uma boa ideia conhecer apenas C++, muito menos conhecer apenas uma linguagem de paradigma único. Seria como ser daltônico ou monoglota: dificilmente você saberia o que estaria perdendo. Muito da inspiração para uma boa programação vem do aprendizado e da apreciação de diferentes estilos de programação e, ainda, de ter visto como eles podem ser usados em linguagens diferentes.

Além disso, considero a programação de qualquer programa não trivial um trabalho para profissionais com uma formação sólida e ampla, e não para pessoas com um "treinamento" restrito e rápido.

para o construtor da classe base, eles são passados na chamada para o construtor do objeto derivado. Em geral, isso é feito por meio da seguinte construção:

```
Subclasse (parâmetros da subclasse) : classe_base (parâmetros da superclasse) {
. . .
}
```

Se nenhum construtor é incluído pelo desenvolvedor em uma definição de classe, o compilador inclui um construtor trivial. Esse construtor padrão chama o construtor da classe base, se ela existir.

Membros de classe podem ser privados, protegidos ou públicos. Membros privados são acessíveis apenas por funções membros e por amigos da classe. Funções independentes, funções membro e classes podem ser declaradas como amigas de uma classe e ter acesso aos membros privados de tal classe. Membros públicos são visíveis em todos os lugares. Membros protegidos são parecidos com os membros privados, exceto nas classes derivadas, cujo acesso é descrito a seguir. Classes derivadas podem modificar a acessibilidade de seus membros herdados. A forma sintática completa de uma classe derivada é a seguinte:

```
class nome_classe_derivada : modo_de_derivação nome_classe_base
 {declarações de membros de dados e de funções membro};
```

O modo de derivação pode ser **público** ou **privado**.[9] (Não confunda derivação pública e privada com membros públicos e privados.) Os membros públicos e protegidos de uma classe base são também públicos e protegidos, respectivamente, em uma classe publicamente derivada. Em uma classe privadamente derivada, os membros públicos e protegidos da classe base são privados. Assim, em uma hierarquia de classes, uma classe derivada privada corta o acesso a todos os membros de todas as classes ancestrais para todas as classes sucessoras. Membros privados de uma classe base são herdados por uma classe derivada, mas não são visíveis para os membros de tal classe derivada e, portanto, não têm utilidade alguma lá. As derivações privadas oferecem a possibilidade de que uma subclasse tenha membros com acesso diferente daquele dos mesmos membros na classe pai. Considere o seguinte exemplo:

```
class base_class {
  private:
    int a;
    float x;
  protected:
    int b;
    float y;
  public:
    int c;
    float z;
};

class subclass_1 : public base_class {. . .};
class subclass_2 : private base_class {. . .};
```

[9] Ele também pode ser protegido, mas essa opção não é discutida aqui.

Na subclass_1, b e y são protegidos, e c e z são públicos. Na subclass_2, b, y, c e z são privados. Nenhuma classe derivada de subclass_2 pode ter membros com acesso a qualquer um dos membros de base_class. Os membros de dados a e x em base_class não são acessíveis nem em subclass_1, nem em subclass_2.

Note que subclasses privadamente derivadas não podem ser subtipos. Por exemplo, se a classe base tem um membro de dados público, na derivação privada esse membro de dados seria privado na subclasse. Logo, se um objeto da subclasse fosse substituído por um objeto da classe base, os acessos a tais membros de dados seriam inválidos no objeto da subclasse. Contudo, as subclasses derivadas públicas podem ser (e normalmente são) subtipos.

Na derivação privada de classe nenhum membro da classe pai é implicitamente visível para as instâncias da classe derivada. Qualquer membro que deve ser tornado visível precisa ser reexportado na classe derivada. Essa reexportação na verdade faz um membro deixar de ser oculto, mesmo que a derivação seja privada. Por exemplo, considere a seguinte definição de classe:

```
class subclass_3 : private base_class {
  base_class :: c;
  . . .
}
```

Agora, instâncias de subclass_3 podem acessar c. Do ponto de vista de c, é como se a derivação fosse pública. Os dois pontos duplos (::) nessa definição de classe são um operador de resolução de escopo. Ele especifica a classe onde a entidade seguinte é definida.

O exemplo a seguir ilustra o propósito e o uso da derivação privada.

Considere o seguinte exemplo de herança em C++, no qual uma classe geral de lista encadeada é definida e então usada para definir duas subclasses úteis:

```
class single_linked_list {
  private:
    class node {
      public:
        node *link;
        int contents;
    };
    node *head;
  public:
    single_linked_list() {head = 0};
    void  insert_at_head(int);
    void  insert_at_tail(int);
    int   remove_at_head();
    int   empty();
};
```

A classe aninhada, node, define uma célula da lista encadeada que consiste em uma variável inteira e um ponteiro para um objeto node. A classe está na cláusula privada, que a oculta de todas as outras classes. Mas seus membros são públicos, então são visíveis para a classe aninhadora, single_linked_list. Se fossem privados, node pre-

cisaria declarar a classe aninhadora como amiga para torná-los visíveis nela. Note que classes aninhadas não têm acesso especial aos membros da classe aninhadora. Apenas membros de dados estáticos da classe aninhadora são visíveis para os métodos da classe aninhada.[10]

A classe aninhadora, `single_linked_list`, tem apenas um membro de dados, um ponteiro para agir como cabeçalho da lista. Ele contém uma função construtora, a qual configura `head` com o valor do ponteiro nulo. As quatro funções membro permitem inserir nós em qualquer uma das extremidades de um objeto lista e removê-los de uma das extremidades da lista; além disso, possibilitam que as listas sejam testadas para verificar se estão vazias.

As seguintes definições fornecem classes para pilha e fila, ambas baseadas na classe `single_linked_list`:

```
class stack : public single_linked_list {
  public:
    stack() {}
    void push(int value) {
      insert_at_head(value);
    }
    int pop() {
      return remove_at_head();
    }
};
class queue : public single_linked_list {
  public:
    queue() {}
    void enqueue(int value) {
      insert_at_tail(value);
    }
    int dequeue() {
      remove_at_head();
    }
};
```

Note que tanto objetos da subclasse `stack` quanto de `queue` podem acessar a função `empty` definida na classe base `single_linked_list` (porque ela é uma derivação pública). Ambas as subclasses definem funções construtoras que não fazem nada. Quando um objeto de uma subclasse é criado, o construtor correto da subclasse é chamado implicitamente. Então, qualquer construtor aplicável da classe base é chamado. Em nosso exemplo, quando um objeto do tipo `stack` é criado, o construtor em `single_linked_list` é chamado; ele faz a inicialização necessária. Depois, o construtor de `stack`, que não faz nada, é chamado.

As classes `stack` e `queue` enfrentam o mesmo problema: clientes de ambas podem acessar todos os membros públicos da classe pai, `single_linked_list`. Um cliente de um objeto `stack` poderia chamar `insert_at_tail`, destruindo a inte-

[10]Uma classe também pode ser definida em um método de uma classe aninhadora. As regras de escopo de tais classes são iguais às regras para classes aninhadas diretamente em outras classes, mesmo para variáveis locais declaradas no método no qual são definidas.

gridade da pilha. De modo similar, um cliente de um objeto `queue` poderia chamar `insert_at_head`. Esses acessos indesejados são permitidos porque tanto `stack` quanto `queue` são subtipos de `single_linked_list`. A derivação pública é usada quando é desejável que a subclasse herde a interface inteira da classe base. A alternativa é usar uma derivação na qual a subclasse herde apenas a implementação da classe base. Nos nossos dois exemplos, as classes derivadas podem ser escritas para não serem subtipos de sua classe pai, o que é feito por meio de derivação **privada** em vez de **pública**.[11] Então, ambas também precisarão reexportar `empty`, porque ela se tornará oculta para suas instâncias. Essa situação ilustra por que a derivação privada é utilizada. As novas definições dos tipos `stack` e `queue`, chamadas de `stack_2` e `queue_2`, são mostradas a seguir:

```
class stack_2 : private single_linked_list {
  public:
    stack_2() {}
    void push(int value) {
      single_linked_list :: insert_at_head(value);
    }
    int pop() {
      return single_linked_list :: remove_at_head();
    }
    single_linked_list:: empty();
};
class queue_2 : private single_linked_list {
  public:
    queue_2() {}
    void enqueue(int value) {
      single_linked_list :: insert_at_tail(value);
    }
    int dequeue() {
      single_linked_list :: remove_at_head();
    }
    single_linked_list:: empty();
};
```

Essas duas classes usam reexportação a fim de permitir o acesso aos métodos da classe base para os clientes. Isso não foi necessário quando foi utilizada derivação pública.

As duas versões de `stack` e `queue` ilustram a diferença entre subtipos e tipos derivados que não são subtipos. A lista encadeada é uma generalização tanto de pilhas quanto de filas, porque ambas podem ser implementadas como listas encadeadas. Então, é natural herdar de uma classe de lista encadeada para definir classes que representam pilhas e filas. No entanto, nenhuma delas é um subtipo da classe de lista encadeada, pois ambas tornam privados os membros públicos da classe pai, o que os torna inacessíveis para os clientes.

Uma das razões pelas quais amigos são necessários é que algumas vezes um subprograma deve ser escrito de forma a poder acessar os membros de duas classes diferentes.

[11] Elas não seriam subtipos porque os membros públicos da classe pai podem ser vistos em um cliente, mas não em um cliente da subclasse, cujos membros são privados.

Por exemplo, suponha que um programa use uma classe para vetores e outra para matrizes, e que um subprograma precise multiplicar um objeto matriz por um objeto vetor. Em C++, a função de multiplicação pode ser tornada amiga de ambas as classes.

Já mencionamos que C++ fornece herança múltipla. Como exemplo, suponha que quiséssemos uma classe para desenhar que precisasse do comportamento de uma classe escrita para desenhar figuras, e que os métodos da nova classe precisassem ser executados em uma linha de execução separada. Poderíamos definir o seguinte:

```
class Thread { . . . };
class Drawing { . . . };
class DrawThread : public Thread, public Drawing { . . . };
```

A classe `DrawThread` herda todos os membros de `Thread` e de `Drawing`. Se tanto `Thread` quanto `Drawing` incluem membros com o mesmo nome, eles podem ser referenciados de maneira não ambígua em objetos da classe `DrawThread` por meio do operador de resolução de escopo (`::`). Esse exemplo de herança múltipla é mostrado na Figura 12.5.

Algumas questões sobre a implementação de C++ de herança múltipla são discutidas na Seção 12.5.

Métodos que sobrescrevem[12] em C++ devem ter exatamente o mesmo perfil de parâmetros do método sobrescrito. Se existir qualquer diferença nos perfis de parâmetros, o método na subclasse será considerado um novo método, não relacionado àquele com o mesmo nome na classe ancestral. O tipo de retorno do que sobrescreve deve ser o mesmo do sobrescrito, ou ser um tipo derivado publicamente do tipo de retorno do sobrescrito.

12.4.2.3 Vinculação dinâmica

Todas as funções membros que definimos até agora são estaticamente vinculadas; ou seja, uma chamada a uma delas é estaticamente vinculada a uma definição de função. Um objeto C++ pode ser manipulado por meio de uma variável de valor, em vez de por um ponteiro ou uma referência. (Tal objeto seria estático ou dinâmico da pilha.) Entretanto, no caso de o tipo do objeto ser conhecido e estático, a vinculação dinâmica não

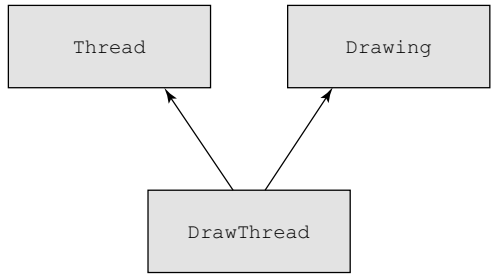


FIGURA 12.5
Herança múltipla.

[12]Lembre-se de que um método sobrescrevedor é definido na classe derivada para substituir um método virtual herdado de uma classe ancestral. As chamadas a um método sobrescrevedor devem ser vinculadas dinamicamente.

é necessária. Por outro lado, uma variável ponteiro que tenha o tipo de uma classe base pode ser usada para apontar para qualquer objeto dinâmico do monte de qualquer classe publicamente derivada a partir dessa classe base, tornando-a uma variável polimórfica. Subclasses publicamente derivadas são subtipos se nenhum dos membros da classe base é privado. Subclasses derivadas privadamente nunca são subtipos. Um ponteiro para uma classe base não pode ser usado para referenciar um método em uma subclasse que não é um subtipo.

C++ não permite que variáveis de valor (de maneira oposta a ponteiros ou a referências) sejam polimórficas. Quando uma variável polimórfica é usada para chamar uma função membro sobrescrita em uma das classes derivadas, a chamada deve ser dinamicamente vinculada à definição de função membro correta. Funções membros a serem dinamicamente vinculadas devem ser declaradas como funções virtuais, precedendo-se seus cabeçalhos com a palavra reservada **`virtual`**, que pode aparecer apenas no corpo de uma classe.

Considere a situação de ter uma classe base chamada `Shape`, com uma coleção de classes derivadas para diferentes tipos de formas, como círculos, retângulos e assim por diante. Se essas formas precisam ser mostradas, a função membro para isso, `draw`, deve ser única para cada descendente ou tipo de forma. Essas versões de `draw` devem ser definidas como virtuais. Quando uma chamada a `draw` é feita com um ponteiro para a classe base das classes derivadas, ela deve ser dinamicamente vinculada à função membro da classe derivada correta. O seguinte exemplo tem as definições esqueleto para a situação descrita:

```
class Shape {
  public:
    virtual void draw() = 0;
  . . .
};
class Circle : public Shape {
  public:
    void draw() { . . . }
  . . .
};
class Rectangle : public Shape {
  public:
    void draw() { . . . }
  . . .
};
class Square : public Rectangle {
  public:
    void draw() { . . . }
  . . .
};
```

Dadas essas definições, o código a seguir tem exemplos tanto de chamadas estaticamente vinculadas quanto de dinamicamente vinculadas.

```
Square* sq = new Square;
Rectangle* rect = new Rectangle;
Shape* ptr_shape;
```

```
ptr_shape = sq;          // Agora ptr_shape aponta para um
                         //  objeto Square
ptr_shape->draw();       // Dinamicamente vinculado a draw
                         //  na classe Square
rect->draw();            // Estaticamente vinculado a draw
                         //  na classe Rectangle
```

Essa situação é mostrada na Figura 12.6.

Note que a função `draw` na definição da classe base `Shape` é configurada como 0. Essa sintaxe peculiar é usada para indicar que essa função membro é uma **função virtual pura**, que não tem corpo e não pode ser chamada. Ela deve ser redefinida nas classes derivadas caso seja chamada. O propósito de uma função virtual pura é fornecer

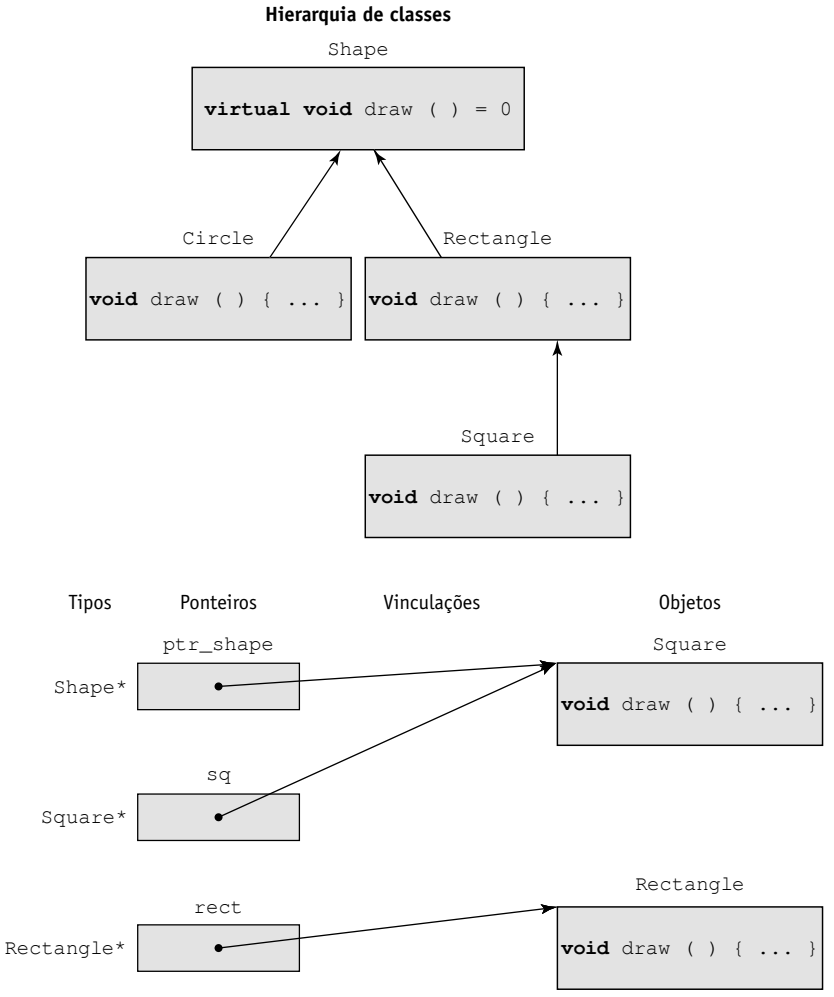


FIGURA 12.6
Vinculação dinâmica.

a interface de uma função sem fornecer sua implementação. Funções virtuais puras são normalmente definidas quando uma função membro real na classe base não seria útil. Lembre-se de que na Seção 12.2.3 foi discutida uma classe base `Building` e que cada subclasse descrevia algum tipo de construção específica. Cada subclasse tinha um método `draw`, mas nenhum deles seria útil na classe base. Assim, `draw` seria uma função virtual pura na classe `Building`.

Qualquer classe que inclua uma função virtual pura é uma **classe abstrata**. Em C++, uma classe abstrata não é marcada com uma palavra reservada. Ela pode incluir métodos completamente definidos. Devido à presença de uma ou mais funções virtuais, é inválido instanciar uma classe abstrata. Em um sentido estrito, uma classe abstrata é usada apenas para representar as características de um tipo. C++ fornece classes abstratas para modelar essas classes realmente abstratas. Se uma subclasse de uma classe abstrata não redefine uma função virtual pura de sua classe pai, essa função permanece como uma função virtual pura na subclasse, que também é uma classe abstrata.

Classes abstratas e herança suportam juntas uma técnica poderosa para desenvolvimento de software. Elas permitem aos tipos serem hierarquicamente definidos, de forma que os tipos relacionados possam ser subclasses de tipos verdadeiramente abstratos que definem suas características abstratas comuns.

A vinculação dinâmica permite aos códigos que usam membros como `draw` serem escritos antes que todas ou qualquer uma das versões de `draw` sejam escritas. Novas classes derivadas podem ser adicionadas anos depois, sem exigir mudança no código que usa tais membros vinculados dinamicamente. Esse é um recurso muito útil das linguagens orientadas a objetos.

Atribuições de referência para objetos dinâmicos da pilha são diferentes de atribuições de ponteiros para objetos dinâmicos do monte. Por exemplo, considere o código a seguir, que usa a mesma hierarquia de classes do último exemplo:

```
Square sq;         // Aloca um objeto Square na pilha
Rectangle rect;    // Aloca um objeto Rectangle
                   //  na pilha
rect = sq;         // Copia os valores dos membros de
                   //  dados do objeto Square
rect.draw();       // Chama draw a partir do objeto
                   //  Rectangle
```

Na atribuição `rect = sq`, o membro de dados do objeto referenciado por `sq` seria atribuído aos membros de dados do objeto referenciado por `rect`, mas `rect` ainda assim referenciaria o objeto `Rectangle`. Logo, a chamada a `draw` por meio do objeto referenciado por `rect` seria aquela da classe `Rectangle`. Se `rect` e `sq` fossem ponteiros para objetos dinâmicos do monte, a mesma atribuição seria de ponteiro, o que faria `rect` apontar para o objeto `Square` e uma chamada a `draw` por meio de `rect` ser vinculada dinamicamente a `draw` no objeto `Square`.

12.4.2.4 Avaliação

É natural comparar os recursos orientados a objetos de C++ com aqueles de Smalltalk. A herança de C++ é mais intrincada que a de Smalltalk em termos de controle de acesso. Ao usar tanto os controles de acesso dentro da definição de classe quanto os de

derivação, e também a possibilidade de funções e classes amigas, o programador C++ tem um controle altamente detalhado sobre o acesso aos membros de classes. Embora C++ forneça herança múltipla e Smalltalk não, muitos profissionais acham que isso não é uma vantagem para C++. Os inconvenientes da herança múltipla superam em muito seu valor. De fato, C++ é a única linguagem discutida neste capítulo que suporta herança múltipla. Por outro lado, as linguagens que oferecem alternativas à herança múltipla, como Objective-C, Java e C#, claramente têm uma vantagem em relação à Smalltalk nessa área.

Em C++, o programador pode especificar se deve ser usada vinculação estática ou dinâmica. Como a vinculação estática é mais rápida, isso é uma vantagem para aquelas situações nas quais a vinculação dinâmica não é necessária. Além disso, a vinculação dinâmica em C++ é rápida se comparada com a de Smalltalk. Vincular uma chamada a uma função membro virtual em C++ com uma definição de função tem um custo fixo, independentemente do quão distante na árvore de herança a definição aparece. Chamadas a funções virtuais exigem apenas cinco referências de memória a mais que chamadas estaticamente vinculadas (Stroustrup, 1988). Em Smalltalk, entretanto, as mensagens são sempre vinculadas dinamicamente aos métodos e, quanto mais longe na hierarquia de herança o método correto está, mais tempo ela leva. A desvantagem de permitir ao usuário decidir quais vinculações são estáticas e quais são dinâmicas é que o projeto original deve incluir essas decisões, que talvez tenham de ser alteradas posteriormente.

A verificação de tipos estática de C++ é uma vantagem sobre Smalltalk, na qual toda a verificação de tipos é dinâmica. Um programa Smalltalk pode ser escrito com mensagens para métodos inexistentes, os quais não são descobertos até que o programa seja executado. Um compilador C++ encontra tais erros, e o reparo deles é mais barato que o daqueles localizados durante os testes.

Smalltalk é basicamente uma linguagem desprovida de tipos, ou seja, todo o código é efetivamente genérico. Isso fornece uma boa dose de flexibilidade, mas em detrimento da verificação estática de tipos. C++ fornece classes genéricas por meio de seu recurso de *templates* (conforme descrito no Capítulo 11), que retém os benefícios da verificação de tipos estática.

A principal vantagem de Smalltalk está na elegância e na simplicidade da linguagem, que resulta da filosofia única de seu projeto. Ela é pura e completamente dedicada ao paradigma orientado a objetos. Por outro lado, C++ é uma linguagem extensa e complexa, sem uma filosofia única como base, exceto o suporte para programação orientada a objetos e o fato de incluir a base de usuários de C. Um de seus objetivos mais significativos é preservar a eficiência e a qualidade de C, oferecendo ainda as vantagens da programação orientada a objetos. Algumas pessoas acham que os recursos dessa linguagem nem sempre se encaixam bem e que ao menos parte de sua complexidade é desnecessária.

De acordo com Chambers e Ungar (1991), Smalltalk executou determinado conjunto de pequenos *benchmarks* no estilo de C com apenas 10% da velocidade de C otimizada. Os programas em C++ exigem apenas um pouco mais de tempo que os programas em C equivalentes (Stroustrup, 1988). Dada a grande diferença, em termos de eficiência, entre Smalltalk e C++, não é difícil deduzir que o uso comercial de C++ é muito maior

que o de Smalltalk. Existem outros fatores nessa diferença, mas a eficiência é um forte argumento a favor de C++. Evidentemente, todas as linguagens compiladas que suportam programação orientada a objetos executam aproximadamente 10 vezes mais rápido que Smalltalk.

12.4.3 Objective-C

Discutimos o suporte para programação orientada a objetos em Objective-C em relação ao de C++. Essas duas linguagens foram projetadas praticamente ao mesmo tempo. Ambas adicionam suporte para programação orientada a objetos à linguagem C. Aparentemente, a maior diferença está na sintaxe de chamadas a métodos, as quais em C++ são intimamente relacionadas às chamadas de função de C, enquanto em Objective-C são mais parecidas com as chamadas de método de Smalltalk.

12.4.3.1 Características gerais

Assim como C#, Objective-C tem tipos primitivos e objetos. Lembre-se (do Capítulo 11) de que uma definição de classe é composta de duas partes: interface e implementação. Frequentemente, essas duas partes são colocadas em arquivos separados; o arquivo de interface usa a extensão de nome `.h` e o de implementação usa a extensão de nome `.m`. Quando a interface está em um arquivo separado, o arquivo de implementação começa com o seguinte:

```
#import "interface_file.h"
```

As variáveis de instância são declaradas em um bloco delimitado por chaves, após o cabeçalho da seção de interface. Objective-C não suporta variáveis de classe diretamente. Contudo, uma variável global estática definida no arquivo de implementação pode ser usada como variável de classe.

A seção de implementação de uma classe contém definições dos métodos declarados na seção de interface correspondente.

Objective-C não permite aninhamento de classes.

12.4.3.2 Herança

Objective-C suporta apenas herança simples. Toda classe deve ter uma classe pai, exceto a classe raiz predefinida, chamada `NSObject`. Uma razão para se ter uma única classe raiz é que existem algumas operações universalmente necessárias. Entre elas estão os métodos de classe `alloc` e `init`. A classe pai de uma nova classe é declarada na diretiva interface, após os dois pontos anexados ao nome da classe definida, como no seguinte:

```
@interface myNewClass: NSObject {
```

Evidentemente, todos os membros de dados protegidos e públicos da classe pai são herdados pela subclasse. Novos métodos e variáveis de instância podem ser adicionados a ela. Lembre-se de que, embora o acesso aos métodos possa ser controlado por especificadores em C++, em Objective-C todos os métodos são públicos, e isso não pode ser

alterado. Um método definido na subclasse que tenha o mesmo nome, o mesmo tipo de retorno e o mesmo número e tipos de parâmetros sobrescreve o método herdado. O método sobrescrito pode ser chamado em outro método da subclasse por meio de **super**, uma referência ao objeto pai. Não há como impedir a sobrescrita de um método herdado. Como todos os membros públicos de uma classe base também são públicos nas subclasses derivadas dessa classe base, todas as subclasses podem ser subtipos.

Como em Smalltalk, em Objective-C qualquer nome de método pode ser chamado em qualquer objeto. Se o sistema de tempo de execução descobre que o objeto não tem tal método (com o protocolo correto), ocorre um erro.

Objective-C não suporta as derivações privadas e protegidas de C++.

Como nas outras linguagens que suportam programação orientada a objetos, o construtor de uma instância de uma subclasse sempre deve chamar o construtor da classe pai antes de qualquer coisa. Se o nome do construtor da classe pai é `init`, isso é feito com a seguinte sentença:

```
[super init];
```

Objective-C inclui duas maneiras de estender uma classe, além das subclasses: categorias e protocolos. Uma coleção de métodos pode ser adicionada a uma classe com uma construção chamada **categoria**. Uma categoria é uma interface secundária de uma classe que contém declarações de métodos. Nenhuma variável de instância nova pode ser incluída na interface secundária. A forma sintática de tal interface é exemplificada pelo seguinte:

```
#import "Stack.h"
@interface Stack (StackExtend)
  -(int) secondFromTop;
  -(void) full;
@end
```

O nome dessa categoria é `StackExtend`. A interface original é acessível porque é importada; portanto, a classe pai não precisa ser mencionada. Os métodos novos são misturados aos métodos da interface original. Consequentemente, às vezes as categorias são denominadas **mixins**. Ocasionalmente, os mixins são usados para adicionar certas funcionalidades a diferentes classes. E, é claro, a classe ainda tem uma superclasse normal a partir da qual herda membros. Então, os mixins fornecem alguns dos benefícios da herança múltipla, sem as colisões de nomes que podem ocorrer se os módulos não exigem nomes de módulos em suas funções. Evidentemente, uma categoria também deve ter uma seção de implementação, a qual contém o nome da categoria entre parênteses, após o nome da classe na diretiva `implementation`, como no seguinte:

```
@implementation Stack (StackExtend)
```

A implementação não precisa implementar todos os métodos da categoria.

Existe outra maneira de fornecer alguns dos benefícios da herança múltipla em Objective-C: os protocolos. Embora Objective-C não forneça classes abstratas, como em C++, os protocolos estão relacionados a elas. Um *protocolo* é uma lista de declarações de método. A sintaxe de um protocolo é exemplificada a seguir:

```
@protocol MatrixOps
  -(Matrix *) add: (Matrix *) mat;
  -(Matrix *) subtract: (Matrix *) mat;
@optional
  -(Matrix *) multiply: (Matrix *) mat;
@end
```

Nesse exemplo, `MatrixOps` é o nome do protocolo. Os métodos `add` e `subtract` devem ser implementados por uma classe que use o protocolo. Esse uso é chamado de *implementação* ou *adoção* do protocolo. A parte opcional especifica que o método `multiply` pode ou não ser implementado por uma classe adotante.

Uma classe que adota um protocolo lista o nome dele entre sinais de menor que e maior que após o nome da classe na diretiva interface, como segue:

```
@interface MyClass: NSObject <YourProtocol>
```

12.4.3.3 Vinculação dinâmica

Em Objective-C, o polimorfismo é implementado de modo diferente do utilizado na maioria das outras linguagens de programação comuns. Uma variável polimórfica é criada por meio da declaração de que ela é do tipo `id`. Tal variável pode referenciar qualquer objeto. O sistema de tempo de execução monitora a classe do objeto ao qual uma variável de tipo `id` se refere. Se uma chamada a um método é feita por meio de tal variável, a chamada é vinculada dinamicamente ao método correto, supondo que exista um.

Por exemplo, suponha que um programa tenha as classes `Circle` e `Square` definidas e que ambas têm métodos chamados `draw`. Considere o esqueleto de código a seguir:

```
// Cria os objetos
Circle *myCircle = [[Circle alloc] init];
Square *mySquare = [[Square alloc] init];

// Inicializa os objetos
[myCircle setCircumference: 5];
[mySquare setSide: 5];

// Cria a variável id
id shapeRef;

//Configura a variável id para referenciar o círculo e desenhá-lo
shapeRef = myCircle;
[shapeRef draw];
```

```
// Configura a variável id para referenciar o quadrado
shapeRef = mySquare;
[shapeRef draw];
```

Primeiramente, esse código desenha o círculo e depois o quadrado, com os dois métodos `draw` chamados por meio da referência ao objeto `shapeRef`.

12.4.3.4 Avaliação

O suporte para programação orientada a objetos em Objective-C é adequado, embora existam algumas pequenas deficiências. Não há como impedir a sobrescrita de um método herdado. O suporte para polimorfismo com seu tipo de dado `id` é excessivo, pois permite que variáveis referenciem qualquer objeto, em vez de apenas os que estão na linha de herança. Embora não exista nenhum suporte direto para herança múltipla, a linguagem inclui uma forma de mixin, as categorias, que oferecem parte dos recursos da herança múltipla sem todas as suas desvantagens. As categorias permitem que uma coleção de comportamentos seja adicionada a qualquer classe. Os protocolos fornecem os recursos das interfaces, como os de Java, os quais também fornecem alguns dos recursos da herança múltipla.

12.4.4 Java

Como o projeto de classes, herança e métodos em Java é similar ao de C++, nesta seção nos concentramos apenas nas áreas nas quais Java difere de C++.

12.4.4.1 Características gerais

Como C++, Java suporta tanto dados de objetos quanto de não objetos. Entretanto, em Java apenas valores dos tipos primitivos escalares (booleano, caracteres e tipos numéricos) não são objetos. As enumerações e matrizes em Java são objetos. A razão para Java ter não objetos é a eficiência.

Em Java 5.0+, os valores primitivos sofrem coerção implicitamente quando são colocados no contexto do objeto. Essa coerção converte o valor primitivo em um objeto da classe wrapper do tipo do valor primitivo. Por exemplo, colocar um valor ou variável **`int`** no contexto de objetos causa a coerção de um objeto `Integer` com o valor do primitivo **`int`**. Essa coerção é chamada de **encaixotamento** (*boxing*).

Embora as classes C++ possam ser definidas como sem ancestrais, isso não é possível em Java. Todas as classes Java devem ser subclasses da classe raiz, `Object`, ou de alguma descendente dela. Uma vantagem disso é que alguns métodos comumente necessários, como `toString` e `equals`, podem ser definidos em `Object` e herdados e utilizados por todas as outras classes.

Todos os objetos em Java são dinâmicos do monte explícitos. A maioria é alocada com o operador **`new`**, mas não existe um operador de liberação explícito. A coleta de lixo é usada para recuperação de armazenamento. Como muitos outros recursos de linguagem, apesar de a coleta de lixo evitar alguns problemas sérios, como ponteiros soltos, ela pode causar outros. Uma dessas dificuldades surge porque o coletor de lixo libera, ou

recupera, o armazenamento ocupado por um objeto, mas não faz nada além disso. Por exemplo, se um objeto tem acesso a algum recurso além da memória do monte, como um arquivo ou um bloqueio em um recurso compartilhado, o coletor de lixo não o recupera. Para tais situações, Java permite a inclusão de um método especial, **`finalize`**, relacionado a uma função destrutora em C++.

Um método **`finalize`** é implicitamente chamado quando o coletor de lixo está prestes a recuperar o armazenamento ocupado pelo objeto. O problema de **`finalize`** é que o momento no qual ele (e o coletor de lixo) será executado não pode ser forçado nem previsto. A alternativa ao uso de **`finalize`** para recuperar recursos utilizados por um objeto que será coletado como lixo é incluir um método que faça a recuperação. O único problema disso é que todos os clientes dos objetos devem estar cientes desse método e precisam lembrar de chamá-lo.

12.4.4.2 Herança

Em Java, um método pode ser definido como **`final`**, ou seja, não pode ser sobrescrito em nenhuma classe descendente. Quando a palavra reservada **`final`** é especificada em uma definição de classe, esta não pode ser uma subclasse. Todos os métodos de uma classe final são implicitamente final, o que significa que as vinculações de chamadas de método aos métodos da classe são feitas estaticamente.

A vantagem de definir uma classe como final é que nenhuma alteração é permitida nela. Por exemplo, `String` é uma classe final e, por isso, qualquer método que receba uma referência `String` em um parâmetro pode depender da estabilidade do significado dos métodos de `String`. A desvantagem é que definir uma classe como final impede reúsos que exigem modificações, mesmo que pequenas.

Java inclui a anotação `@Override`, a qual diz ao compilador que faça uma verificação a fim de determinar se o método seguinte sobrescreve um método em uma classe ancestral. Se não sobrescrever, o compilador emitirá uma mensagem de erro.

Como C++, Java requer que o construtor da classe pai seja chamado antes do construtor da subclasse. Se parâmetros serão passados para o construtor da classe pai, ele deve ser explicitamente chamado, como no seguinte exemplo:

```
super(100, true);
```

Se não existe uma chamada explícita ao construtor da classe pai, o compilador insere uma chamada para o construtor com zero parâmetros na classe pai.

Java não suporta as derivações privadas de C++. É possível supor que os projetistas de Java acreditavam que subclasses devem ser subtipos, o que não acontece quando são suportadas derivações privadas. Assim, eles não as incluíram. Portanto, as subclasses de Java podem ser subtipos.

As primeiras versões de Java incluíam uma coleção, `Vector`, que continha uma longa lista de métodos para manipular dados em uma construção de coleção. Essas versões de Java incluíam também uma subclasse de `Vector`, `Stack`, a qual adicionava métodos para as operações push e pop. Infelizmente, como Java não tem derivação privada, todos os métodos de `Vector` também eram visíveis na classe `Stack`, o que tornava os objetos `Stack` capazes de uma variedade de operações que podiam invalidá-los.

Java suporta diretamente apenas herança simples. Entretanto, inclui um tipo de classe abstrata, chamada de **interface**, que fornece suporte parcial para herança múltipla.[13] Uma definição de interface é similar a uma definição de classe, exceto que pode conter apenas constantes nomeadas e declarações de métodos (não definições). Ela não pode conter construtores, métodos não abstratos nem declarações de variável. Então, uma interface é precisamente o que seu nome indica – ela define apenas a especificação de uma classe. (Lembre-se de que uma classe abstrata em C++ pode ter variáveis de instância e todos os métodos podem ser completamente definidos, exceto um.) Uma classe não herda de uma interface; ela a implementa. Na verdade, uma classe pode implementar qualquer número de interfaces. A classe deve implementar todos os métodos cujas especificações (mas não os corpos) aparecem na definição de uma interface a fim de implementá-la.

Uma interface pode ser usada para simular herança múltipla. Uma classe pode ser derivada de outra e implementar uma interface, com esta tomando o lugar de uma segunda classe pai. Algumas vezes isso é chamada de herança mista, já que as constantes e métodos da interface são misturados com os métodos e dados herdados da superclasse, assim como quaisquer novos dados e/ou métodos definidos na subclasse.

Uma das capacidades mais interessantes das interfaces é o fato de fornecerem outro tipo de polimorfismo. Isso ocorre porque as interfaces podem ser tratadas como tipos. Por exemplo, um método pode especificar um parâmetro formal que é uma interface. Tal parâmetro formal pode aceitar um parâmetro real de qualquer classe que implemente a interface, tornando o método polimórfico.

Uma variável que não é um parâmetro também pode ser declarada como do tipo de uma interface. Tal variável pode referenciar qualquer objeto de qualquer classe que implemente a interface.

Um dos problemas da herança múltipla ocorre quando uma classe é derivada de duas classes pais e ambas definem um método público com o mesmo nome e protocolo. Esse problema é evitado com as interfaces. Embora uma classe que implemente uma interface deva fornecer definições para todos os métodos especificados na interface, se a classe e a interface incluem métodos com o mesmo nome e protocolo, ela não precisa reimplementar esses métodos. Os conflitos de nomes de método que podem ocorrer com herança múltipla não ocorrem com herança simples e interfaces. Além disso, os conflitos de nomes de variável são completamente evitados, pois as interfaces não podem definir variáveis.

Uma interface não substitui a herança múltipla, porque nesta existe reuso de código, enquanto aquela não fornece essa opção. Existe uma diferença significativa, já que o reuso de código é um dos principais benefícios da herança. Java fornece um modo de evitar parcialmente essa deficiência. Uma das interfaces implementadas poderia ser substituída por uma classe abstrata, a qual poderia incluir código que pudesse ser herdado, oferecendo assim algum reúso de código.

Um problema que resulta do fato de as interfaces substituírem a herança múltipla é o seguinte: se uma classe tenta implementar duas interfaces e ambas definem métodos que têm o mesmo nome e protocolo, não há como implementar ambos na classe.

[13] Uma interface Java é semelhante a um protocolo em Objective-C.

Como exemplo de uma interface, considere o método `sort` da classe padrão Java, `Array`. Qualquer classe que use esse método deve fornecer uma implementação de um método para comparar os elementos a serem ordenados. A interface genérica `Comparable` fornece o protocolo para esse método comparador, chamado de `compareTo`. O código para a interface `Comparable` é:

```
public interface  Comparable <T> {
    public int  compareTo(T b);
}
```

O método `compareTo` deve retornar um inteiro negativo se o objeto por meio do qual ele foi chamado for menor que o objeto passado como parâmetro, zero se forem iguais e um inteiro positivo se o parâmetro for menor que o objeto por meio do qual `compareTo` foi chamado. Uma classe que implementa a interface `Comparable` pode ordenar o conteúdo de qualquer matriz de objetos do tipo genérico, desde que o método `compareTo` implementado para o tipo genérico esteja implementado e forneça o valor apropriado.

Além das interfaces, Java também suporta classes abstratas, semelhantes às de C++. Os métodos abstratos de uma classe abstrata Java são representados apenas como o cabeçalho do método, o qual inclui a palavra reservada **`abstract`**. A classe abstrata também é marcada com **`abstract`**. Evidentemente, classes abstratas não podem ser instanciadas.

O Capítulo 14 ilustra o uso de interfaces na manipulação de eventos em Java.

12.4.4.3 Vinculação dinâmica

Em C++, um método deve ser definido como virtual para permitir vinculação dinâmica. Em Java, todas as chamadas a métodos são dinamicamente vinculadas, a menos que o método chamado tenha sido declarado como **`final`**, de modo que não pode ser sobrescrito e todas as vinculações são estáticas. A vinculação estática também é usada se o método for **`static`** ou **`private`**, no qual ambos os modificadores não permitem sobrescrita.

12.4.4.4 Classes aninhadas

Java tem diversas variedades de classes aninhadas, todas com a vantagem de serem ocultas de todas as classes em seus pacotes, exceto a aninhadora. As classes não estáticas diretamente aninhadas em outra classe são chamadas **classes internas**. Cada instância de uma classe interna deve ter um ponteiro implícito para a instância de sua classe aninhadora à qual pertence. Isso fornece aos métodos da classe aninhada acesso a todos os membros da classe aninhadora, incluindo os privados. Classes aninhadas estáticas não têm esse ponteiro; então, não podem acessar membros da classe aninhadora. Logo, classes aninhadas estáticas em Java são como as aninhadas de C++.

Apesar de isso parecer estranho em uma linguagem de escopo estático, os membros da classe interna, mesmo os privados, são acessíveis na classe externa. Tais referências

devem incluir a variável que referencia o objeto da classe interna. Por exemplo, suponha que a classe externa cria uma instância da classe interna com a seguinte sentença:

```
myInner = this.new Inner();
```

Então, se a classe interna define uma variável chamada `sum`, ela pode ser referenciada na classe externa como `myInner.sum`.

Uma instância de uma classe aninhada pode existir apenas dentro de uma instância de sua aninhadora. Classes aninhadas também podem ser anônimas. Estas têm sintaxe complexa, mas são uma forma abreviada de definir uma classe usada a partir de apenas um lugar. Um exemplo de uma classe aninhada anônima aparece no Capítulo 14.

Uma **classe aninhada local** é definida em um método de sua aninhadora. Classes aninhadas locais nunca são definidas com um especificador de acesso (**`private`** ou **`public`**). Seu escopo é sempre limitado à sua aninhadora. Um método em uma classe aninhada local pode acessar as variáveis definidas em sua classe aninhadora e as variáveis com modificador **`final`** definidas no método no qual a aninhada local é definida. Os membros de uma classe aninhada local são visíveis apenas no método no qual ela é definida.

12.4.4.5 Avaliação

O projeto de Java para suporte à programação orientada a objetos é similar ao de C++, mas emprega uma aderência mais consistente aos princípios de orientação a objetos. Java não permite classes sem pais e usa vinculação dinâmica como a maneira "normal" de vincular chamadas a métodos às definições de métodos. Isso, é claro, aumenta um pouco o tempo de execução em relação às linguagens nas quais muitas vinculações de método são estáticas. Contudo, quando essa decisão de projeto foi tomada, a maioria dos programas Java era interpretada, de modo que o tempo de interpretação tornava o tempo de vinculação extra insignificante. O controle de acesso para os conteúdos de uma definição de classe é de certa forma simples, quando comparado com a miríade de controles de acesso de C++, que variam desde controles de derivação até funções amigas. Por fim, Java usa interfaces para oferecer uma forma de suporte para herança múltipla, a qual não tem todos os inconvenientes da herança múltipla real.

12.4.5 C#

O suporte de C# para programação orientada a objetos é similar ao de Java.

12.4.5.1 Características gerais

C# inclui tanto classes quanto estruturas; as classes são bastante similares às de Java, e as estruturas funcionam como construções menos poderosas. Uma diferença importante é que as estruturas são tipos de valor; isto é, são dinâmicas da pilha. Isso poderia causar o problema do fatiamento de objetos, mas ele é impedido porque estruturas não podem ser subclasses. Mais detalhes sobre como as estruturas C# diferem de suas classes estão no Capítulo 11.

12.4.5.2 Herança

C# usa a sintaxe de C++ para definir classes. Por exemplo,

```
public class  NewClass : ParentClass { . . . }
```

Um método herdado da classe pai pode ser substituído na classe derivada marcando-se sua definição na subclasse com **new**. O método **new** oculta o método com o mesmo nome na classe pai para acesso normal. Entretanto, a versão da classe pai ainda pode ser chamada se a chamada for prefixada com **base**. Por exemplo,

```
base.Draw();
```

Assim como em Java, as subclasses podem ser subtipos. O suporte de C# para interfaces é o mesmo de Java. Ela não suporta herança múltipla.

12.4.5.3 Vinculação dinâmica

Para permitir vinculação dinâmica de chamadas a métodos em C#, tanto o método base quanto seus métodos correspondentes nas classes derivadas devem ser especialmente marcados. O método da classe base deve ser marcado com **virtual**, como em C++. Para tornar clara a intenção de um método de uma subclasse, o qual tem o mesmo nome e protocolo de um método virtual de uma classe ancestral, C# exige que tais métodos sejam marcados com **override** se forem sobrescrever o método virtual da classe pai.[14] Por exemplo, a versão C# da classe C++ `Shape` que aparece na Seção 12.4.2.3 é:

```
public class Shape {
  public virtual void Draw() { . . . }
  . . .
}
public class Circle : Shape {
  public override void Draw() { . . . }
  . . .
}
 public class  Rectangle : Shape {
  public override void Draw() { . . . }
  . . .
}
public class Square : Rectangle {
  public override void Draw() { . . . }
  . . .
}
```

C# inclui métodos abstratos similares aos de C++, exceto aqueles especificados com sintaxe diferente. Por exemplo, o seguinte método é abstrato em C#:

```
abstract public void Draw();
```

[14] Lembre-se de que em Java isso pode ser especificado com a anotação `@Override`.

Uma classe que inclui pelo menos um método abstrato é uma classe abstrata, e toda classe abstrata deve ser marcada com **`abstract`**. Classes abstratas não podem ser instanciadas. Portanto, qualquer subclasse de uma classe abstrata que vai ser instanciada deve implementar todos os métodos abstratos que herda.

Como em Java, todas as classes C# são, em última análise, derivadas de uma classe raiz única, `Object`. A classe `Object` define uma coleção de métodos, incluindo `ToString`, `Finalize` e `Equals`, herdados por todos os tipos C#.

12.4.5.4 Classes aninhadas

Uma classe C# diretamente aninhada em uma classe se comporta como uma classe aninhada estática em Java (como uma classe aninhada em C++). Como C++, C# não suporta classes aninhadas que se comportam como as aninhadas não estáticas de Java.

12.4.5.5 Avaliação

Como C# é uma linguagem orientada a objetos baseada em C projetada recentemente, deve-se esperar que seus projetistas tenham aprendido com seus antecessores, reproduzindo o que foi bem-sucedido no passado e remediando alguns dos problemas. Um resultado disso, relacionado aos poucos problemas de Java, é que as diferenças entre o suporte de C# e o de Java para programação orientada a objetos são relativamente pequenas. A disponibilidade de estruturas em C#, que Java não tem, pode ser considerada um aprimoramento. Assim como o de Java, o suporte de C# para programação orientada a objetos é mais simples que o de C++, o qual muitos consideram um aprimoramento.

12.4.6 Ruby

Conforme mencionado, Ruby é uma linguagem de programação orientada a objetos no sentido de Smalltalk. Praticamente tudo na linguagem é um objeto, e toda a computação é feita por passagem de mensagens. Apesar de os programas terem expressões que usam operadores comuns (usando a forma infixa) e que possuem a mesma aparência das expressões em linguagens como Java, essas expressões são, na verdade, avaliadas por meio de passagem de mensagens. Como acontece com Smalltalk, quando alguém escreve `a + b`, a expressão é avaliada como o envio da mensagem `+` para o objeto referenciado por `a`, passando uma referência ao objeto `b` como parâmetro. Em outras palavras, `a + b` é implementado como `a.+ b`.

12.4.6.1 Características gerais

As definições de classe em Ruby diferem daquelas de linguagens como C++ e Java no sentido de que são executáveis. Por isso, é permitido que permaneçam abertas durante a execução. Um programa pode adicionar membros a uma classe qualquer número de vezes, simplesmente por meio de definições secundárias que incluam os novos membros. Durante a execução, a definição atual da classe é a união de todas as definições executadas. Definições de métodos também são executáveis, o que permite a um programa escolher entre duas versões de uma definição de método durante a execução,

colocando-se as duas definições nas cláusulas senão (else) e então (then) de uma construção de seleção.

Os objetos Ruby são criados com **new**, que chama implicitamente um construtor. O construtor usual em uma classe Ruby é chamado de `initialize`. Um construtor em uma subclasse pode inicializar os membros de dados da classe pai que têm métodos de escrita definidos. Isso é feito chamando-se **super** com os valores iniciais como parâmetro reais. **super** chama o método da classe pai que tem o mesmo nome do método no qual a chamada a **super** aparece.

As classes Ruby podem ser aninhadas, mas a classe aninhada não tem nenhum acesso especial às variáveis ou aos métodos da classe aninhadora.

Todas as variáveis em Ruby são referências a objetos e são todas sem tipos. Lembre-se de que os nomes de todas as variáveis de instância em Ruby começam com um sinal de arroba (`@`).

Em um claro distanciamento em relação às outras linguagens de programação comuns, o controle de acesso em Ruby é diferente para dados e para métodos. Todas as variáveis de instância têm acesso privado por padrão e isso não pode ser modificado. Portanto, em Ruby nenhuma subclasse é um subtipo. Se o acesso externo a uma variável de instância é exigido, métodos de acesso devem ser definidos. Por exemplo, considere o esqueleto de definição de classe:

```
class MyClass

# Um construtor
  def initialize
    @one = 1
    @two = 2
  end

# Um método de leitura para @one
  def one
    @one
  end

# Um método de escrita para @one
  def one=(my_one)
    @one = my_one
  end

end  # da classe MyClass
```

O sinal de igualdade (=) anexado ao nome do método de escrita significa que sua variável é atribuível. Então, todos os métodos de escrita têm sinais de igualdade anexados aos seus nomes. O corpo do método de escrita `one` ilustra o projeto de métodos de Ruby retornando o valor da última expressão avaliada onde não existe uma sentença de retorno. Nesse caso, o valor de `@one` é retornado.

Como os métodos de leitura e de escrita são tão necessários, Ruby fornece atalhos para criá-los. Se é desejável que uma classe tenha métodos de leitura para as duas

variáveis de instância, `@one` e `@two`, esses métodos podem ser especificados com uma sentença na classe:

```
attr_reader :one, :two
```

`attr_reader` é uma chamada à função, usando `:one` e `:two` como parâmetros reais. Preceder uma variável com dois pontos (`:`) faz com que o nome da variável seja usado, e não desreferenciado para o objeto ao qual ela se refere. Em vez de um valor ou um endereço, é passado o texto do nome da variável. É exatamente assim que os parâmetros de macro são passados.

A função que cria métodos de escrita de maneira similar é chamada de `attr_writer`. Ela tem o mesmo perfil de parâmetros de `attr_reader`.

As funções para criar métodos de leitura e de escrita são assim chamadas porque fornecem o protocolo para objetos da classe, os quais são chamados de **atributos**. Então, os atributos de uma classe definem a interface de dados (os dados tornados públicos por meio de métodos de acesso) para os objetos da classe.

Variáveis de classe, especificadas precedendo-se seus nomes por dois sinais de arroba (`@@`), são privadas para a classe e para suas instâncias. Essa privacidade não pode ser modificada. Além disso, diferentemente das variáveis globais e de instância, as de classe devem ser inicializadas antes de serem usadas.

12.4.6.2 Herança

As subclasses são definidas em Ruby com o símbolo de menor que (<), em vez dos dois pontos de C++. Por exemplo,

```
class MySubClass < BaseClass
```

Uma peculiaridade dos controles de acesso a métodos de Ruby é que eles podem ser modificados em uma subclasse simplesmente por meio de chamadas às funções de controle de acesso. Isso significa que duas subclasses podem acessar um método definido em uma classe base, mas objetos das outras subclasses não podem. Isso também permite que alguém modifique o acesso de um método publicamente acessível na classe base para um método privadamente acessível na subclasse.

12.4.6.3 Vinculação dinâmica

O suporte para vinculação dinâmica em Ruby é o mesmo de Smalltalk. As variáveis não são tipadas; em vez disso, são todas referências a objetos de qualquer classe. Então, todas as variáveis são polimórficas e todas as vinculações de chamadas a métodos aos métodos propriamente ditos são dinâmicas.

12.4.6.4 Avaliação

Como Ruby é uma linguagem de programação orientada a objetos no sentido mais puro, obviamente seu suporte para programação orientada a objetos é adequado. Entretanto, o controle de acesso aos membros de classe é mais fraco que o de C++. Ruby não suporta classes abstratas nem interfaces, apesar de seus mixins serem estreitamente relacionados às interfaces. Por fim, em grande parte pelo fato de Ruby ser interpretada, a eficiência de sua execução é bem pior que a das linguagens compiladas.

TABELA 12.1 Projetos

Questão de projeto/Linguagem	Smalltalk	C++	Objective-C	Java	C#	Ruby
Exclusividade de objetos	Todos os dados são objetos	Tipos primitivos, mais objetos	Tipos primitivos, mais objetos	Tipos primitivos, mais objetos	Tipos primitivos, mais objetos	Todos os dados são objetos
As subclasses são subtipos?	Podem ser e normalmente são	Podem ser e normalmente são, se a derivação for pública	Podem ser e normalmente são	Podem ser e normalmente são	Podem ser e normalmente são	Nenhuma subclasse é subtipo
Herança simples e múltipla	Apenas simples	Ambas	Apenas simples, mas alguns efeitos com protocolos	Apenas simples, mas alguns efeitos com interfaces	Apenas simples, mas alguns efeitos com interfaces	Apenas simples, mas alguns efeitos com módulos
Alocação e liberação de objetos	Todos os objetos são alocados no monte; a alocação é explícita e a liberação é implícita	Os objetos podem ser estáticos, dinâmicos da pilha ou dinâmicos do monte; a alocação e a liberação são explícitas	Todos os objetos são dinâmicos do monte; a alocação é explícita e a liberação é implícita	Todos os objetos são dinâmicos do monte; a alocação é explícita e a liberação é implícita	Todos os objetos são dinâmicos do monte; a alocação é explícita e a liberação é implícita	Todos os objetos são dinâmicos do monte; a alocação é explícita e a liberação é implícita
Vinculação dinâmica e estática	Todas as vinculações de método são dinâmicas	A vinculação de método pode ser uma ou outra	A vinculação de método pode ser uma ou outra	A vinculação de método pode ser uma ou outra	A vinculação de método pode ser uma ou outra	Todas as vinculações de método são dinâmicas
Classes aninhadas?	Não	Sim	Não	Sim	Sim	Sim
Inicialização	Construtores devem ser chamados explicitamente	Construtores são chamados implicitamente	Construtores devem ser chamados explicitamente	Construtores são chamados implicitamente	Construtores são chamados implicitamente	Construtores são chamados implicitamente

A Tabela 12.1 resume como os projetistas das linguagens listada nesta seção optaram por lidar com as questões de projeto descritas na Seção 12.3.

12.5 IMPLEMENTAÇÃO DE CONSTRUÇÕES ORIENTADAS A OBJETOS

Existem ao menos dois aspectos do suporte de linguagem para programação orientada a objetos que geram questões interessantes para os implementadores de linguagens: as estruturas de armazenamento para variáveis de instância e as vinculações dinâmicas de mensagens a métodos. Nesta seção, as examinamos brevemente.

12.5.1 Armazenamento de dados de instâncias

Em C++, as classes são definidas como extensões de estruturas de registro de C – structs. Essa similaridade sugere uma estrutura de armazenamento para as variáveis de instância de instâncias de classes – a mesma de um registro. Essa forma de estrutura é chamada de **registro de instância de classe (RIC)**. A estrutura de um RIC é estática; então, ele é construído em tempo de compilação e usado como um gabarito para a criação dos dados das instâncias de classe. Cada classe tem seu próprio RIC. Quando uma derivação ocorre, o RIC da subclasse é uma cópia do da superclasse, com entradas para as novas variáveis de instância adicionadas no final.

Como a estrutura do RIC é estática, o acesso a todas as variáveis de instância pode ser feito como nos registros, por meio de deslocamentos constantes a partir do início da instância do RIC. Isso torna esses acessos tão eficientes como os dos campos dos registros.

12.5.2 Vinculação dinâmica de chamadas a métodos

Métodos em uma classe estaticamente vinculados não precisam se envolver no RIC para a classe. Entretanto, métodos que serão vinculados dinamicamente devem ter entradas nessa estrutura. Tais entradas podem simplesmente ter um ponteiro para o código do método, configurado no momento da criação do objeto. Chamadas a um método podem então ser conectadas ao código correspondente por meio desse ponteiro no RIC. A desvantagem dessa técnica é que cada instância precisa armazenar ponteiros para todos os métodos dinamicamente vinculados que poderiam ser chamados a partir dela.

Note que a lista de métodos dinamicamente vinculados que podem ser chamados a partir de uma instância de uma classe é a mesma para todas as instâncias dela. Portanto, a lista de tais métodos precisa ser armazenada apenas uma vez. O RIC para uma instância precisa apenas de um ponteiro para essa lista, de forma a permitir que ele encontre os métodos chamados. A estrutura de armazenamento para a lista é geralmente chamada de **tabela virtual de métodos (vtable)**. As chamadas a métodos podem ser representadas como deslocamentos a partir do início da vtable. Variáveis polimórficas de uma classe ancestral sempre se referenciam ao RIC do objeto do tipo correto; então, a obtenção da versão correta de um método dinamicamente vinculado é garantida.

Considere o seguinte exemplo em Java, no qual todos os métodos são dinamicamente vinculados:

```
public class A {
  public int a, b;
  public void draw() { . . . }
  public int area() { . . . }
}
public class B extends  A {
  public int c, d;
  public void draw() { . . . }
  public void sift() { . . . }
}
```

Os RICs para as classes `A` e `B`, com suas vtables, são mostrados na Figura 12.7. Observe que o ponteiro para o método `area` na vtable de `B` aponta para o código do método area de `A`. Isso ocorre porque `B` não sobrescreve o método `area` de `A`; então, se um cliente de `B` chama `area`, ele chama o método `area` herdado de `A`. Por outro lado, os ponteiros para `draw` e `sift` na vtable de `B` apontam para `draw` e `sift` de `B`. O método `draw` é sobrescrito em `B` e `sift` é definido como uma adição em `B`.

A herança múltipla complica a implementação da vinculação dinâmica. Considere três definições de classe em C++:

```
class A {
  public:
     int a;
```

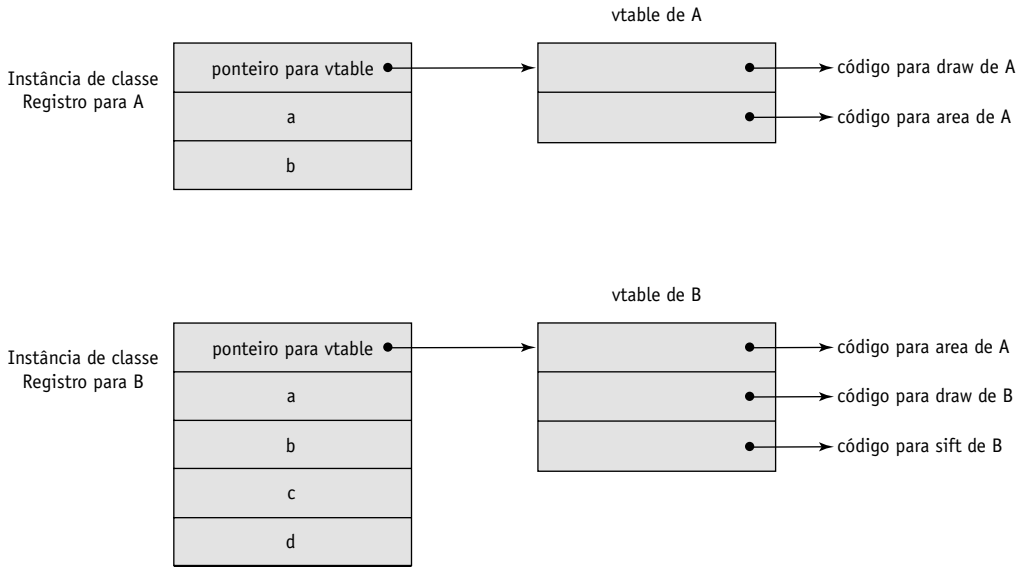


FIGURA 12.7
Um exemplo dos RICs com herança simples.

```
    virtual void fun() { . . . }
    virtual void init() { . . . }
};
class  B {
  public:
    int b;
    virtual void sum() { . . . }
};
class C : public A, public  B {
  public:
    int  c;
    virtual void fun() { . . . }
    virtual void dud() { . . . }
};
```

A classe `C` herda a variável `a` e o método `init` da classe `A`. Ela redefine o método `fun`, apesar de tanto seu `fun` quanto o de sua classe pai `A` serem potencialmente visíveis por meio de uma variável polimórfica (do tipo `A`). De `B`, `C` herda a variável `b` e o método `sum`. `C` define sua própria variável, `c`, e define um método não herdado, `dud`. Um RIC para C deve incluir os dados de `A`, de `B` e de `C`, assim como alguma forma de acessar todos os métodos visíveis. Sob herança simples, o RIC incluiria um ponteiro para uma vtable que tivesse o endereço do código de todos os métodos visíveis. Com herança múltipla, entretanto, isso não é tão simples. Devem existir ao menos duas visões diferentes disponíveis no RIC – uma para cada classe pai, uma das quais inclui a visão para a subclasse, `C`. Essa inclusão da visão da subclasse na visão da classe pai é a mesma que apareceria na implementação de herança simples.

Devem existir também duas vtables: uma para `A` e para a visão de `C` e uma para a visão de `B`. A primeira parte do RIC de `C` nesse caso pode ser o `C` e a visão de `A`, a qual inicia com um ponteiro na vtable para os métodos de `C` e para aqueles herdados de `A` e inclui os dados herdados de `A`. Seguindo isso, no RIC de `C` está a parte da visão de `B` que inicia com um ponteiro na vtable para os métodos virtuais de `B`, seguido pelos dados herdados de `B` e pelos dados definidos em `C`. O RIC para `C` é mostrado na Figura 12.8.

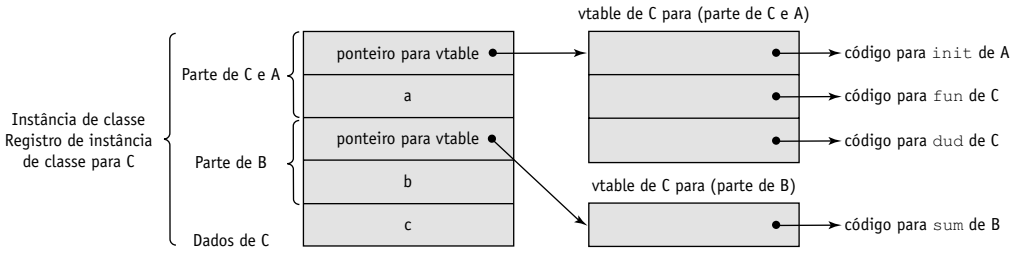


FIGURA 12.8
Um exemplo de RIC de uma subclasse com múltiplos pais.

12.6 REFLEXÃO

Uma discussão sobre reflexão não se encaixa perfeitamente em um capítulo sobre orientação a objetos, mas também não se enquadra em qualquer outro capítulo deste livro.

12.6.1 Introdução

Em geral, quanto mais tarde as vinculações ocorrem em uma linguagem de programação, mais flexível ela é. Por exemplo, a vinculação tardia de tipos de dados em linguagens de *scripting* e em linguagens funcionais permite que seus programas sejam mais genéricos que os das linguagens de tipo estático. Do mesmo modo, a vinculação dinâmica de chamadas de método a métodos que faz parte das linguagens orientadas a objetos permite que seus programas sejam mais fáceis de manter e estender. Entre outras coisas, a reflexão oferece a possibilidade de vinculação tardia de chamadas a métodos que estão fora da hierarquia de herança do código chamador.

12.6.2 O que é reflexão?

Uma linguagem de programação que suporta reflexão permite que seus programas tenham acesso em tempo de execução aos seus tipos e à sua estrutura e sejam capazes de modificar seu comportamento dinamicamente. Para que um programa examine seus tipos e sua estrutura, essas informações devem ser coletadas pelo compilador ou interpretador e disponibilizadas para o programa. Assim como as informações sobre a estrutura de um banco de dados, os tipos e a estrutura de um programa são chamados *metadados*. O processo pelo qual um programa examina seus metadados é denominado *introspecção*. Um programa pode modificar seu comportamento dinamicamente de várias formas diferentes: ele pode alterar seus metadados diretamente, usar os metadados ou interceder na sua execução. A primeira alternativa é complicada; a segunda é menos complexa e é comum entre as linguagens; a terceira é frequentemente chamada de *intercessão*.

Alguns dos principais usos da reflexão estão na construção de ferramentas de software. Um navegador de classe precisa enumerar as classes de um programa. Os ambientes de desenvolvimento integrado visuais podem usar informações de tipo para ajudar um desenvolvedor na construção de código de tipo correto. Depuradores devem ser capazes de examinar campos e métodos privados de classes. Sistemas de teste precisam descobrir todos os métodos de uma classe para garantir que os dados do teste experimentem todos eles.

Para ilustrar um uso comum e relativamente simples de reflexão, apresentamos o seguinte problema. Um jardim zoológico tem uma área grande dedicada às aves. O viveiro de cada espécie contém uma placa que fornece informações gerais sobre os habitantes. Na placa há uma pequena tela na qual o visitante que tenha especial interesse na espécie pode digitar o número de seu tíquete de entrada. Na saída da exposição de aves, o visitante pode digitar o número de seu tíquete novamente em uma pequena tela, o que faz um computador imprimir imagens das aves nas quais ele se interessou. O sistema de computador que suporta essas atividades tem um objeto que inclui um método que desenha a imagem de cada uma das aves na tela. Isso parece bem simples. Quando um visitante seleciona uma ave do viveiro, o sistema coloca uma referência para um objeto associado a essa ave em uma lista. Na saída da exposição, o sistema chama o método de

desenho de cada objeto da lista do visitante. O processo é complicado pelo fato de que o jardim zoológico fornece apenas alguns dos objetos ave. Alguns deles são adquiridos de outros fornecedores e alguns são doados por benfeitores. Devido às várias fontes de objetos ave, eles não têm uma classe base comum (a não ser Object) e não implementam uma interface comum; então, quais referências de tipo podem ser salvas? Uma solução óbvia é tornar cada objeto ave a subclasse de uma classe base. As referências do tipo da classe base poderiam ser armazenadas na lista, e vinculação dinâmica poderia ser usada para chamar os métodos de desenho. O inconveniente dessa estratégia é que cada classe de ave precisaria ser modificada para transformar as novas classes em subclasses da classe base comum. Seria melhor se as novas classes de ave pudessem simplesmente ser adicionadas a um arquivo de código, sem modificação. Outra solução possível seria usar instanceof e cast para determinar os tipos concretos das referências. Isso adicionaria muito código ao sistema, aumentando sua complexidade e o custo de manutenção. Uma solução melhor é usar vinculação dinâmica, que é possível com reflexão.

12.6.3 Reflexão em Java

Java oferece suporte limitado para reflexão. A principal classe dos metadados é definida no espaço de nomes `java.lang.Class`.[15] Essa classe tem o nome desafortunadamente enganoso, `Class`. O sistema de tempo de execução Java instancia uma instância de `Class` para cada objeto do programa. A classe `Class` fornece uma coleção de métodos para examinar as informações de tipo e os membros dos objetos do programa. `Class` é o ponto de acesso para toda a API de reflexão.

Se o programa tem uma referência a um objeto (não um primitivo), o objeto `Class` desse objeto pode ser obtido chamando-se seu método `getClass`. Todas as classes herdam `getClass` de `Object`, a partir da qual todos os objetos descendem. Considere os exemplos a seguir:

```
float[] totals = new float[100];
Class fltlist = totals.getClass();
Class stg = "hello".getClass();
```

O valor da variável `fltlist` será o objeto `Class` do objeto vetor `totals`. O valor de `stg` será o objeto `Class` de `String` (porque `"hello"` é uma instância de `String`).

Se não há nenhum objeto de uma classe, seu objeto `Class` pode ser obtido por meio do nome da classe, anexando-se `.class` a ele. Por exemplo, poderíamos ter o seguinte:

```
Class stg = String.class;
```

Se a classe não tem nome, seu objeto `Class` ainda pode ser obtido anexando-se `.class` à definição da classe. Por exemplo, considere o seguinte:

```
Class intmat = int[][].class;
```

[15] Ela é assim chamada porque suas instanciações são classes.

O modificador `.class` também pode ser anexado a tipos primitivos. Embora `float.getClass()` seja inválido, `float.class` não é.

Existem quatro métodos para se obter a `Class` de um método. O método `getMethod` pesquisa uma classe para encontrar um método público específico definido na classe ou herdado por ela. O método `getMethods` retorna um vetor de todos os métodos públicos definidos em uma classe ou herdados por ela. O método `getDeclaredMethod` procura um método específico declarado em uma classe, incluindo métodos privados. O método `getDeclaredMethods` retorna todos os métodos definidos em uma classe.

Se o objeto `Class` de um objeto é conhecido e é encontrado um método em particular definido pela classe do objeto, esse método pode ser chamado por meio do objeto `Method` do método, com o método `invoke`. Por exemplo, se o objeto `Method` chamado `method` é encontrado com `getMethod`, ele pode ser chamado com o seguinte:

```
method.invoke(…);
```

Agora podemos criar uma solução em Java para o problema apresentado na Seção 12.6.2. O centro dessa aplicação é uma classe que define um método que recebe uma referência a `Object`. O método determina a classe da referência passada, encontra um método `draw` dessa classe e chama-o. A classe da solução é testada com uma segunda classe, `ReflectTest`, a qual cria um vetor de três referências a `Object` para classes que representam três aves diferentes. Cada uma delas define um método `draw` que, quando chamado, exibe uma mensagem indicando a chamada. Então, o teste chama o método da classe, passando os elementos do vetor de referências.

O método chamador pode lançar três exceções diferentes, cada uma das quais tratada no método.

```
// Um projeto para ilustrar chamada de método dinâmica
//  usando reflexão em Java
package reflect;
import java.lang.reflect.*;
// Uma classe para testar a classe Reflect
//  Cria três objetos que representam aves diferentes
//  e chama um método que chama dinamicamente os métodos draw
//  das três classes de ave
public class ReflectTest {
   public static void main(String[] args) {
         Object[] birdList = new Object[3];
         birdList[0] = new Bird1();
         birdList[1] = new Bird2();
         birdList[2] = new Bird3();
         Reflect.callDraw(birdList[2]);
         Reflect.callDraw(birdList[0]);
         Reflect.callDraw(birdList[1]);
        }
}
```

```
// Uma classe para definir o método que chama dinamicamente os
//  métodos de um objeto classe passado
class Reflect {
    public static void callDraw(Object birdObj) {
        Class cls = birdObj.getClass();
        try {
            // Encontra o método draw da classe dada
            Method method = cls.getMethod("draw");
            // Chama o método dinamicamente
            method.invoke(birdObj);
        }
        // No caso de a classe dada não suportar draw
        catch (NoSuchMethodException e) {
            throw new IllegalArgumentException (
                cls.getName() + "does not support draw");
        }
        // No caso de callDraw não poder chamar draw
        catch (IllegalAccessException e) {
            throw new IllegalArgumentException (
                "Insufficient access permissions to call" +
                "draw in class " + cls.getName());
        }
        // No caso de draw lançar uma exceção
        catch (InvocationTargetException e) {
            throw new RuntimeException(e);
        }
      }
}
class Bird1 {
    public void draw() {
        System.out.println("This is draw from Bird1");
    }
}
class Bird2 {
    public void draw() {
        System.out.println("This is draw from Bird2");
    }
}
class Bird3 {
    public void draw() {
        System.out.println("This is draw from Bird3");
    }
]
```

A saída desse programa é:

```
This is the draw from Bird3
This is the draw from Bird1
This is the draw from Bird2
```

12.6.4 Reflexão em C#

O suporte para reflexão em C# é semelhante ao de Java, com algumas diferenças importantes. Em C#, como em todas as linguagens .NET, o compilador coloca o código intermediário, escrito em Common Intermediate Language (CIL), em uma montagem, a qual pode incluir vários arquivos. Uma montagem também contém um número de versão e os metadados de todas as classes nela definidas, assim como de todas as classes externas que utiliza.

Em vez do espaço de nomes `java.lang.Class`, `System.Type` é usado em .NET; em vez de `java.lang.reflect`, `System.Reflection` é usado. Em vez do método `getClass`, `getType` é usado para obter a classe de uma instância. Além disso, as linguagens .NET usam o operador `typeof` no lugar do campo `.class` utilizado em Java. A seguir está uma versão em C# do projeto Java mostrado anteriormente:

```
using System;
using System.Reflection;
namespace TestReflect
{
// Um projeto para ilustrar chamada de método dinâmica
// usando reflexão em C#
// Uma classe para testar a classe Reflect
//  Cria três objetos que representam aves diferentes
//  e chama um método que chama dinamicamente os métodos draw
//  das três classes de ave
public class ReflectTest {
   public static void Main(String[] args) {
        Object[] birdList = new Object[3];
        birdList[0] = new Bird1();
        birdList[1] = new Bird2();
        birdList[2] = new Bird3();
        Reflect.callDraw(birdList[2]);
        Reflect.callDraw(birdList[0]);
        Reflect.callDraw(birdList[1]);
       }
}
// Uma classe para definir o método que chama dinamicamente os
//  métodos de um objeto classe passado
class Reflect {
    public static void callDraw(Object birdObj) {
        Type typ = birdObj.GetType();
            // Encontra o método draw da classe dada
            MethodInfo method = typ.GetMethod("draw");
            // Chama o método dinamicamente
            method.Invoke(birdObj, null);
    }
}
class Bird1 {
    public void draw() {
        Console.WriteLine("This is draw from Bird1");
    }
```

```
}
class Bird2 {
    public void draw() {
        Console.WriteLine("This is draw from Bird2");
    }
}
class Bird3 {
    public void draw() {
        Console.WriteLine("This is draw from Bird3");
    }
}
}
```

Nosso exemplo simples de vinculação dinâmica de método mostra apenas um dos muitos usos da reflexão.

Além dos métodos e campos de uma classe, os seguintes elementos de programa podem ser acessados com reflexão, tanto em Java como em C#: modificadores de classe – como public, static e final –, construtores, tipos de parâmetro de método e interfaces implementadas. Além disso, pode-se fazer a introspecção de uma descrição do caminho de herança de uma classe. Em C#, mas não em Java, os nomes dos parâmetros formais de métodos podem ser descobertos.

Uma diferença significativa entre a reflexão de Java e a de C# é o espaço de nomes `System.Reflection.Emit`, que faz parte de .NET. Esse espaço de nomes oferece a capacidade de criar o código CIL e uma montagem para abrigar esse código. Java não oferece tal capacidade, embora isso possa ser feito com ferramentas de outros fornecedores.

Apesar de a reflexão adicionar uma variedade de recursos às linguagens de tipo estático Java e C#, o usuário de reflexão deve conhecer seus inconvenientes:

- O desempenho quase sempre é prejudicado com o uso de reflexão. Resolver tipos, métodos e campos em tempo de execução não faz parte do custo de executar código não reflexivo. Além disso, quando tipos são resolvidos dinamicamente, algumas otimizações não podem ser feitas no código.
- A reflexão expõe campos e métodos privados, o que viola as regras da abstração e da ocultação de informação, e também pode resultar em efeitos colaterais inesperados e afetar negativamente a portabilidade.
- Embora haja a vantagem de a verificação de tipos antecipada ser amplamente aceita, a vinculação tardia, que é possível com a reflexão, obviamente a contradiz.
- Algumas operações reflexivas podem não funcionar quando o código é executado sob um gerenciador de segurança, tornando-o também não portável. Um desses ambientes de segurança é o da execução de applets. Na maioria dos casos, se um problema pode ser resolvido sem reflexão, ela não deve ser usada.

A reflexão é parte integrante da maioria das linguagens dinamicamente tipadas. Em LISP, ela é usada rotineiramente, e a construção e a execução dinâmicas de código não são incomuns. Em outras linguagens interpretadas, como JavaScript, Perl e Python, a tabela de símbolos é mantida durante a interpretação, fornecendo todas as informações de tipo úteis.

Em Python, por exemplo, o método `type` retorna o tipo de um valor dado. Por exemplo, `type([7, 14, 21])` é `list`. O método `isinstance` retorna um valor booleano se seu primeiro parâmetro tem o tipo nomeado em seu segundo parâmetro. Por exemplo, `isinstance(17, int)` retorna `True`. A função `callable` é usada para determinar se uma expressão retorna um objeto função. A função `dir` retorna a lista de atributos, tanto dados como métodos, de seu objeto parâmetro.

RESUMO

A programação orientada a objetos é baseada em três conceitos fundamentais: tipos de dados abstratos, herança e vinculação dinâmica. Linguagens de programação orientadas a objetos suportam o paradigma com classes, métodos, objetos e passagem de mensagens.

A discussão sobre as linguagens de programação orientadas a objetos neste capítulo gira em torno de sete questões de projeto: exclusividade de objetos, subclasses e subtipos, verificação de tipo e polimorfismo, herança simples e múltipla, vinculação dinâmica, liberação explícita ou implícita de objetos e classes aninhadas.

Smalltalk é uma linguagem orientada a objetos pura – tudo é objeto e todas as computações são realizadas por passagem de mensagens. Toda a verificação de tipos e toda a vinculação de mensagens a métodos são dinâmicas e toda a herança é simples. Smalltalk não tem um operador de liberação explícito.

C++ fornece suporte para abstração de dados, herança e vinculação dinâmica opcional de mensagens para métodos, com todos os recursos convencionais de C. Isso significa que ela tem dois sistemas de tipo distintos. C++ fornece herança múltipla e liberação explícita de objetos. Ela inclui uma variedade de controles de acesso para as entidades em classes, alguns dos quais impedem que as subclasses sejam subtipos. Tanto métodos construtores quanto destrutores podem ser incluídos nas classes; normalmente, ambos são implicitamente chamados.

Embora a vinculação dinâmica de Smalltalk forneça um pouco mais de flexibilidade de programação que a linguagem híbrida C++, é muito menos eficiente.

Objective-C suporta programação procedural e orientada a objetos. Ela é menos complexa e menos usada que C++. Apenas herança simples é suportada, embora tenha categorias, as quais permitem mixins de métodos que podem ser adicionados a uma classe. Ela também tem protocolos, que são semelhantes às interfaces de Java. Uma classe pode adotar qualquer número de protocolos. Construtores podem ter qualquer nome, mas devem ser chamados explicitamente. O polimorfismo é suportado com o tipo predefinido **`id`**. Uma variável de tipo **`id`** pode referenciar qualquer objeto. Quando um método é chamado por meio de um objeto referenciado por uma variável de tipo **`id`**, a vinculação é dinâmica.

Diferentemente de C++, Java não é uma linguagem híbrida; ela foi projetada para suportar apenas programação orientada a objetos. Java tem tanto tipos escalares primitivos quanto classes. Todos os objetos são alocados do monte e acessados por variáveis de referência. Não existe uma operação de liberação explícita de objetos – a coleta de lixo é usada. Os únicos subprogramas são métodos, e eles podem ser chamados apenas por meio de objetos ou de classes. Apenas herança simples é diretamente suportada, apesar de um tipo de herança múltipla ser possível, usando-se interfaces. Toda a vinculação

de mensagens a métodos é dinâmica, exceto no caso de métodos que não podem ser sobrescritos. Além das classes, Java inclui pacotes como uma segunda construção de encapsulamento.

C#, baseada em C++ e em Java, suporta programação orientada a objetos. Objetos podem ser instanciados a partir de classes ou de estruturas. Os objetos de estruturas são dinâmicos da pilha e não suportam herança. Métodos em uma classe derivada podem chamar os métodos ocultos da classe pai incluindo **`base`** no nome do método. Métodos que podem ser sobrescritos devem ser marcados com **`virtual`** e métodos que sobrescrevem devem ser marcados com **`override`**. Todas as classes (e todos os tipos primitivos) são derivadas de `Object`.

Ruby é uma linguagem de *scripting* orientada a objetos na qual todos os dados são objetos. Como em Smalltalk, todos os objetos são alocados do monte e todas as variáveis são referências desprovidas de tipo para objetos. Todos os construtores são chamados de `initialize`. Todos os dados de instância são privados, mas métodos de leitura e de escrita podem ser facilmente incluídos. A coleção de todas as variáveis de instância para as quais métodos de acesso foram fornecidos forma a interface pública para a classe. Tais dados de instância são chamados de atributos. As classes Ruby são dinâmicas no sentido de serem executáveis e poderem ser modificadas a qualquer momento. Ruby suporta apenas herança simples.

As variáveis de instância de uma classe são armazenadas em um RIC, cuja estrutura é estática. Subclasses têm seus próprios RICs, assim como o RIC de suas classes pais. A vinculação dinâmica é suportada com uma tabela de métodos virtual, a qual armazena ponteiros para métodos específicos. A herança múltipla complica muito a implementação de RICs e de tabelas de método virtual.

QUESTÕES DE REVISÃO

1. Descreva os três recursos característicos das linguagens orientadas a objetos.
2. Qual é a diferença entre uma variável de classe e uma variável de instância?
3. O que é herança múltipla?
4. O que é uma variável polimórfica?
5. O que é um método sobrescrevedor?
6. Descreva uma situação na qual a vinculação dinâmica tem uma grande vantagem sobre a vinculação estática.
7. O que é um método virtual?
8. O que é um método abstrato? O que é uma classe abstrata?
9. Descreva brevemente as sete questões de projeto usadas neste capítulo para linguagens orientadas a objetos.
10. O que é uma classe aninhadora?
11. O que é o protocolo de mensagem de um objeto?

12. De onde os objetos em Smalltalk são alocados?
13. Explique como as mensagens Smalltalk são vinculadas a métodos. Quando isso ocorre?
14. Que verificação de tipos é feita em Smalltalk? Quando ela ocorre?
15. Que tipo de herança, simples ou múltipla, Smalltalk suporta?
16. Quais são os dois efeitos mais importantes de Smalltalk na computação?
17. Basicamente, todas as variáveis Smalltalk são de um único tipo. Que tipo é esse?
18. De onde os objetos de C++ podem ser alocados?
19. Como os objetos alocados do monte em C++ são liberados?
20. Todas as subclasses em C++ são subtipos? Se sim, explique. Se não, por quê?
21. Sob quais circunstâncias uma chamada a método em C++ é estaticamente vinculada a um método?
22. Que desvantagem existe em permitir que os projetistas especifiquem que métodos podem ser estaticamente vinculados?
23. Quais são as diferenças entre derivações privadas e públicas em C++?
24. O que é uma função amiga (*friend*) pura em C++?
25. O que é uma função virtual pura em C++?
26. Como os parâmetros são enviados para o construtor de uma superclasse em C++?
27. Qual é a diferença prática mais importante entre Smalltalk e C++?
28. Se um método em Objective-C não retorna nada, qual é o tipo de retorno indicado em seu cabeçalho?
29. Objective-C suporta herança múltipla?
30. Uma classe em Objective-C não pode especificar uma classe pai em seu cabeçalho?
31. O que é a classe raiz em Objective-C?
32. Em Objective-C, como um método pode indicar que não pode ser sobrescrito nas classes descendentes?
33. Qual é o propósito de uma categoria em Objective-C?
34. Qual é o propósito de um protocolo em Objective-C?
35. Qual é o principal uso do tipo `id` em Objective-C?
36. Como o sistema de tipos de Java é diferente do de C++?
37. De onde os objetos em Java podem ser alocados?
38. O que é encaixotamento (*boxing*)?
39. Como os objetos em Java são liberados?

40. Todas as subclasses em Java são subtipos?
41. Como os construtores da superclasse são chamados em Java?
42. Sob quais circunstâncias uma chamada a método em Java é estaticamente vinculada a um método?
43. De que maneira métodos sobrecarregados em C# diferem sintaticamente de seus correspondentes em C++?
44. Como a versão pai de um método herdado sobrescrito em uma subclasse pode ser chamada em tal subclasse em C#?
45. Como Ruby implementa tipos primitivos, como aqueles para dados inteiros ou de ponto flutuante?
46. Como os métodos de leitura são definidos em uma classe Ruby?
47. Que controles de acesso Ruby suporta para variáveis de instância?
48. Que controles de acesso Ruby suporta para métodos?
49. Todas as subclasses em Ruby são subtipos?
50. Ruby suporta herança múltipla?
51. O que a reflexão permite que um programa faça?
52. No contexto da reflexão, o que são metadados?
53. O que é introspecção?
54. O que é intercessão?
55. Qual classe em Java armazena informações sobre classes em um programa?
56. Para que é usada a extensão de nome Java `.class`?
57. O que o método Java `getMethods` faz?
58. Para que é usado o espaço de nomes C# `System.Reflection.Emit`?

PROBLEMAS

1. Que parte importante do suporte para programação orientada a objetos faltava em SIMULA 67?
2. Explique o princípio da substituição.
3. Explique as maneiras pelas quais podem ser criadas subclasses que não são subtipos.
4. Compare a vinculação dinâmica de C++ com a de Java.
5. Compare os controles de acesso de entidades de classe de C++ e Java.
6. Compare a herança múltipla de C++ com a fornecida por interfaces em Java.

7. Cite uma situação de programação na qual a herança múltipla tenha uma vantagem significativa sobre as interfaces.
8. Explique os dois problemas dos tipos de dados abstratos minimizados pela herança.
9. Descreva as categorias de mudanças que uma subclasse pode realizar em sua classe pai.
10. Explique uma desvantagem da herança.
11. Explique as vantagens e as desvantagens de se ter todos os valores em uma linguagem como objetos.
12. O que exatamente significa para uma subclasse ter um relacionamento é-um(a) com sua classe pai?
13. Descreva a questão de quão fortemente os parâmetros de um método sobrescrevedor devem se associar os parâmetros do método que sobrescreve.
14. Explique a verificação de tipos de Smalltalk.
15. Os projetistas de Java obviamente pensaram que não valia a pena o ganho adicional de eficiência ao se permitir que qualquer método fosse estaticamente vinculado, como no caso de C++. Quais são os argumentos a favor e contra o projeto de Java?
16. Qual é a principal razão pela qual todos os objetos Java têm um ancestral comum?
17. Qual é o propósito da cláusula **`finalize`** em Java?
18. O que seria ganho se Java permitisse objetos dinâmicos da pilha, assim como objetos dinâmicos do monte? Qual seria a desvantagem de se ter ambos?
19. Quais são as diferenças entre uma classe abstrata C++ e uma interface Java?
20. Compare o suporte para polimorfismo em C++ com o de Objective-C.
21. Compare os recursos e o uso de protocolos Objective-C com a interfaces Java.
22. Avalie de forma crítica a decisão tomada pelos projetistas de Objective-C de usar a sintaxe da Smalltalk para chamadas de método, em vez da sintaxe convencional utilizada pela maioria das linguagens imperativas que suportam programação orientada a objetos.
23. Explique por que permitir que uma classe implemente múltiplas interfaces em Java e em C# não origina os mesmos problemas criados pela herança múltipla em C++.
24. Estude e explique por que C# não inclui as classes aninhadas não estáticas de Java.
25. Você pode definir uma variável de referência para uma classe abstrata? Que uso tal variável teria?
26. Compare os controles de acesso para variáveis de instância em Java e em Ruby.
27. Compare a detecção de erros de tipo para variáveis de instância em Java e em Ruby.
28. Explique os inconvenientes da reflexão.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Reescreva as classes `single_linked_list`, `stack_2`, e `queue_2` da Seção 12.5.2 em Java e compare o resultado com a versão C++ em termos de legibilidade e facilidade de programação.
2. Repita o Exercício de programação 1 usando Ruby.
3. Repita o Exercício de programação 1 usando Objective_C.
4. Projete e implemente um programa que defina uma classe base A, que tem uma subclasse B, que por sua vez tem uma subclasse C. A classe A deve implementar um método sobrescrito tanto em B quanto em C. Você também deve escrever uma classe de testes que instancie A, B e C e que inclua três chamadas a método. Uma das chamadas deve ser vinculada estaticamente ao método de A. Uma chamada deve ser vinculada dinamicamente ao método de B e uma deve ser vinculada dinamicamente ao método de C. Todas as chamadas a métodos devem ser feitas por meio de um ponteiro para a classe A.
5. Escreva um programa em C++ que chame um grande número de vezes tanto um método dinamicamente vinculado quanto um método estaticamente vinculado, cronometrando o tempo das chamadas para os dois métodos. Compare os tempos e calcule a diferença de tempo exigida pelos dois. Explique os resultados.
6. Repita o Exercício de programação 4 com Java, forçando a vinculação estática com **`final`**.

13

Concorrência

13.1 Introdução

13.2 Introdução à concorrência em nível de subprograma

13.3 Semáforos

13.4 Monitores

13.5 Passagem de mensagens

13.6 Suporte de Ada para concorrência

13.7 Linhas de execução em Java

13.8 Linhas de execução em C#

13.9 Concorrência em linguagens funcionais

13.10 Concorrência em nível de sentença

Este capítulo começa com introduções aos vários tipos de concorrência em nível de subprogramas ou de unidades, e também em nível de sentenças. É incluída uma breve descrição dos tipos mais comuns de arquiteturas multiprocessadas de computadores. A seguir, é apresentada uma extensa discussão sobre concorrência em nível de unidade. Ela começa com uma descrição dos conceitos fundamentais que devem ser entendidos antes de discutirmos os problemas e desafios do suporte linguístico para a concorrência em nível de unidade, especificamente a sincronização de competição e a de cooperação. A seguir, são descritas as questões de projeto relacionadas ao fornecimento de suporte linguístico para concorrência. Posteriormente, há uma discussão detalhada sobre as três principais estratégias para o suporte linguístico para concorrência: semáforos, monitores e passagem de mensagens. Um programa exemplo em pseudocódigo é apresentado para demonstrar como os semáforos podem ser usados. Ada e Java são usadas para ilustrar monitores; para passagem de mensagens, utilizamos Ada. Os recursos de Ada que suportam concorrência são descritos em algum detalhe. Apesar de as tarefas serem o foco, objetos protegidos (os quais são, efetivamente, monitores) também são discutidos. Então, é discutido o suporte para concorrência em nível de unidade em Java e C#, incluindo estratégias de sincronização. Esse assunto é seguido por breves panoramas do suporte para concorrência em várias linguagens de programação funcionais. A última seção do capítulo é uma breve discussão sobre a concorrência em nível de sentença, incluindo uma introdução à parte do suporte de linguagem fornecido a ela em Fortran de Alto Desempenho.

13.1 INTRODUÇÃO

A concorrência na execução de software pode ocorrer em quatro níveis: em nível de instrução (executar duas ou mais instruções de máquina simultaneamente), em nível de sentença (executar duas ou mais sentenças de linguagem de alto nível simultaneamente), em nível de unidade (executar duas ou mais unidades de subprograma simultaneamente) e em nível de programa (executar dois ou mais programas simultaneamente). Como nenhuma questão de projeto de linguagem está envolvida nelas, as concorrências em nível de instrução e em nível de programa não são discutidas neste capítulo. As concorrências em nível de subprograma e de sentença são discutidas, especialmente a primeira.

À primeira vista, a concorrência pode parecer um conceito simples, mas apresenta desafios significativos para o programador, para o projetista de linguagem de programação e para o projetista de sistema operacional (pois grande parte do suporte para concorrência é fornecida pelo sistema operacional).

Mecanismos de controle de concorrência aumentam a flexibilidade da programação. Eles foram originalmente desenvolvidos para serem usados em problemas particulares oriundos dos sistemas operacionais, mas são exigidos por uma variedade de outras aplicações de programação. Alguns dos programas mais usados são os navegadores Web, cujo projeto é fortemente baseado em concorrência. Os navegadores devem executar muitas funções diferentes ao mesmo tempo, entre elas enviar e receber dados de servidores Web, desenhar texto e imagens na tela e reagir às ações do usuário com o mouse e com o teclado. Alguns navegadores contemporâneos, como o Internet Explorer 9, usam

os processadores de núcleo extras, que fazem parte de muitos computadores pessoais atuais, para realizar algum processamento, por exemplo, a interpretação de código de *scripting* no lado do cliente. Outro exemplo são os sistemas de software projetados para simular sistemas físicos reais que consistem em vários subsistemas concorrentes. Para todos esses tipos de aplicações, a linguagem de programação (ou uma biblioteca ou, pelo menos, o sistema operacional) deve suportar concorrência em nível de unidade.

A concorrência em nível de sentença é, de certa forma, diferente. Do ponto de vista de um projetista de linguagem, a concorrência em nível de sentença é uma questão de especificar como os dados devem ser distribuídos por várias memórias e quais sentenças podem ser executadas concorrentemente.

O objetivo de desenvolver software concorrente é produzir algoritmos concorrentes escaláveis e portáveis. Um algoritmo concorrente é **escalável** se a velocidade de sua execução aumenta quando mais processadores estão disponíveis. Isso é importante, porque o número de processadores às vezes aumenta com as novas gerações de máquinas. Os algoritmos devem ser portáveis, porque o tempo de vida de hardware é relativamente curto. Logo, os sistemas de software não devem depender de uma arquitetura específica – ou seja, devem funcionar eficientemente em máquinas com diferentes arquiteturas.

A intenção deste capítulo é discutir os aspectos de concorrência mais relevantes para questões de projeto de linguagem, e não apresentar um estudo definitivo sobre todas as questões de concorrência, incluindo o desenvolvimento de programas concorrentes. Claramente, isso seria inadequado para um livro sobre linguagens de programação.

13.1.1 Arquiteturas multiprocessadas

Diversas arquiteturas de computadores têm mais de um processador e podem suportar alguma forma de execução concorrente. Antes de iniciarmos a discussão sobre a execução concorrente de programas e de sentenças, descreveremos brevemente algumas dessas arquiteturas.

Os primeiros computadores com múltiplos processadores apresentavam um processador de propósito geral e um ou mais processadores, chamados de periféricos, usados apenas para operações de entrada e saída. Essa arquitetura permitiu àqueles computadores, surgidos no final dos anos 1950, executar um programa enquanto realizavam entrada ou saída para outros programas.

No início dos anos 1960, havia máquinas que tinham vários processadores completos. Esses processadores eram usados pelo escalonador de processos do sistema operacional, o qual distribuía processos separados de uma fila de processos em lote para os processadores separados. Sistemas com essa estrutura suportavam concorrência em nível de programa.

As máquinas que apareceram em meados dos anos 1960 tinham vários processadores parciais idênticos. Eles eram alimentados com instruções de um único fluxo. Por exemplo, algumas máquinas possuíam dois ou mais multiplicadores de ponto flutuante, enquanto outras tinham duas ou mais unidades aritméticas de ponto flutuante completas. Os compiladores para essas máquinas precisavam determinar quais instruções poderiam ser executadas concorrentemente e agendar essas instruções de forma correta. Sistemas com essa estrutura suportavam concorrência em nível de instruções.

Em 1966, Michael J. Flynn sugeriu uma categorização das arquiteturas de computadores, definida em relação aos fluxos de dados e de instruções, que poderiam ser simples ou múltiplos. Os nomes dessas categorias foram bastante usados desde os anos 1970 até o início dos 2000. As duas categorias que usavam vários fluxos de dados são definidas como segue: os computadores que tinham vários processadores executando a mesma instrução simultaneamente, cada um com dados diferentes, são chamados de computadores de arquitetura SIMD – Single Instruction, Multiple Data (uma instrução, múltiplos dados). Em um computador SIMD, cada processador tem sua própria memória local. Um processador controla a operação dos outros processadores. Como todos os processadores, exceto o controlador, executam a mesma operação ao mesmo tempo, nenhuma sincronização é necessária em software. Talvez as máquinas SIMD mais utilizadas formem uma categoria de **processadores vetoriais**. Elas têm grupos de registros que armazenam os operandos de uma operação vetorial, na qual a mesma instrução é executada simultaneamente no conjunto inteiro de operandos. Originalmente, os tipos de programas que podiam se beneficiar mais dessa arquitetura eram para computação científica, uma área geralmente alvo de máquinas multiprocessadas. Contudo, os processadores SIMD agora são usados para diversas áreas de aplicação, entre elas o processamento de imagens gráficas e vídeo. Até pouco tempo, em sua maioria, os supercomputadores eram processadores vetoriais.

Computadores com múltiplos processadores que operam independentemente, mas cujas operações podem ser sincronizadas, são chamados de MIMD – Multiple Instruction Multiple Data (múltiplas instruções, múltiplos dados). Cada processador em um computador MIMD executa seu próprio fluxo de instruções. Computadores MIMD podem ter duas configurações distintas: sistemas de memória compartilhados e distribuídos. As máquinas MIMD distribuídas, nas quais cada processador tem sua própria memória, podem ser construídas tanto em um único chassi quanto distribuídas, talvez por uma grande área. As máquinas MIMD de memória compartilhada obviamente devem fornecer alguma forma de sincronização para evitar colisões de acesso à memória. Mesmo máquinas MIMD distribuídas exigem sincronização para operarem juntas em programas únicos. Computadores MIMD, mais gerais que os computadores SIMD, suportam concorrência em nível de unidade. O foco principal deste capítulo é o projeto de linguagens para computadores MIMD de memória compartilhada, os quais frequentemente são chamados de **multiprocessados**.

O advento de chips únicos de computadores – poderosos, mas de baixo custo – tornou possível ter um grande número desses microprocessadores conectados em pequenas redes dentro de um único chassi. Apareceram diversos fabricantes desses tipos de computadores, que usam microprocessadores de prateleira.

O software não evoluiu rápido o suficiente para usar as máquinas concorrentes em razão do crescente poder dos processadores. Um dos principais apelos das máquinas concorrentes é o aumento da velocidade de computação. Entretanto, dois fatores ligados a hardware foram combinados para fornecer computação mais rápida, sem exigir mudança na arquitetura dos sistemas de software. Primeiro, as taxas de *clock* dos processadores têm se tornado mais rápidas a cada geração de processadores (mais ou menos a cada 18 meses). Segundo, diversos tipos de concorrência vêm predefinidos nas arquiteturas dos processadores. Entre eles estão o *pipelining* de instruções e dados da

memória para o processador (as instruções são obtidas e decodificadas para execução futura, enquanto a instrução atual é executada), o uso de linhas separadas para instruções e dados, pré-carregando instruções e dados, e o paralelismo na execução de operações aritméticas. Todos esses tipos são coletivamente chamados de **concorrência oculta**. O resultado dos aumentos na velocidade de execução são grandes ganhos de produtividade que não exigem que os desenvolvedores produzam sistemas de software concorrentes.

Entretanto, a situação está mudando. O fim da sequência de aumentos significativos na velocidade de processadores individuais está próximo. Aumentos significativos no poder da computação resultam agora do aumento no número de processadores, por exemplo, grandes sistemas de servidor, como aquele executado pelo Google e pela Amazon e por aplicações de pesquisa científica. Muitas outras tarefas de computação volumosas são agora executadas em máquinas com grande número de processadores relativamente pequenos.

Outro avanço recente no hardware de computação foi o desenvolvimento de vários processadores em um único chip, como os Intel Core Duo e Core Quad, que estão pressionando os desenvolvedores de software a usar mais os múltiplos processadores disponíveis nas máquinas. Se eles não fizerem isso, a concorrência em hardware será perdida e os ganhos de produtividade diminuirão significativamente.

13.1.2 Categorias de concorrência

Existem duas categorias de controle de unidades concorrentes. A mais natural é aquela na qual, assumindo a disponibilidade de mais de um processador, diversas unidades do mesmo programa são executadas simultaneamente. Essa é a **concorrência física**. Um ligeiro relaxamento desse conceito de concorrência permite ao programador e ao aplicativo de software assumirem a existência de múltiplos processadores fornecendo concorrência, quando a execução real dos programas está ocorrendo de maneira intercalada em um processador. Essa é a **concorrência lógica**. Do ponto de vista do programador e do projetista de linguagem, concorrência lógica é o mesmo que concorrência física. É tarefa do implementador da linguagem, por meio das capacidades do sistema operacional, mapear a concorrência lógica no sistema de hardware hospedeiro. Tanto a concorrência lógica quanto a física permitem que o conceito de concorrência seja usado como uma metodologia de projeto de programas. Para o restante deste capítulo, a discussão se aplica tanto à concorrência física quanto à lógica.

Uma técnica útil para visualizar o fluxo de execução por meio de um programa é imaginar uma linha de execução (*thread*) sobre as sentenças do código-fonte do programa. Cada sentença alcançada em determinada execução é coberta pela linha que representa essa execução. Seguir visualmente a linha pelo programa-fonte rastreia o fluxo de execução pela versão executável do programa. É claro, em todos os programas, exceto nos mais simples, a linha de execução segue um caminho muito complexo, impossível de ser acompanhada visualmente. Formalmente, uma **linha de execução de controle** em um programa é a sequência de pontos alcançada à medida que o controle flui por ele.

Entretanto, programas com corrotinas (veja o Capítulo 9), mas não subprogramas concorrentes, são chamados de **quasi-concorrentes**, com apenas uma linha de execução

de controle. Programas executados com concorrência física podem ter múltiplas linhas de execução de controle. Cada processador pode executar uma das linhas. Apesar de a execução de programas logicamente concorrentes poder ter apenas uma linha de execução, eles podem ser projetados e analisados somente imaginando-os com múltiplas linhas de execução de controle. Diz-se que um programa projetado para ter mais de uma linha de execução de controle é ***multithreaded***. Quando um programa com múltiplas linhas de execução de controle executa em uma máquina com um processador, suas linhas de execução são mapeadas para uma única linha de execução. Nesse cenário, ele se torna um programa praticamente *multithreaded*.

A concorrência em nível de sentença é um conceito relativamente simples. Em um uso comum dela, laços com sentenças inclusas que operam em elementos de matrizes são separados, de forma que possam distribuir o processamento por vários processadores. Por exemplo, um laço que executa 500 repetições e inclui uma sentença que opera em um dos 500 elementos de uma matriz pode ser separado de forma que cada um dos 10 processadores possam processar simultaneamente 50 elementos da matriz.

13.1.3 Motivações para o uso de concorrência

Existem ao menos quatro razões para se projetar sistemas de software concorrentes. A primeira é a velocidade de execução de programas em máquinas com múltiplos processadores. Essas máquinas fornecem uma maneira efetiva de aumentar a velocidade de execução dos programas, desde que sejam projetados para usar a concorrência em hardware. Existe agora um grande número de computadores instalados com múltiplos processadores, incluindo muitos dos pessoais comercializados nos últimos anos. É um desperdício não usar essa capacidade de hardware.

A segunda razão é que, mesmo quando uma máquina tem apenas um processador, um programa escrito para usar execução concorrente pode ser mais rápido que o mesmo programa escrito para execução sequencial (não concorrente). O requisito para que isso aconteça é o programa não ser vinculado à computação (a versão sequencial não utiliza totalmente o processador).

A terceira razão é que a concorrência fornece um método diferente de conceituar soluções de programas para problemas. Muitos domínios de problema se prestam naturalmente à concorrência, da mesma forma que a recursão é uma maneira natural de projetar soluções para alguns problemas. Além disso, muitos programas são escritos para simular entidades e atividades físicas. Em muitos casos, o sistema simulado inclui mais de uma entidade, e elas fazem tudo simultaneamente – por exemplo, aeronaves voando em um espaço aéreo controlado, estações de relés em uma rede de comunicações e as várias máquinas em uma indústria. Software que usa concorrência deve ser utilizado para simular tais sistemas de forma precisa.

A quarta razão para se usar concorrência é a programação de aplicações que são distribuídas por várias máquinas, localmente ou pela Internet. Muitas máquinas, por exemplo, os carros, têm mais de um computador embarcado, cada um dedicado a alguma tarefa específica. Em muitos casos, essas coleções de computadores precisam sincronizar suas execuções de programa. Jogos pela Internet são outro exemplo de software distribuído por múltiplos processadores.

Agora a concorrência é usada em numerosas tarefas de computação diárias. Servidores Web processam pedidos de documentos concorrentemente. Navegadores Web

utilizam processadores de núcleo secundário para fazer processamento gráfico e interpretar código de programação incorporado a documentos. Em todo sistema operacional existem muitos processos concorrentes sendo executados o tempo todo, gerenciando recursos, obtendo entrada de teclados, exibindo saída de programas e lendo e escrevendo em dispositivos de memória externos. Em resumo, a concorrência se tornou um aspecto onipresente da computação.

13.2 INTRODUÇÃO À CONCORRÊNCIA EM NÍVEL DE SUBPROGRAMA

Antes de considerarmos o suporte de linguagem para concorrência, devemos entender os conceitos subjacentes à concorrência e os requisitos para que tal suporte seja útil. Esses tópicos são abordados nesta seção.

13.2.1 Conceitos fundamentais

Uma **tarefa** é uma unidade de um programa, similar a um subprograma, que pode estar em execução concorrente com outras unidades do mesmo programa. Cada tarefa pode suportar uma linha de execução de controle. Às vezes as tarefas são chamadas de **processos**. Em algumas linguagens, por exemplo, Java e C#, certos métodos servem como tarefas. Tais métodos são executados em objetos chamados **linhas de execução**.

Três características das tarefas as distinguem dos subprogramas. Primeiro, uma tarefa pode ser implicitamente iniciada, enquanto um subprograma precisa ser explicitamente chamado. Segundo, quando uma unidade de programa invoca uma tarefa, em alguns casos ela não precisa esperar que a tarefa conclua sua execução antes de continuar a sua própria. Terceiro, quando a execução de uma tarefa estiver concluída, o controle pode ou não retornar à unidade que iniciou sua execução.

As tarefas são enquadradas em duas categorias gerais: pesadas (*heavyweight*) e leves (*lightweight*). Simplificando, uma **tarefa pesada** executa em seu próprio espaço de endereçamento. Já as **tarefas leves** são todas executadas no mesmo espaço de endereçamento. É mais fácil implementar tarefas leves do que tarefas pesadas. Além disso, elas podem ser mais eficientes do que as pesadas, porque menos esforço é necessário para gerenciar sua execução.

Uma tarefa pode se comunicar com outras por meio de variáveis não locais compartilhadas, passagem de mensagens ou parâmetros. Se uma tarefa não se comunica com outra ou não afeta a execução de qualquer outra no programa, é chamada de **disjunta**. Como as tarefas geralmente trabalham juntas para criar simulações ou resolver problemas – e, dessa forma, são não disjuntas –, devem usar a mesma forma de comunicação para sincronizar suas execuções ou compartilhar dados (ou ambos).

A **sincronização** é um mecanismo que controla a ordem na qual as tarefas são executadas. Dois tipos de sincronização são necessários quando as tarefas compartilham dados: cooperação e competição. A **sincronização de cooperação** é necessária entre as tarefas `A` e `B` quando `A` deve esperar que `B` conclua alguma tarefa específica antes que possa começar ou continuar sua execução. A **sincronização de competição** é necessária entre duas tarefas quando ambas exigem o uso de algum recurso que não pode

ser simultaneamente utilizado. Especificamente, se a tarefa A precisa acessar a posição de dados compartilhada x enquanto a tarefa B está acessando x, A deve aguardar que B conclua seu processamento de x. Então, para a sincronização de cooperação, talvez as tarefas precisem esperar o término do processamento específico do qual sua operação correta depende, enquanto para a sincronização de competição as tarefas talvez precisem esperar o término de qualquer outro processamento, feito por qualquer outra tarefa, que atualmente estiver ocorrendo em dados compartilhados específicos.

Uma forma simples de sincronização de cooperação pode ser ilustrada por um problema comum, chamado de **problema do produtor-consumidor**. Esse problema foi originado no desenvolvimento de sistemas operacionais, no qual uma unidade de programa produz algum valor de dado ou recurso e outra unidade o utiliza. Os dados produzidos são normalmente colocados em um *buffer* de armazenamento pela unidade produtora e removidos pela unidade consumidora. A sequência de armazenamentos e de remoções do *buffer* deve ser sincronizada. A unidade consumidora não pode obter dados do *buffer* se ele estiver vazio. Da mesma forma, o produtor não pode colocar novos dados se ele estiver cheio. Esse é um problema de sincronização de cooperação, porque os usuários da estrutura de dados compartilhada têm de cooperar para que o *buffer* seja utilizado corretamente.

A sincronização de competição evita que duas tarefas acessem uma estrutura de dados compartilhada exatamente ao mesmo tempo – essa situação poderia destruir a integridade dos dados compartilhados. Para fornecer sincronização de competição deve ser garantido o acesso mutuamente exclusivo aos dados compartilhados.

Para esclarecer o problema da competição, considere o cenário a seguir: suponha que a tarefa A tenha a sentença `TOTAL += 1`, onde `TOTAL` é uma variável inteira compartilhada. Além disso, suponha que a tarefa B tenha a sentença `TOTAL *= 2`. As tarefas A e B poderiam tentar modificar `TOTAL` ao mesmo tempo.

Em nível de linguagem de máquina, cada tarefa pode efetuar sua operação em `TOTAL` com o seguinte processo de três etapas:

1. Obter o valor de `TOTAL`.
2. Efetuar a operação aritmética.
3. Colocar o novo valor de volta em `TOTAL`.

Sem a sincronização de competição, dadas as operações descritas anteriormente, efetuadas pelas tarefas A e B em `TOTAL`, quatro valores diferentes poderiam resultar, dependendo da ordem das etapas das operações. Suponha que `TOTAL` tenha o valor `3` antes que A ou B tente modificá-la. Se a tarefa A completasse sua operação antes do início de B, o valor seria `8`, que aqui assumimos estar correto. Mas se tanto A como B obtivessem o valor de `TOTAL` antes de qualquer uma das tarefas colocar seu novo valor de volta, o resultado seria incorreto. Se A colocasse seu valor de volta primeiro, o valor de `TOTAL` seria `6`. Esse caso é mostrado na Figura 13.1. Se B colocasse seu valor de volta primeiro, o valor de `TOTAL` seria `4`. Por fim, se B completasse sua operação antes de A iniciar, o valor seria `7`. Uma situação que leva a esses problemas às vezes é chamada de **condição de corrida**, porque duas ou mais tarefas estão correndo para usar o recurso compartilhado e o comportamento do programa depende de qual tarefa chega primeiro (e vence a corrida). A importância da sincronização de competição deve estar clara agora.

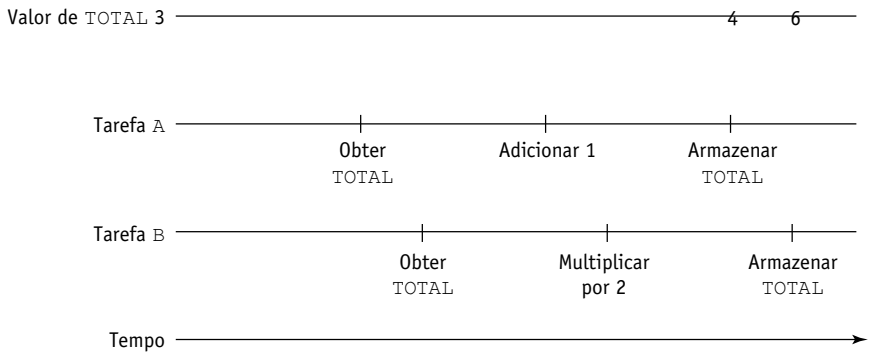


FIGURA 13.1
A necessidade de sincronização de competição.

Um método geral para fornecer acesso mutuamente exclusivo (para suportar a sincronização de competição) a um recurso compartilhado é considera-lo como algo que uma tarefa pode possuir e permitir que apenas uma única tarefa possa possuí-lo em dado momento. Para obter a posse de um recurso compartilhado, uma tarefa deve requisitá-lo. A posse só será dada se nenhuma outra tarefa estiver com ela. Enquanto uma tarefa possui um recurso, todas as outras tarefas são impedidas de acessar esse recurso. Quando uma tarefa tiver terminado de usar um recurso seu, deve liberá-lo a fim de que ele fique disponível para as outras.

Três métodos de fornecimento de acesso mutuamente exclusivo para um recurso compartilhado são: os semáforos, discutidos na Seção 13.3; os monitores, discutidos na Seção 13.4; e a passagem de mensagens, discutida na Seção 13.5.

Mecanismos para sincronização devem ser capazes de atrasar a execução de uma tarefa. A sincronização impõe uma ordem de execução das tarefas garantida com esses atrasos. Para entender o que acontece com as tarefas ao longo de seus tempos de vida, precisamos considerar como a execução de tarefas é controlada. Independentemente de uma máquina ter um ou mais processadores, sempre existe a possibilidade de haver mais tarefas que processadores. Um programa de sistema de tempo de execução, chamado **escalonador**, gerencia o compartilhamento de processadores entre as tarefas. Se nunca existissem interrupções e todas as tarefas tivessem a mesma prioridade, o escalonador poderia simplesmente dar a cada tarefa uma fatia de tempo, como 0,1 segundo, e quando o turno de uma tarefa chegasse, o escalonador poderia deixá-la ser executada por essa quantidade de tempo em um processador. É claro, existem diversos eventos complicadores, atrasos de tarefa para sincronização e para operações de entrada e saída. Como operações de entrada e saída são muito lentas em relação à velocidade do processador, uma tarefa não pode manter um processador enquanto espera pela conclusão de uma delas.

As tarefas podem estar em diversos estados:

1. *Nova*: quando foi criada, mas ainda não iniciou sua execução.
2. *Pronta*: uma tarefa pronta para ser executada, mas não executada atualmente. Ou ainda não foi dado a ela tempo de processador pelo escalonador, ou ela já executou anteriormente, mas foi bloqueada de uma das maneiras descritas no quarto parágra-

fo desta subseção. As tarefas prontas para serem executadas são armazenadas em uma fila chamada de **fila de tarefas prontas**.

3. *Executando*: uma tarefa que está sendo executada; ou seja, tem um processador e seu código está sendo executado.
4. *Bloqueada*: uma tarefa que estava executando, mas foi interrompida por um entre diversos eventos – o mais comum é uma operação de entrada ou de saída. Além de entrada e saída, algumas linguagens fornecem operações para que um programa de usuário especifique que uma tarefa deve ser bloqueada.
5. *Morta*: uma tarefa morta não está mais ativa em nenhum sentido. Ela morre quando sua execução está concluída, ou quando é morta explicitamente pelo programa.

Um diagrama de fluxo dos estados de uma tarefa é mostrado na Figura 13.2.

Uma questão importante na execução de tarefas é a seguinte: como uma tarefa pronta é escolhida para ser movida para o estado "executando" quando a tarefa que está atualmente sendo executada se tornou bloqueada ou a fatia de tempo expirou? Diver-

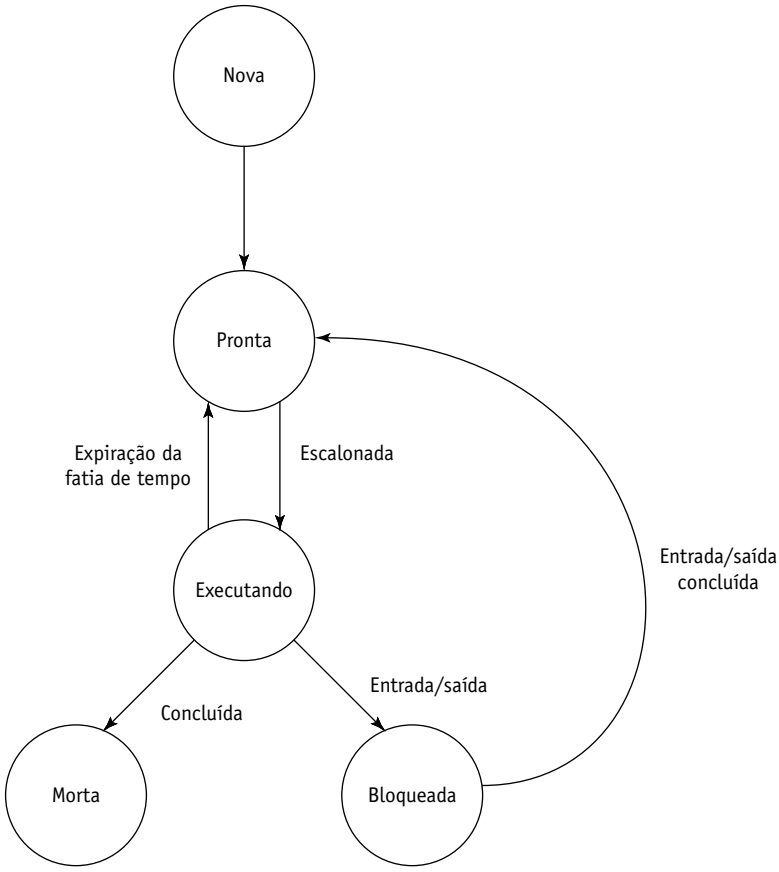


FIGURA 13.2
Diagrama de fluxo de estados de tarefa.

sos algoritmos diferentes têm sido usados para fazer essa escolha, alguns baseados em níveis especificáveis de prioridade. O algoritmo que faz a escolha é implementado no escalonador.

Associado à execução concorrente de tarefas e ao uso de recursos compartilhados está o conceito de vivacidade (*liveness*). No ambiente de programas sequenciais, um programa tem a característica de **vivacidade** se continua a executar, eventualmente sendo concluído. Em termos mais gerais, a vivacidade significa que, se um evento – digamos, o término do programa – supostamente ocorreria, ele ocorrerá, em um momento ou outro. Ou seja, o progresso é continuamente feito. Em um ambiente concorrente e com recursos compartilhados, a vivacidade de uma tarefa pode deixar de existir, isto é, o programa pode não continuar e, dessa forma, nunca terminará.

Por exemplo, suponha que tanto a tarefa `A` quanto a tarefa `B` precisem dos recursos compartilhados `X` e `Y` para completar seu trabalho. Além disso, suponha que `A` ganhe a posse de `X` e `B` ganhe a posse de `Y`. Após algum tempo de execução, a tarefa `A` precisa do recurso `Y` para continuar, então ela solicita `Y`, mas deve esperar até que `B` o libere. De modo similar, a tarefa `B` solicita `X`, mas deve esperar até que `A` o libere. Nenhuma delas libera os recursos que possui e, como resultado, ambas perdem sua vivacidade, garantindo que a execução do programa nunca será completada normalmente. Esse tipo específico de perda de vivacidade é chamado de **impasse** (*deadlock*). *Deadlock* é uma ameaça séria para a confiabilidade de um programa – dessa forma, evitá-lo demanda sérias considerações tanto no projeto de linguagens quanto no de programas.

Agora, estamos prontos para discutir alguns dos mecanismos linguísticos utilizados para fornecer controle de unidades concorrentes.

13.2.2 Projeto de linguagem para concorrência

Em alguns casos, a concorrência é implementada por meio de bibliotecas. Entre elas está a OpenMP, uma interface de programação de aplicações usada para suportar programação multiprocessada de memória compartilhada em C, C++ e Fortran, em uma variedade de plataformas. Nosso interesse neste livro, evidentemente, é o suporte da linguagem para a concorrência. Diversas linguagens foram projetadas para suportar a concorrência, começando com PL/I, em meados dos anos 1960, e incluindo as linguagens contemporâneas Ada 95, Java, C#, F#, Python e Ruby.[1]

13.2.3 Questões de projeto

As questões de projeto mais importantes para o suporte de linguagem para concorrência já foram discutidas em detalhes: sincronização de competição e de cooperação. Além delas, existem diversos pontos de importância secundária. Proeminente entre eles é a questão de como uma aplicação pode influenciar o escalonamento de tarefas. Também existem questões relacionadas a como e quando as tarefas iniciam e terminam suas execuções, e como e quando elas são criadas.

[1]Nos casos de Python e Ruby, os programas são interpretados; portanto, só pode haver concorrência lógica. Mesmo que a máquina tenha vários processadores, esses programas não podem utilizar mais de um.

Tenha em mente que nossa discussão sobre concorrência é intencionalmente incompleta, e são discutidas apenas as questões de projeto de linguagem mais importantes relacionadas ao suporte à concorrência.

As seções a seguir discutem três estratégias alternativas às questões de projeto para concorrência: semáforos, monitores e passagem de mensagens.

13.3 SEMÁFOROS

Um semáforo é um mecanismo simples que pode ser usado para fornecer sincronização de tarefas. Embora os semáforos sejam uma estratégia primitiva para fornecer sincronização, eles ainda são utilizados tanto em linguagens contemporâneas como em sistemas com suporte à concorrência baseado em biblioteca. Nos parágrafos seguintes, descrevemos os semáforos e discutimos como eles podem ser usados para esse propósito.

13.3.1 Introdução

Em um esforço para fornecer sincronização de competição por meio de acesso mutuamente exclusivo às estruturas de dados compartilhadas, Edsger Dijkstra projetou os semáforos em 1965 (Dijkstra, 1968b). Os semáforos também podem ser usados para prover sincronização de cooperação.

Para fornecer acesso limitado a uma estrutura de dados, guardas podem ser colocadas em torno do código que acessa a estrutura. Uma **guarda** é um dispositivo linguístico que permite ao código guardado ser executado apenas quando uma condição específica for verdadeira. Assim, uma guarda pode ser usada para permitir que apenas uma tarefa acesse uma estrutura de dados específica compartilhada em dado momento. Um semáforo é uma implementação de uma guarda. Especificamente, um **semáforo** é uma estrutura de dados que consiste em um inteiro e em uma fila que armazena descritores de tarefas. Um **descritor de tarefa** é uma estrutura de dados que armazena todas as informações relevantes acerca do estado de execução de uma tarefa.

Uma parte integrante de um mecanismo de guarda é um procedimento usado para garantir que todas as execuções tentadas do código de guarda ocorram em algum momento. A estratégia típica é ter requisições para o acesso que ocorram quando ele não pode ser dado, ou que sejam armazenadas na fila de descritores de tarefas, da qual elas podem ser obtidas posteriormente, de forma a ser permitido deixar e executar o código guardado. Essa é a razão pela qual um semáforo deve ter tanto um contador quanto uma fila de descritores de tarefa.

As duas operações fornecidas pelos semáforos foram originalmente chamadas de P e V por Dijkstra, em referência às duas palavras holandesas *passeren* (passar) e *vrygeren* (liberar) (Andrews e Schneider, 1983). No restante desta seção, vamos nos referir a essas operações como *wait* e *release*, respectivamente.

13.3.2 Sincronização de cooperação

Em grande parte deste capítulo, usamos o exemplo de um *buffer* compartilhado utilizado por produtores e consumidores a fim de ilustrar as diferentes estratégias para fornecer

sincronização de cooperação e de competição. Para a sincronização de cooperação, o *buffer* deve ter alguma maneira de gravar tanto o número de posições vazias quanto o de posições preenchidas (para impedir transbordamentos [*overflows*] e transbordamentos negativos [*underflows*] do *buffer*). O componente de contagem de um semáforo pode ser usado para esse propósito. Uma variável de semáforo – por exemplo, `emptyspots` – pode usar seu contador para manter o número de posições vazias em um *buffer* compartilhado usado por produtores e consumidores, e outra – digamos, `fullspots` – pode usar seu contador para manter o número de posições preenchidas no *buffer*. As filas desses semáforos podem armazenar os descritores de tarefas forçadas a esperar pelo acesso ao *buffer*. A fila `emptyspots` pode armazenar tarefas de produtor à espera por posições disponíveis no *buffer*; a fila `fullspots` pode armazenar tarefas de consumidor que estão aguardando os valores serem colocados no *buffer*.

Nosso *buffer* de exemplo é projetado como um tipo de dados abstrato no qual todos os dados entram por meio do subprograma `DEPOSIT` e todos deixam o *buffer* por meio do subprograma `FETCH`. O subprograma `DEPOSIT` precisa apenas verificar o semáforo `emptyspots` para ver se existe alguma posição vazia. Se existir ao menos uma, ele pode continuar com `DEPOSIT`, o que deve ter o efeito colateral de decrementar o contador de `emptyspots`. Se o *buffer* estiver cheio, o chamador de `DEPOSIT` deve esperar na fila de `emptyspots` até que uma posição vaga seja disponibilizada. Quando `DEPOSIT` tiver concluído sua tarefa, o subprograma `DEPOSIT` incrementa o contador do semáforo `fullspots` para indicar que existe mais uma posição preenchida no *buffer*.

O subprograma `FETCH` tem a sequência oposta de `DEPOSIT`. Ele verifica o semáforo `fullspot` para ver se o *buffer* contém ao menos um item. Se contém, um item é removido e o semáforo `emptyspots` tem seu contador incrementado por 1. Se o *buffer* estiver vazio, a tarefa chamadora é colocada na fila de `fullspots` para esperar até que um item apareça. Quando `FETCH` termina, ele precisa incrementar o contador de `emptyspots`.

As operações em tipos semáforos geralmente não são diretas – elas são feitas por meio dos subprogramas `wait` e `release`. Logo, a operação `DEPOSIT` descrita anteriormente é, na verdade, realizada em parte por chamadas a `wait` e a `release`. Note que `wait` e `release` devem ser capazes de acessar a fila de tarefas prontas.

O subprograma de semáforo `wait` é usado para testar o contador de uma variável semáforo. Se o valor for maior que zero, o chamador pode continuar sua operação. Nesse caso, o valor do contador da variável semáforo é decrementado para indicar que agora há um item a menos daquilo que o semáforo estiver contando. Se o valor do contador é zero, o chamador deve ser colocado na fila de espera da variável semáforo e outra tarefa pronta deve ser dada ao processador.

O subprograma de semáforo `release` é usado por uma tarefa para permitir a outra ter um item do contador da variável de semáforo especificado, independentemente do item que tal variável estiver contando. Se a fila da variável semáforo especificada estiver vazia, ou seja, se nenhuma tarefa está esperando, `release` incrementa seu contador (para indicar a existência de mais um item sendo controlado e agora disponível). Se uma ou mais tarefas estão esperando, `release` move uma delas da fila do semáforo para a fila de tarefas prontas.

A seguir estão descrições concisas em pseudocódigo de `wait` e `release`:

```
wait(aSemaphore)
if contador de umSemaforo > 0 then
    decrementa contador de umSemaforo
else
    coloca o chamador na fila de umSemaforo
    tenta transferir o controle para alguma tarefa pronta
    (se a fila de tarefas prontas estiver vazia, ocorre um impasse)
end if

release(umSemaforo)
if a fila de umSemaforo estiver vazia (nenhuma tarefa está esperando) then
    incrementa o contador de umSemaforo
else
    coloca a tarefa chamadora na fila de tarefas prontas
    transfere o controle para uma tarefa da fila de umSemaforo
end
```

Agora, podemos apresentar um programa de exemplo que implementa sincronização de cooperação para um *buffer* compartilhado. Nesse caso, o *buffer* compartilhado armazena valores inteiros e é uma estrutura logicamente circular. Ele é projetado para o uso de várias tarefas produtoras e consumidoras.

O seguinte pseudocódigo mostra a definição das tarefas produtora e consumidora. Dois semáforos são usados para garantir que não ocorrerão transbordamentos do *buffer* (positivos ou negativos), fornecendo sincronização de cooperação. Suponha que o *buffer* tenha tamanho `BUFLEN` e que as rotinas que na verdade o manipulam já existam como `FETCH` e `DEPOSIT`. Os acessos ao contador de um semáforo são especificados pela notação por pontos. Por exemplo, se `fullspots` é um semáforo, seu contador é referenciado como

```
semaphore fullspots, emptyspots;
fullspots.count = 0;
emptyspots.count = BUFLEN;
task producer;
  loop
  -- produce VALUE --
  wait(emptyspots);     { espera por um espaço }
  DEPOSIT(VALUE);
  release(fullspots);   { aumenta os espaços preenchidos }
  end loop;
end producer;

task consumer;
   loop
```

```
   wait(fullspots);      { garante que não esteja vazio }
   FETCH(VALUE);
   release(emptyspots); { aumenta os espaços vazios }
   -- consume VALUE --
   end loop
end consumer;
```

O semáforo `fullspots` faz a tarefa `consumer` ser enfileirada para esperar por uma entrada no *buffer* se ele estiver vazio. O semáforo `emptyspots` faz a tarefa `producer` ser enfileirada para esperar por um espaço vazio no *buffer* se ele estiver cheio.

13.3.3 Sincronização de competição

Nosso exemplo de *buffer* não fornece sincronização de competição. O acesso à estrutura pode ser controlado com um semáforo adicional. Esse semáforo não precisa contar nada, mas pode simplesmente indicar com seu contador se o *buffer* está em uso. A sentença `wait` permite o acesso apenas se o contador do semáforo tiver o valor `1`, indicando que o *buffer* compartilhado não é acessado. Se o contador do semáforo tem o valor `0`, existe um acesso atual ocorrendo e a tarefa é colocada na fila do semáforo. Note que o contador do semáforo deve ser inicializado com `1`. As filas dos semáforos devem ser sempre inicializadas como vazias antes que seu uso possa começar.

Um semáforo que requer apenas um contador de valor binário, como aquele usado para fornecer sincronização de competição no exemplo a seguir, é chamado de **semáforo binário**.

O pseudocódigo a seguir ilustra o uso de semáforos para fornecer tanto sincronização de competição quanto de cooperação para um *buffer* compartilhado acessado concorrentemente. O semáforo `access` é usado para garantir acesso mutuamente exclusivo ao *buffer*. Lembre-se de que pode existir mais de um produtor e mais de um consumidor.

```
semaphore access, fullspots, emptyspots;
access.count = 1;
fullspots.count = 0;
emptyspots.count = BUFLEN;

task producer;
  loop
  -- produzir VALUE --
  wait(emptyspots);      { espera por um espaço }
  wait(access);          { espera por acesso }
  DEPOSIT(VALUE);
  release(access);       { libera o acesso }
  release(fullspots);    { aumenta os espaços preenchidos }
  end loop;
end producer;
```

```
task consumer;
  loop
  wait(fullspots);       { garante que não esteja vazio }
  wait(access);          { espera por acesso }
  FETCH(VALUE);
  release(access);       { libera o acesso }
  release(emptyspots);   { aumenta os espaços vazios }
  -- consumir VALUE --
  end loop
end consumer;
```

Uma breve olhada nesse exemplo pode levar alguém a acreditar que existe um problema. Especificamente, suponha que, enquanto uma tarefa está esperando a chamada `wait(access)` em `consumer`, outra receba o último valor do *buffer* compartilhado. Felizmente, isso não pode ocorrer, já que `wait(fullspots)` reserva um valor no *buffer* para a tarefa que o chama por meio do decremento do contador `fullspots`.

Existe um aspecto crucial dos semáforos até agora não discutido. Lembre-se da descrição anterior do problema da sincronização de competição: operações em dados compartilhados não devem se sobrepor. Se uma segunda operação começa enquanto uma operação anterior ainda está em andamento, os dados compartilhados podem ser corrompidos. Um semáforo é um objeto de dados compartilhado, logo as operações nos semáforos também são suscetíveis ao mesmo problema. É essencial que as operações de semáforo não possam ser interrompidas. Muitos computadores têm instruções que não podem ser interrompidas, projetadas especificamente para operações de semáforos. Se tais instruções não estão disponíveis, usar semáforos para fornecer sincronização de competição é um problema sério, sem uma solução simples.

13.3.4 Avaliação

Usar semáforos para sincronização de cooperação cria um ambiente de programação inseguro. Não há como verificar estaticamente a exatidão de seu uso, o qual depende da semântica do programa em que aparecem. No exemplo do *buffer*, omitir a sentença `wait(emptyspots)` da tarefa `producer` resultaria em um transbordamento do *buffer*. Omitir `wait(fullspots)` da tarefa `consumer` resultaria em um transbordamento negativo do *buffer*. Omitir qualquer uma das liberações resultaria em um impasse. Essas são falhas de sincronização de cooperação.

Os problemas de confiabilidade que os semáforos causam ao fornecer sincronização de cooperação também surgem quando são eles usados para sincronização de competição. Omitir a sentença `wait(access)` em qualquer uma das tarefas pode causar acessos inseguros ao *buffer*. Omitir a sentença `release(access)` em qualquer das tarefas resultaria em um impasse. Essas são falhas de sincronização de competição. Notando o perigo do uso de semáforos, Per Brinch Hansen (1973) escreveu: "O semáforo é uma ferramenta de sincronização elegante para um programador ideal que nunca comete erros". Infelizmente, programadores ideais são raros.

13.4 MONITORES

Uma solução para alguns dos problemas dos semáforos em um ambiente concorrente é encapsular as estruturas de dados compartilhadas com suas operações e ocultar suas representações – ou seja, fazer com que elas sejam tipos de dados abstratos, com algumas restrições especiais. Essa solução pode fornecer sincronização de competição sem semáforos, transferindo a responsabilidade de sincronização para o sistema de tempo de execução.

13.4.1 Introdução

Quando os conceitos de abstração de dados foram formulados, as pessoas envolvidas nesse esforço aplicaram os mesmos conceitos a dados compartilhados em ambientes de programação concorrente para produzir monitores. De acordo com Per Brinch Hansen (Brinch Hansen, 1977, p. xvi), Edsger Dijkstra sugeriu, em 1971, que todas as operações de sincronização em dados poderiam ser descritas em apenas uma unidade de programa. Brinch Hansen (1973) formalizou esse conceito no ambiente de sistemas operacionais. No ano seguinte, Hoare (1974) chamou essas estruturas de *monitores*.

A primeira linguagem de programação a incorporar monitores foi Pascal Concorrente (Brinch Hansen, 1975). Modula (Wirth, 1977), CSP/k (Holt et al., 1978) e Mesa (Mitchell et al., 1979) também fornecem monitores. Entre as linguagens contemporâneas, eles são suportados por Ada, Java e C#, todas discutidas mais adiante neste capítulo.

13.4.2 Sincronização de competição

Um dos recursos mais importantes dos monitores é o fato de que os dados compartilhados residem no monitor, em vez de em uma das unidades clientes. O programador não sincroniza o acesso mutuamente exclusivo aos dados compartilhados por meio de semáforos ou outros mecanismos. Como os mecanismos de acesso fazem parte do monitor, a implementação de um pode ser feita de forma a garantir acesso sincronizado, permitindo apenas um acesso de cada vez. Chamadas a procedimentos do monitor são implicitamente bloqueadas e armazenadas em uma fila se ele estiver ocupado no momento da chamada.

13.4.3 Sincronização de cooperação

Apesar de o acesso mutuamente exclusivo a dados compartilhados ser intrínseco a um monitor, a cooperação entre processos ainda é tarefa do programador. Em particular, o programador deve garantir que um *buffer* compartilhado não sofra transbordamentos positivos ou negativos. Diferentes linguagens fornecem maneiras distintas de programar a sincronização de cooperação, todas relacionadas aos semáforos.

Um programa que contém quatro tarefas e um monitor que fornece acesso sincronizado a um *buffer* compartilhado concorrentemente é mostrado na Figura 13.3. Nessa

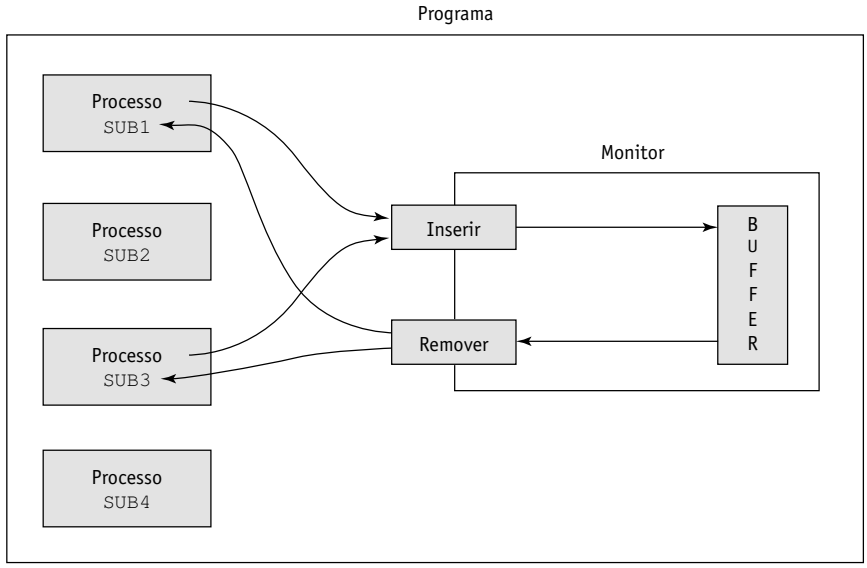


FIGURA 13.3
Um programa usando um monitor para controlar o acesso a um *buffer* compartilhado.

figura, a interface para o monitor é mostrada como duas caixas, rotuladas com `insert` (inserir) e `remove` (remover) (para a inserção e remoção de dados). O monitor aparece exatamente como um tipo de dado abstrato – uma estrutura de dados com acesso limitado –, ou seja, como ele é.

13.4.4 Avaliação

Monitores são uma forma melhor de fornecer sincronização de competição do que os semáforos, principalmente devido aos problemas dos semáforos discutidos na Seção 13.3. A sincronização de cooperação ainda é um problema dos monitores, o que ficará claro quando as implementações de monitores em Ada e em Java forem discutidas nas seções seguintes.

Semáforos e monitores são igualmente poderosos para expressar o controle de concorrência – os semáforos podem ser usados para implementar monitores e os monitores podem ser usados para implementar semáforos.

Ada fornece duas maneiras de implementar monitores. Ada 83 inclui um modelo geral de tarefas que pode ser usado para suportá-los. Ada 95 adicionou uma maneira mais limpa e eficiente de construí-los, chamada de *objetos protegidos*. Ambas as estratégias usam passagem de mensagens como o modelo básico para suportar concorrência. O modelo de passagem de mensagens permite que as unidades concorrentes sejam distribuídas, enquanto os monitores não permitem. A passagem de mensagens é descrita na Seção 13.5. O suporte de Ada para passagem de mensagens é discutido na Seção 13.6.

Em Java, um monitor pode ser implementado em uma classe projetada como um tipo de dados abstrato, com os dados compartilhados sendo o tipo. Acessos a objetos da

classe são controlados por meio da adição do modificador **`synchronized`** aos métodos de acesso. Um exemplo de um monitor para o *buffer* compartilhado escrito em Java é dado na Seção 13.7.4.

C# tem uma classe predefinida, `Monitor`, projetada para implementar monitores.

13.5 PASSAGEM DE MENSAGENS

Esta seção apresenta o conceito fundamental de passagem de mensagens na concorrência. Observe que ele não está relacionado à passagem de mensagens usada na programação orientada a objetos para executar métodos.

13.5.1 Introdução

Os primeiros esforços para projetar linguagens que fornecem a capacidade de passar mensagens entre tarefas concorrentes foram os de Brich Hansen (1978) e Hoare (1978). Eles também desenvolveram uma técnica para tratar do impasse que ocorria quando várias requisições simultâneas eram feitas por tarefas que desejavam se comunicar com uma dada tarefa. Foi decidido que alguma forma de não determinismo era necessária para haver justiça na escolha de qual requisição seria atendida primeiro. Essa justiça pode ser definida de várias maneiras, mas em geral significa que é dada a todos os requisitantes uma chance igual de se comunicar com uma tarefa (supondo que todos os requisitantes tenham a mesma prioridade). Construções não determinísticas para controle em nível de sentenças, chamadas de *comandos protegidos*, foram introduzidas por Dijkstra (1975). Os comandos protegidos são discutidos no Capítulo 8. Eles são a base da construção projetada para controlar a passagem de mensagens.

13.5.2 O conceito de passagem síncrona de mensagens

A passagem de mensagens pode ser síncrona ou assíncrona. Aqui, descrevemos a passagem síncrona. O seu conceito básico decorre do fato de que as tarefas estão normalmente ocupadas e não podem ser interrompidas por outras unidades. Suponha que as tarefas `A` e `B` estejam em execução e que `A` deseja enviar uma mensagem para `B`. Se `B` estiver ocupada, não é desejável permitir que outra tarefa a interrompa. Isso impediria seu o processamento atual. Além disso, as mensagens normalmente causam processamento associado no receptor, o que não seria desejado se outro processamento estivesse incompleto. A alternativa é fornecer um mecanismo linguístico capaz de permitir a uma tarefa especificar para outras quando ela está pronta para receber mensagens. Essa estratégia é, de alguma forma, parecida com a de um executivo que instrui sua secretária a colocar em espera todas as chamadas até que outra atividade, talvez uma conversa importante, tenha terminado. Posteriormente, quando a conversa atual estiver concluída, o executivo dirá à secretária que aceita conversar com uma das pessoas colocadas em espera.

Uma tarefa pode ser projetada de forma a suspender sua execução em determinado momento, seja porque está desocupada ou porque precisa de informações de outra unidade para poder continuar. Isso é como uma pessoa esperando uma chamada importante. Em alguns casos, não há nada a fazer além de sentar e esperar. Entretanto, se uma tarefa `A` está esperando por uma mensagem no momento em que `B` a envia, a mensagem pode

ser transmitida. Essa transmissão real da mensagem é chamada de ***rendezvous***. Note que um *rendezvous* pode ocorrer apenas se tanto o remetente quanto o destinatário quiserem que ele aconteça. Durante um *rendezvous*, a informação da mensagem pode ser transmitida em qualquer das direções (ou em ambas).

Tanto a sincronização de tarefas de cooperação quanto a de competição podem ser manipuladas pelo modelo de passagem de mensagens, como descrito na seção a seguir.

13.6 SUPORTE DE ADA PARA CONCORRÊNCIA

Esta seção descreve o suporte para concorrência fornecido por Ada. Ada 83 suporta apenas passagem de mensagens síncrona.

13.6.1 Fundamentos

O projeto de Ada para tarefas é parcialmente baseado no trabalho de Brinch Hansen e Hoare, no sentido de que a passagem de mensagens é o projeto básico e o não determinismo é usado para escolher entre as tarefas que enviaram mensagens.

O modelo completo de tarefas de Ada é complexo, e a discussão a seguir é limitada. O foco aqui será na versão de Ada com mecanismo de passagem síncrona de mensagens.

As tarefas Ada podem ser mais ativas que os monitores. Estes são entidades passivas que fornecem serviços de gerenciamento para os dados compartilhados que armazenam. Eles fornecem seus serviços, apesar de o fazerem apenas quando são solicitados. Quando usadas para gerenciar dados compartilhados, as tarefas Ada funcionam como gerentes que podem residir com os recursos que gerenciam. Elas têm diversos mecanismos, alguns determinísticos e outros não determinísticos, que as permitem escolher entre requisições que competem pelo acesso aos seus recursos.

Existem duas partes em uma tarefa Ada – uma de especificação e uma de corpo –, ambas com o mesmo nome. A interface de uma tarefa são seus pontos de entrada, ou posições onde ela pode receber mensagens de outras tarefas. Como esses pontos de entrada integram sua interface, é natural que sejam listados na parte de especificação de uma tarefa. Como um *rendezvous* pode envolver uma troca de informações, as mensagens podem ter parâmetros; logo, os pontos de entradas de tarefas ainda devem permitir parâmetros, os quais também devem ser descritos na parte de especificação. Na aparência, uma especificação de tarefa é similar ao pacote de especificação para um tipo de dados abstrato.

Como exemplo de especificação de tarefa em Ada, considere o código a seguir, que inclui um único ponto de entrada chamado `Entry_1`, que por sua vez tem um parâmetro no modo de entrada:

```
task Task_Example is
  entry Entry_1(Item : in Integer);
end Task_Example;
```

O corpo de uma tarefa deve incluir alguma forma sintática dos pontos de entrada que correspondem às cláusulas **entry** na parte de especificação da tarefa. Em Ada, esses pontos de entrada no corpo de tarefas são especificados por cláusulas introduzidas pela palavra reservada **accept**. Uma **cláusula accept** é definida na faixa de sentenças que começam com a palavra reservada **accept** e terminam com a palavra reservada **end** correspondente. As cláusulas **accept** em si são relativamente simples, mas outras construções nas quais elas podem ser incorporadas podem tornar sua semântica complexa. Uma cláusula **accept** simples tem a seguinte forma:

```
accept entry_name (formal parameters) do
  ...
end entry_name;
```

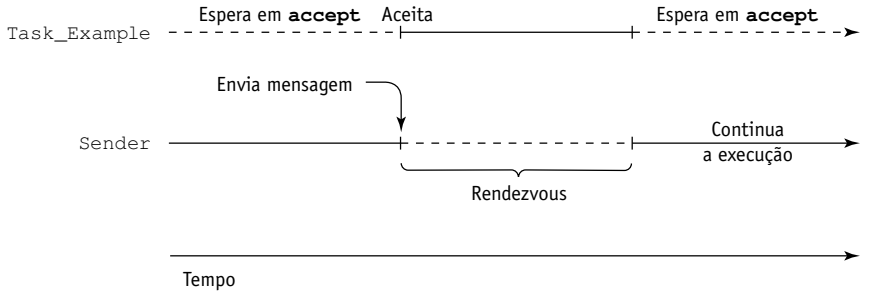
O nome de entrada em **accept** casa com o nome em uma cláusula **entry** na parte de especificação da tarefa associada. Os parâmetros opcionais fornecem as maneiras de comunicação de dados entre a tarefa chamadora e a chamada. As sentenças entre o **do** e o **end** definem as operações que ocorrem durante o *rendezvous*. Juntas, essas sentenças são chamadas de **corpo da cláusula accept**. Durante o *rendezvous* real, a tarefa remetente é suspensa.

Sempre que uma cláusula **accept** receber uma mensagem que não está pronta para aceitar por qualquer razão, a tarefa remetente deve ser suspensa até que a cláusula **accept** na tarefa receptora esteja pronta para aceitar a mensagem. É claro, a cláusula **accept** deve também lembrar das tarefas que enviaram mensagens não aceitas. Para esse propósito, cada cláusula **accept** em uma tarefa possui uma fila associada para armazenar uma lista de outras tarefas que tentaram se comunicar com ela sem sucesso.

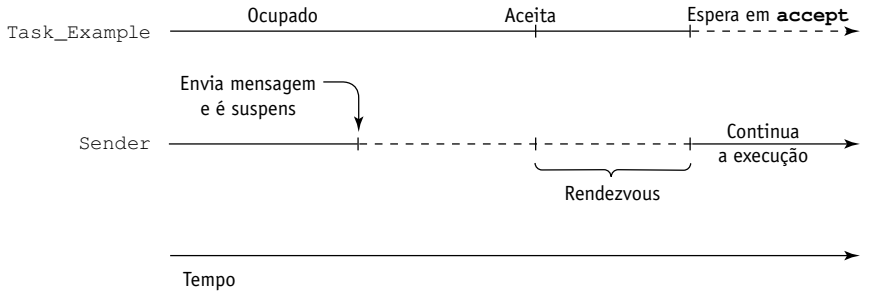
A seguir está o esqueleto do corpo da tarefa cuja especificação foi dada anteriormente:

```
task body Task_Example is
  begin
  loop
    accept Entry_1(Item : in Integer) do
      ...
    end Entry_1;
  end loop;
  end Task_Example;
```

A cláusula **accept** desse corpo de tarefa é a implementação da entrada chamada `Entry_1` na especificação da tarefa. Se a execução de `Task_Example` inicia e alcança a cláusula **accept** `Entry_1` antes de qualquer outra tarefa enviar uma mensagem a `Entry_1`, `Task_Example` é suspensa. Se outra tarefa envia uma mensagem a `Entry_1` enquanto `Task_Example` está suspensa em seu **accept**, um *rendezvous* ocorre e o corpo da cláusula **accept** é executado. Então, devido ao laço, a execução continua novamente a partir do **accept**. Se nenhuma outra tarefa enviou uma mensagem a `Entry_1`, a execução é mais uma vez suspensa para esperar pela próxima mensagem.



(a) `Task_Example` espera por `Sender`



(b) `Sender` espera por `Task_Example`

FIGURA 13.4
Duas maneiras pelas quais um *rendezvous* com `Task_Example` pode ocorrer.

Um *rendezvous* pode ocorrer de duas maneiras básicas nesse exemplo simples. Primeiro, a tarefa receptora, `Task_Example`, pode estar esperando que outra tarefa envie uma mensagem para a entrada `Entry_1`. Quando a mensagem é enviada, o *rendezvous* ocorre. Essa é a situação descrita anteriormente. Segundo, a tarefa receptora pode estar ocupada com um *rendezvous* ou com algum outro processamento não associado a um *rendezvous*, quando outra tarefa tenta enviar uma mensagem para a mesma entrada. Nesse caso, a remetente é suspensa até que a receptora esteja livre para aceitar essa mensagem em um *rendezvous*. Se diversas mensagens chegarem quando a receptora estiver ocupada, as remetentes são enfileiradas para esperar sua vez por um *rendezvous*.

Os dois *rendezvous* descritos anteriormente são ilustrados com os diagramas de linha do tempo na Figura 13.4.

As tarefas não precisam ter pontos de entrada. Elas são chamadas de **tarefas atrizes** (*actor tasks*), pois não precisam esperar por um *rendezvous* para realizar seu trabalho; podem realizar *rendezvous* com outras tarefas enviando mensagens a elas. Em contraste, uma tarefa pode ter cláusulas **`accept`**, mas nenhum código fora o das cláusulas **`accept`**, podendo apenas reagir a outras tarefas. Esta é chamada de **tarefa servidora**.

Uma tarefa Ada que envia uma mensagem para outra deve conhecer o nome de entrada na tarefa. Contudo, o oposto não é verdade: uma entrada de tarefa não precisa conhecer o nome das tarefas das quais ela receberá mensagens. Essa assimetria está em contraste com o projeto da linguagem conhecida como Processos Sequenciais Comunicantes (CSP - Communicating Sequential Processes) (Hoare, 1978). Nela, que também usa o modelo de concorrência de passagem de mensagens, as tarefas aceitam mensagens apenas de outras explicitamente nomeadas. A desvantagem desse projeto é que não podem ser construídas bibliotecas de tarefas para uso geral.

O método gráfico usual de descrever um *rendezvous* no qual a tarefa `A` envia uma mensagem para a tarefa `B` é mostrado na Figura 13.5.

As tarefas são declaradas na parte de declaração de um pacote, subprograma ou bloco. Tarefas[2] criadas estaticamente começam sua execução no mesmo momento das sentenças no código ao qual a parte declarativa está anexada. Por exemplo, uma tarefa declarada em um programa principal inicia sua execução no mesmo momento em que a primeira sentença no corpo de código do programa principal. O término de uma tarefa, que é uma questão complexa, é discutido posteriormente nesta seção.

As tarefas podem ter qualquer quantidade de entradas. A ordem na qual as cláusulas **`accept`** associadas aparecem na tarefa determina a ordem na qual as mensagens podem ser aceitas. Se uma tarefa tem mais de um ponto de entrada e requer que eles sejam capazes de receber mensagens em qualquer ordem, ela usa uma sentença **`select`** para

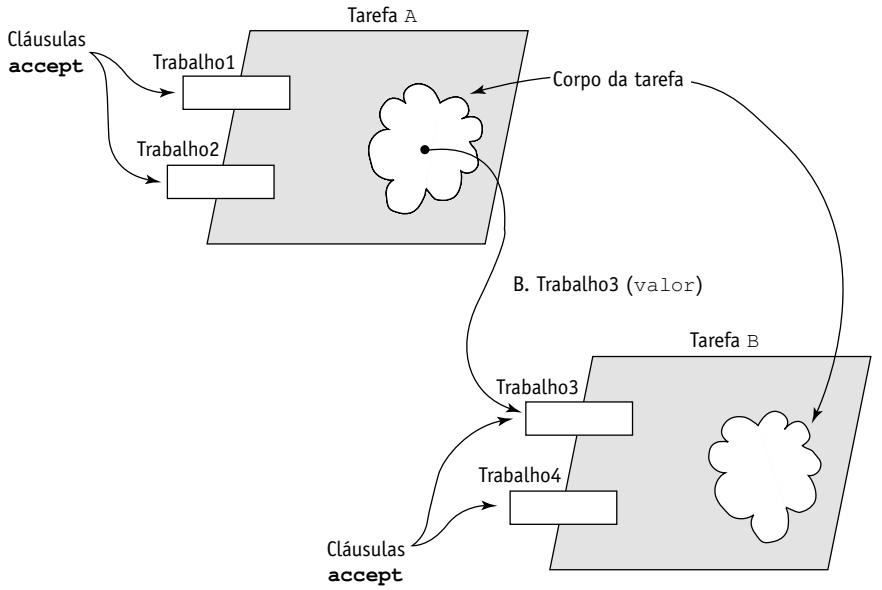


FIGURA 13.5
Representação gráfica de um *rendezvous* causado por uma mensagem enviada da tarefa `A` para a tarefa `B`.

[2] As tarefas também podem ser criadas dinamicamente, mas essas não são abordadas aqui.

envolver as entradas. Por exemplo, suponha que uma tarefa modele as atividades de um atendente bancário, que deve atender clientes dentro de uma agência e em uma janela do tipo *drive-up*. O seguinte esqueleto de tarefa para o atendente ilustra uma construção **select**:

```
task body Teller is
begin
  loop
    select
      accept Drive_Up(formal parameters) do
        ...
      end Drive_Up;
      ...
    or
      accept Walk_Up(formal parameters) do
        ...
      end Walk_Up;
      ...
    end select;
  end loop;
end Teller;
```

Nessa tarefa, existem duas cláusulas **accept**, `Walk_Up` e `Drive_Up`, cada uma com uma fila associada. A ação do **select**, quando executado, é examinar as filas associadas com as duas cláusulas **accept**. Se uma das filas está vazia, mas a outra contém ao menos uma mensagem esperando (cliente), a cláusula **accept** associada à mensagem (ou mensagens) em espera tem um *rendezvous* com a tarefa responsável por enviar a primeira mensagem recebida. Se ambas as cláusulas **accept** têm filas vazias, o **select** espera até que uma das entradas seja chamada. Se ambas as cláusulas **accept** têm filas não vazias, uma delas é escolhida não deterministicamente para ter um *rendezvous* com um de seus chamadores. O laço força a sentença **select** a ser executada repetidamente, para sempre.

O fim (**end**) da cláusula **accept** marca o fim do código que atribui ou referencia os parâmetros formais da cláusula. O código, se existir algum, entre uma cláusula **accept** e o próximo **or** (ou o **end select**, se a cláusula **accept** for a última no **select**) é chamado de **cláusula accept estendida**. Esta é executada apenas após a cláusula **accept** associada (que a precede imediatamente) ser executada. Essa execução da cláusula **accept** estendida não faz parte do *rendezvous* e pode ocorrer concorrentemente com a execução da tarefa chamadora. A remetente é suspensa durante o *rendezvous*, mas é colocada de volta na fila de tarefas prontas quando o final da cláusula **accept** é alcançado. Se uma cláusula **accept** não tem parâmetros formais, o **do–end** não é obrigatório e a cláusula **accept** pode consistir inteiramente em uma cláusula **accept** estendida. Tal cláusula **accept** seria usada exclusivamente para sincronização. Cláusulas **accept** estendidas são ilustradas na tarefa `Buf_Task` na Seção 13.6.3.

13.6.2 Sincronização de cooperação

Cada cláusula **accept** pode ter uma guarda anexada, na forma de uma cláusula **when**, que pode postergar um *rendezvous*. Por exemplo,

```
when not Full(Buffer) =>
  accept Deposit(New_Value) do
    ...
  end
```

Uma cláusula **accept** com uma **when** pode estar aberta ou fechada. Se a expressão booleana da cláusula **when** é verdadeira, a **accept** é chamada de aberta (**open**); se a expressão booleana é falsa, a **accept** é dita fechada (**closed**). Uma cláusula **accept** que não tem uma guarda é sempre aberta. Uma cláusula aberta está disponível para *rendezvous*; uma fechada não pode realizar um *rendezvous*.

Suponha que existam diversas cláusulas **accept** guardadas em uma cláusula **select**. Tal cláusula **select** normalmente é colocada em um laço infinito. O laço faz com que a cláusula **select** seja executada repetidamente, com cada cláusula **when** avaliada em cada repetição. Cada repetição faz uma lista de cláusulas **accept** abertas ser construída. Se precisamente uma das cláusulas abertas tiver uma fila não vazia, uma mensagem dessa fila é retirada e um *rendezvous* ocorre. Se mais de uma das cláusulas **accept** possuem filas não vazias, uma das filas é escolhida não deterministicamente, uma mensagem é retirada dela e um *rendezvous* ocorre. Se as filas de todas as cláusulas abertas estiverem vazias, a tarefa espera pela chegada de uma mensagem em uma das cláusulas **accept**, fazendo um *rendezvous* ocorrer no momento da chegada. Se um **select** for executado e todas as cláusulas **accept** estiverem fechadas, ocorrerá um erro ou exceção em tempo de execução. Essa possibilidade pode ser evitada certificando-se de que uma das cláusulas **when** é sempre verdadeira ou adicionando-se uma cláusula **else** no **select**. Uma cláusula **else** pode incluir qualquer sequência de sentenças, exceto uma cláusula **accept**.

Uma cláusula **select** pode ter uma sentença especial, **terminate**, selecionada apenas quando ela está aberta e nenhuma outra cláusula **accept** está. Uma cláusula **terminate**, quando selecionada, significa que o trabalho da tarefa está finalizado, mas ainda não terminado. O término de tarefas é discutido posteriormente nesta seção.

13.6.3 Sincronização de competição

Os recursos descritos até agora fornecem sincronização de cooperação e comunicação entre tarefas. A seguir, discutimos como o acesso mutuamente exclusivo às estruturas de dados compartilhadas pode ser realizado em Ada.

Se o acesso a uma estrutura de dados deve ser controlado por uma tarefa, o acesso mutuamente exclusivo pode ser obtido declarando-se a estrutura de dados dentro de uma tarefa. A semântica da execução de tarefa normalmente garante acesso mutuamente exclusivo à estrutura, porque apenas uma cláusula **accept** na tarefa pode estar ativa em dado momento. As únicas exceções para isso ocorrem quando as tarefas são aninhadas em procedimentos ou em outras tarefas. Por exemplo, se uma tarefa que define uma estrutura de dados compartilhada tem uma tarefa aninhada, esta também pode acessar a

estrutura compartilhada, o que poderia destruir a integridade dos dados. Logo, as tarefas que supostamente devem controlar o acesso a uma estrutura de dados compartilhada não devem definir tarefas.

A seguir, temos um exemplo de uma tarefa Ada que implementa um monitor para um *buffer*. O *buffer* se comporta de maneira muito parecida com o da Seção 13.3, no qual a sincronização é controlada com semáforos.

```
task Buf_Task is
   entry Deposit(Item : in Integer);
   entry Fetch(Item : out Integer);
end Buf_Task;

task body Buf_Task is
  Bufsize : constant Integer := 100;
  Buf   :  array (1..Bufsize) of Integer;
  Filled : Integer range 0..Bufsize := 0;
  Next_In,
  Next_Out : Integer range 1..Bufsize := 1;
begin
  loop
    select
      when Filled < Bufsize =>
         accept Deposit(Item : in Integer) do
           Buf(Next_In) := Item;
         end Deposit;
        Next_In := (Next_In mod Bufsize) + 1;
        Filled := Filled + 1;
    or
      when Filled > 0 =>
        accept Fetch(Item : out Integer) do
          Item := Buf(Next_Out);
        end Fetch;
        Next_Out := (Next_Out mod Bufsize) + 1;
        Filled := Filled - 1;
    end select;
   end loop;
end Buf_Task;
```

Nesse exemplo, as duas cláusulas **accept** são estendidas. Elas podem ser executadas concorrentemente com as tarefas que chamaram as cláusulas **accept** associadas.

As tarefas para um produtor e um consumidor que poderiam usar `Buf_Task` têm a seguinte forma:

```
task Producer;
task Consumer;
task body Producer is
  New_Value : Integer;
begin
```

```
  loop
    -- produzir New_Value --
    Buf_Task.Deposit(New_Value);
  end loop;
end Producer;

task body Consumer is
  Stored_Value : Integer;
begin
  loop
    Buf_Task.Fetch(Stored_Value);
    -- consume Stored_Value --
  end loop;
end Consumer;
```

13.6.4 Objetos protegidos

Como vimos, o acesso a dados compartilhados pode ser controlado envolvendo-se os dados em uma tarefa e permitindo-se o acesso apenas por meio de entradas de tarefas, as quais fornecem sincronização de competição implicitamente. Um problema desse método é a dificuldade de implementar o mecanismo de *rendezvous* de forma eficiente. Os objetos protegidos de Ada 95 oferecem um método alternativo de fornecer sincronização de competição que não envolve o mecanismo de *rendezvous*.

Um objeto protegido não é uma tarefa; é mais como um monitor, como descrito na Seção 13.4. Objetos protegidos podem ser acessados por subprogramas protegidos ou por entradas sintaticamente semelhantes às cláusulas **accept** nas tarefas.[3] Os subprogramas protegidos podem ser procedimentos protegidos, os quais fornecem acesso de leitura e escrita mutuamente exclusivo para os dados do objeto protegido, ou funções protegidas, as quais fornecem acesso concorrente somente leitura para esses dados. As entradas diferem dos subprogramas protegidos porque podem ter guardas.

Dentro do corpo de um procedimento protegido, a instância atual da unidade protegida envolvida é definida como uma variável; dentro do corpo de uma função protegida, a instância atual da unidade protegida envolvida é definida como uma constante, permitindo acessos concorrentes do tipo somente leitura.

Chamadas de entrada para um objeto protegido fornecem comunicação síncrona com uma ou mais tarefas por meio do mesmo objeto protegido. Essas chamadas de entrada permitem acesso similar àquele fornecido para os dados envoltos em uma tarefa.

O problema do *buffer* resolvido com uma tarefa na subseção anterior pode ser resolvido de maneira mais simples com um objeto protegido. Note que esse exemplo não inclui subprogramas protegidos.

```
protected Buffer is
  entry Deposit(Item : in Integer);
  entry Fetch(Item : out Integer);
```

[3] Entradas em corpos de objetos protegidos usam a palavra reservada `entry` em vez do `accept` utilizado no corpo das tarefas.

```
private
  Bufsize : constant Integer := 100;
  Buf   : array  (1..Bufsize) of Integer;
  Filled : Integer range 0..Bufsize := 0;
  Next_In,
  Next_Out : Integer range 1..Bufsize := 1;
  end Buffer;

protected body Buffer is
  entry Deposit(Item : in Integer)
     when Filled < Bufsize is
    begin
    Buf(Next_In) := Item;
    Next_In := (Next_In mod Bufsize) + 1;
    Filled := Filled + 1;
    end Deposit;
  entry Fetch(Item : out Integer) when Filled > 0 is
    begin Item := Buf(Next_Out);
    Next_Out := (Next_Out mod Bufsize) + 1;
    Filled := Filled - 1;
    end Fetch;
end Buffer;
```

13.6.5 Avaliação

Usar o modelo geral de passagem de mensagens de concorrência para construir monitores é como usar os pacotes Ada para suportar tipos de dados abstratos – ambos são ferramentas mais gerais do que o necessário. Objetos protegidos são uma maneira melhor de fornecer acesso sincronizado a dados compartilhados.

Na ausência de processadores distribuídos com memórias independentes, a escolha entre monitores e tarefas com passagem de mensagens como uma forma de implementar acesso sincronizado a dados compartilhados em um ambiente concorrente é uma questão de preferência pessoal. Entretanto, no caso de Ada, os objetos protegidos são claramente melhores que as tarefas para suportar acesso concorrente a dados compartilhados. O código não só é mais simples, mas também muito mais eficiente.

Para sistemas distribuídos, a passagem de mensagens é um modelo melhor para a concorrência, porque suporta o conceito de processos separados executando em paralelo em processadores separados.

13.7 LINHAS DE EXECUÇÃO EM JAVA

As unidades concorrentes em Java são métodos chamados `run`, cujo código pode estar em execução concorrente com outros métodos `run` (de outros objetos) e com o método `main`. O processo no qual o método `run` executa é chamado de linha de execução (**thread**). As linhas de execução em Java são tarefas leves, ou seja, todas executam no mesmo espaço de endereçamento. Elas são diferentes das tarefas de Ada, as quais são linhas de

execução pesadas (elas executam em seus próprios espaços de endereçamento).[4] Um resultado importante dessa diferença é que as linhas de execução de Java exigem muito menos sobrecarga que as tarefas de Ada.

Existem duas maneiras de definir uma classe com um método `run`. Uma delas é definir uma subclasse da classe predefinida `Thread` e sobrescrever seu método `run`. Entretanto, se a nova classe tem um pai natural necessário, defini-la como uma subclasse de `Thread` obviamente não funcionará. Nessas situações, definimos uma subclasse que herda de seu pai natural e implementa a interface `Runnable`. Esta fornece o protocolo do método `run`; então, qualquer classe que implemente `Runnable` deve definir `run`. Um objeto da classe que implementa `Runnable` é passado para o construtor de `Thread`. Portanto, essa estratégia ainda assim requer um objeto `Thread`, como veremos no exemplo da Seção 13.7.5.

Em Ada, as tarefas podem ser atrizes ou servidoras e podem se comunicar por meio de cláusulas **`accept`**. Todos os métodos `run` em Java são atores e não há nenhum mecanismo para eles se comunicarem, exceto o método `join` (consulte a Seção 13.7.1) e por meio de dados compartilhados.

As linhas de execução de Java são um tópico complexo – esta seção fornece apenas uma introdução aos aspectos mais simples, porém mais úteis.

13.7.1 A classe Thread

A classe `Thread` não é o pai natural de nenhuma outra. Ela fornece alguns serviços para suas subclasses, mas não está relacionada de nenhuma maneira natural com seus propósitos computacionais. `Thread` é a única classe disponível para a criação de programas Java concorrentes. Conforme já mencionado, a Seção 13.7.5 discutirá brevemente o uso da interface `Runnable`.

A classe `Thread` inclui cinco construtores e uma coleção de métodos e constantes. O método `run`, que descreve as ações da linha de execução, é sempre sobrescrito por subclasses de `Thread`. O método `start` de `Thread` inicia sua linha de execução como uma unidade concorrente, chamando seu método `run`.[5] A chamada a `start` é incomum, pois o controle retorna imediatamente para o chamador, o qual continua sua execução em paralelo com o método `run` recém-iniciado.

A seguir está um esqueleto de subclasse de `Thread` e um fragmento de código que cria um objeto da subclasse e inicia a execução do método `run` na nova linha de execução:

```
class MyThread extends Thread {
  public void run() { . . . }
}
. . .
Thread myTh = new MyThread();
myTh.start();
```

[4]Na verdade, embora as tarefas de Ada se comportem como se fossem pesadas, em alguns casos elas são implementadas como linhas de execução. Às vezes isso é feito com bibliotecas, como a IBM Rational Apex Native POSIX Threading Library.

[5]Chamar o método `run` diretamente nem sempre funciona, porque a inicialização algumas vezes necessária é incluída no método `start`.

Quando um programa aplicativo em Java começa a executar, uma linha de execução é criada (na qual o método **main** executará) e **main** é chamado. Logo, todos os programas Java são executados em linhas de execução.

Quando um programa tem múltiplas linhas de execução, um escalonador deve determinar qual ou quais linhas executarão em dado momento. Em muitos casos, existe apenas um processador disponível, então apenas uma linha de execução é realmente executada nesse momento. É difícil descrever com precisão como o escalonador Java funciona, porque as diferentes implementações (Solaris, Windows e assim por diante) não necessariamente escalonam as linhas de execução da mesma forma. Entretanto, o escalonador costuma fornecer fatias de tempo de tamanho igual a cada linha de execução pronta, de maneira similar a um algoritmo *round-robin*, supondo que todas essas linhas de execução têm a mesma prioridade. A Seção 13.7.2 descreve como diferentes prioridades podem ser dadas a diferentes linhas de execução.

A classe `Thread` fornece diversos métodos para controlar a execução das linhas de execução. O método `yield`, que não recebe parâmetros, é uma requisição da linha de execução que está em ação para voluntariamente desistir do processador.[6] A linha de execução é imediatamente colocada na fila de tarefas prontas, tornando-se apta a ser executada. O escalonador escolhe a linha de execução com a prioridade mais alta na fila de tarefas prontas. Se não existirem outras linhas de execução prontas com prioridade mais alta que aquela que desistiu do processador, ela também pode ser a próxima linha de execução a obtê-lo.

O método `sleep` tem um único parâmetro, isto é, o número inteiro de milissegundos durante os quais o chamador de `sleep` quer bloquear a linha de execução. Após o número especificado de milissegundos, a linha de execução será colocada na fila de tarefas prontas. Como não existe uma maneira de saber por quanto tempo uma linha de execução estará na fila de tarefas prontas antes de ser executada, o parâmetro para `sleep` é a mínima quantidade de tempo durante o qual a linha de execução *não* estará em execução. O método `sleep` pode lançar uma `InterruptedException`, a qual deve ser manipulada pelo método que chama `sleep`. Exceções são descritas em detalhes no Capítulo 14.

O método `join` é usado para forçar um método a postergar sua execução até que o método `run` de outra linha de execução tenha concluído sua execução. Ele é usado quando o processamento de um método não pode continuar até que o trabalho de outra linha de execução seja concluído. Por exemplo, poderíamos ter o seguinte método `run`:

```
public void run() {
  . . .
  Thread myTh = new Thread();
  myTh.start();
  // realiza parte da computação desta linha de execução
  myTh.join();  // Esperar por myTh ser completada
  // realiza o restante da computação desta linha de execução
}
```

O método `join` coloca a linha de execução que o chama no estado bloqueado, que pode ser finalizado apenas pelo término da linha de execução na qual `join` foi chamado. Se

[6] O método **yield** é definido como uma "sugestão" para o escalonador, que pode ou não segui-la (embora normalmente o faça).

essa linha de execução estiver bloqueada, existe a possibilidade de um impasse. Como forma de prevenção, `join` pode ser chamado com um parâmetro, ou seja, o tempo limite em milissegundos que a linha de execução chamadora esperará para que a linha de execução chamada seja concluída. Por exemplo, a seguinte chamada a `join` fará com que a linha de execução chamadora espere dois segundos para que `myTh` termine. Se ela não tiver concluído sua execução após dois segundos, a linha de execução chamadora será colocada novamente na fila de tarefas prontas, ou seja, continuará sua execução assim que for possível (de acordo com o escalonamento).

```
myTh.join(2000);
```

As versões iniciais de Java incluíam três outros métodos de `Thread`: `stop`, `suspend` e `resume`. Todos foram descontinuados por problemas de segurança. O método `stop` às vezes é sobrescrito com um método simples que destrói a linha de execução ao configurar sua variável de referência como **`null`**.

Normalmente, um método `run` termina sua execução ao alcançar o final de seu código. Entretanto, em muitos casos, as linhas de execução executam até que se diga a elas que devem terminar. Em relação a isso, existe a questão de como uma linha de execução pode determinar se deve continuar ou parar. O método `interrupt` é uma maneira de comunicar a uma linha de execução que ela deve parar. Esse método não para a linha; em vez disso, envia uma mensagem que apenas configura um bit no objeto da linha de execução. O bit é verificado com o método predicado, `isInterrupted`. Essa não é uma solução perfeita, porque a linha de execução que alguém tenta interromper pode estar dormindo ou esperando no momento em que o método `interrupt` é chamado, ou seja, ela não estará verificando se foi interrompida. Para essas situações, o método `interrupt` ainda lança uma exceção, `InterruptedException`, que também faz a linha de execução acordar (dos estados dormindo ou esperando). Então, uma linha de execução pode, periodicamente, verificar se foi interrompida e, caso tenha sido, se pode terminar. A linha de execução não pode perder a interrupção; se estava dormindo ou esperando quando a interrupção ocorreu, será acordada por ela. Na verdade, existem mais detalhes relacionados às ações e aos usos de `interrupt`, mas eles não são abordados aqui (Arnold et al., 2006).

13.7.2 Prioridades

As prioridades das linhas de execução não precisam ser as mesmas. A prioridade padrão de uma linha de execução inicialmente é a mesma da linha de execução que a criou. Se `main` cria uma linha de execução, sua prioridade padrão é a constante `NORM_PRIORITY`, que normalmente é 5. `Thread` define outras duas constantes de prioridade, `MAX_PRIORITY` e `MIN_PRIORITY`, cujos valores normalmente são 10 e 1, respectivamente.[7] A prioridade de uma linha de execução pode ser alterada com o método `setPriority`. A nova prioridade pode ser qualquer uma das constantes predefinidas ou qualquer outro número entre `MIN_PRIORITY` e `MAX_PRIORITY`. O método `getPriority` retorna a prioridade atual de uma linha de execução. As constantes de prioridade são definidas em `Thread`.

[7] O número de prioridades é dependente da implementação, então podem existir mais ou menos níveis que 10 em algumas implementações.

Quando existem linhas de execução com prioridades diferentes, o comportamento do escalonador é controlado por essas prioridades. Quando a linha de execução que está executando é bloqueada ou concluída, ou sua fatia de tempo expira, o escalonador escolhe a linha de execução com a prioridade mais alta na fila de tarefas prontas. Uma linha de execução com uma baixa prioridade executará apenas se uma de prioridade mais alta não estiver na fila de tarefas prontas quando surgir a oportunidade.

13.7.3 Semáforos

O pacote `java.util.concurrent.Semaphore` define a classe `Semaphore`. Objetos dessa classe implementam semáforos de contagem. Um semáforo de contagem tem um contador, mas nenhuma fila para armazenar descritores de linha de execução. A classe `Semaphore` define dois métodos, `acquire` e `release`, que correspondem às operações `wait` e `release` descritas na Seção 13.3.

O construtor básico de `Semaphore` recebe um parâmetro inteiro, o qual inicializa o contador do semáforo. Por exemplo, o seguinte poderia ser usado para inicializar os semáforos `fullspots` e `emptyspots` para o exemplo de *buffer* da Seção 13.3.2:

```
fullspots =  new  Semaphore(0);
emptyspots =  new  Semaphore(BUFLEN);
```

A operação `deposit` do método `producer` apareceria como segue:

```
emptyspots.acquire();
deposit(value);
fullspots.release();
```

Do mesmo modo, a operação `fetch` do método `consumer` apareceria como segue:

```
fullspots.acquire();
fetch(value);
emptyspots.release();
```

Os métodos `deposit` e `fetch` poderiam usar a estratégia utilizada na Seção 13.7.4 para fornecer a sincronização de competição exigida para os acessos ao *buffer*.

13.7.4 Sincronização de competição

Os métodos (mas não os construtores) de Java podem ser especificados para serem sincronizados (**`synchronized`**). Um método sincronizado chamado por meio de um objeto específico deve completar sua execução antes que qualquer outro método sincronizado possa executar em tal objeto. A sincronização de competição em um objeto é implementada ao especificarmos como sincronizados os métodos que acessam dados compartilhados. O mecanismo sincronizado é implementado como segue: todo objeto Java tem um cadeado. Métodos sincronizados devem adquirir o cadeado do objeto antes de poderem ser executados, impedindo outros métodos sincronizados de executarem no objeto durante esse tempo. Um método sincronizado libera o cadeado do objeto no qual

está executando quando conclui sua execução, mesmo que esse término seja devido a uma exceção. Considere a seguinte definição de um esqueleto de classe:

```
class ManageBuf {
  private int [100] buf;
  . . .
  public synchronized void deposit(int item) { . . . }
  public synchronized int fetch() { . . . }
  . . .
}
```

Os dois métodos definidos em `ManageBuf` são configurados como sincronizados, o que evita que um interfira no outro enquanto estiverem executando no mesmo objeto ao serem chamados por linhas de execução separadas.

Um objeto cujos métodos são todos sincronizados é efetivamente um monitor. Note que um objeto pode ter um ou mais métodos sincronizados, assim como um ou mais métodos não sincronizados. Um método não sincronizado pode executar em um objeto em dado momento, mesmo durante a execução de um método sincronizado.

Em alguns casos, o número de sentenças que tratam com a estrutura de dados compartilhada é significativamente menor que o número de outras sentenças na qual ela reside. Nesses casos, é melhor sincronizar o segmento de código que modifica a estrutura de dados compartilhada do que o método como um todo. Isso pode ser feito com a chamada *sentença sincronizada*, cuja forma geral é a seguinte:

```
synchronized (expressão){
 sentenças
}
```

A expressão nesse código deve ser avaliada para um objeto, e a sentença pode ser uma única sentença ou uma sentença composta. O objeto é cadeado durante a execução da sentença ou da sentença composta, então esta é executada exatamente como se estivesse no corpo de um método sincronizado.

Um objeto com métodos sincronizados definidos deve ter uma fila associada capaz de armazenar os métodos sincronizados que tentaram executar nele enquanto era operado por outro desses métodos. Na verdade, todo objeto tem uma fila chamada **fila de condição intrínseca**. Essas filas são fornecidas implicitamente. Quando um método sincronizado termina sua execução em um objeto, um método que estiver esperando em sua fila de condição intrínseca, se existir, é colocado na fila de tarefas prontas.

13.7.5 Sincronização de cooperação

A sincronização de cooperação em Java é implementada com os métodos `wait`, `notify` e `notifyAll`, definidos em `Object`, a classe raiz de todas as classes Java. Todas as classes, exceto `Object`, herdam esses métodos. Cada objeto tem uma lista de espera de todas as linhas de execução que chamaram `wait` nele. O método `notify` é chamado para informar a uma linha de execução em espera que um evento esperado já ocorreu. A

linha de execução específica acordada por `notify` não pode ser determinada, porque a máquina virtual Java (JVM) escolhe uma aleatoriamente a partir da lista de espera do objeto da linha de execução. Por isso, e pelo fato de que as linhas de execução em espera podem estar aguardando condições diferentes, o método `notifyAll` é chamado, em vez de `notify`. O método `notifyAll` acorda todas as linhas de execução da lista de espera do objeto, colocando-as na fila de tarefas prontas.

Os métodos `wait`, `notify`, e `notifyAll` podem ser chamados apenas de dentro de um método sincronizado, porque usam o cadeado colocado em um objeto por tal método. A chamada a `wait` é sempre colocada em um laço **`while`** controlado pela condição esperada pelo método. Esse laço é necessário porque o `notify` ou o `notifyAll` que acordou a linha de execução talvez tenha sido chamado devido a uma alteração em uma condição que não era a esperada pela linha. Se foi uma chamada a `notifyAll`, há uma chance ainda menor de que a condição esperada agora seja verdadeira. Devido ao uso de `notifyAll`, alguma outra linha de execução pode ter modificado a condição para falsa desde a última vez em que foi testada.

O método `wait` pode lançar `InterruptedException`, que é descendente de `Exception`. O tratamento de exceções de Java é discutido no Capítulo 14. Logo, qualquer código que chama `wait` também deve capturar `InterruptedException`. Supondo que a condição esperada seja chamada `theCondition`, a maneira convencional de usar `wait` é:

```
try  {
  while (!theCondition)
    wait();
  -- Faz o que for preciso após a condição theCondition tornar-se
verdadeira
}
catch(InterruptedException myProblem) { . . . }
```

O seguinte programa implementa uma fila circular para armazenar valores inteiros (**`int`**). Ele ilustra tanto a sincronização de cooperação quanto a de competição.

```
// Queue
// Esta classe implementa uma fila circular para armazenar
// valores inteiros. Ela inclui um construtor para alocar
// e inicializar a fila para um tamanho específico. Ela tem
// métodos sincronizados para inserir
// e remover valores da fila.

class Queue {
  private int [] que;
  private int nextIn,
              nextOut,
              filled,
              queSize;
```

```
  public Queue(int size) {
    que = new int [size];
    filled = 0;
    nextIn = 1;
    nextOut = 1;
    queSize = size;
  }  //** fim do construtor de Queue

  public synchronized void deposit (int item)
         throws InterruptedException {
    try {
      while (filled == queSize)
        wait();
      que [nextIn] = item;
      nextIn = (nextIn % queSize) + 1;
      filled++;
      notifyAll();
    }  //** fim da cláusula try
    catch(InterruptedException e) {}
  }  //** fim do método deposit

  public synchronized int fetch()
      throws InterruptedException {
    int item = 0;
    try {
      while (filled == 0)
        wait();
      item = que [nextOut];
      nextOut = (nextOut % queSize) + 1;
      filled--;
      notifyAll();
    }  //** fim da cláusula try
    catch(InterruptedException e) {}
    return item;
  }  //** fim do método fetch
}  //** fim da classe Queue
```

Note que o manipulador de exceções (**`catch`**) não faz nada aqui.

Classes para definir objetos produtores e consumidores que podem usar a classe `Queue` podem ser definidas como:

```
class Producer extends Thread {
  private Queue buffer;
  public Producer(Queue que) {
    buffer = que;
  }
```

```
  public void run() {
     int new_item;
     while (true) {
      //-- Cria um new_item
      buffer.deposit(new_item);
    }
  }
}

class Consumer extends Thread {
  private Queue buffer;
  public Consumer(Queue que) {
    buffer = que;
  }
  public void run() {
    int stored_item;
    while (true) {
      stored_item = buffer.fetch();
      //-- Consome o stored_item
    }
  }
}
```

O código a seguir cria um objeto `Queue` – e um objeto `Producer` e um `Consumer`, ambos anexados ao objeto `Queue` – e inicia sua execução:

```
Queue buff1 = new Queue(100);
Producer producer1 = new Producer(buff1);
Consumer consumer1 = new Consumer(buff1);
producer1.start();
consumer1.start();
```

Podemos definir `Producer` e/ou `Consumer` como implementações da interface `Runnable`, em vez de como subclasses de `Thread`. A única diferença é na primeira linha, que deveria aparecer agora como segue:

```
class Producer implements Runnable { . . . }
```

Para criar e executar um objeto de tal classe, ainda é necessário criar um objeto `Thread` conectado ao objeto. Isso é ilustrado no seguinte código:

```
Producer producer1 = new Producer(buff1);
Thread producerThread = new Thread(producer1);
producerThread.start();
```

Note que o objeto *buffer* é passado para o construtor de `Producer` e que o objeto `Producer` é passado para o construtor de `Thread`.

13.7.6 Sincronização sem bloqueio

Java contém algumas classes para controlar os acessos a certas variáveis que não incluem bloqueio nem espera. O pacote `java.util.concurrent.atomic` define classes que permitem certo acesso sincronizado sem bloqueio a variáveis dos tipos primitivos **`int`**, **`long`** e **`boolean`**, assim como referências e vetores. Por exemplo, a classe `AtomicInteger` define métodos de leitura e escrita, além de métodos para operações de adicionar, incrementar e decrementar. Todas essas operações são atômicas, ou seja, não podem ser interrompidas, de modo que não são exigidos cadeados para garantir a integridade dos valores das variáveis afetadas em um programa com múltiplas linhas de execução. Isso é sincronização granular — apenas uma variável. Atualmente, a maioria das máquinas têm instruções atômicas para essas operações em tipos **`int`** e **`long`**, de modo que frequentemente são fáceis de implementar (não são exigidos cadeados implícitos).

A vantagem da sincronização sem bloqueio é a eficiência: o acesso sem bloqueio que não ocorre durante a disputa não será mais lento e, normalmente, é mais rápido que o que usa **`synchronized`**. Um acesso sem bloqueio que ocorra durante a disputa definitivamente será mais rápido que um que use **`synchronized`**, porque este último exigirá a suspensão e o reescalonamento de linhas de execução.

13.7.7 Cadeados explícitos

Java 5.0 introduziu os cadeados explícitos como uma alternativa ao método e aos bloqueios **`synchronized`**, os quais fornecem cadeados implícitos. A interface `Lock` declara os métodos `lock`, `unlock` e `tryLock`. A classe predefinida `ReentrantLock` implementa a interface `Lock`. Para travar um bloqueio de código, o seguinte idioma pode ser usado:

```
Lock lock = new ReentrantLock();
. . .
Lock.lock();
try {
   // O código que acessa os dados compartilhados
} finally  {
  Lock.unlock();
}
```

Esse esqueleto de código cria um objeto `Lock` e chama o método `lock` nele. Então, usa um bloco **`try`** para envolver o código crítico. A chamada a `unlock` está em uma cláusula **`finally`** para garantir que o cadeado seja liberado, independentemente do que acontece no bloco **`try`**.

Existem pelo menos duas situações nas quais cadeados explícitos são usados em lugar de cadeados implícitos: primeiro, se a aplicação precisa tentar adquirir um cadeado, mas não pode esperar para sempre por ele, a interface `Lock` inclui um método, `tryLock`, que recebe um parâmetro de tempo limite. Se o cadeado não for adquirido dentro do tempo limite, a execução continua na sentença seguinte à chamada de

`tryLock`. Segundo, cadeados explícitos são usados quando não é conveniente ter o bloqueio de pares lock-unlock estruturado. Cadeados implícitos são sempre destravados no final da sentença composta na qual são travados. Cadeados explícitos podem ser destravados em qualquer lugar no código, independentemente da estrutura do programa.

Um perigo no uso de cadeados explícitos (que não existe no uso de cadeados implícitos) é omitir o destravamento. Os cadeados implícitos são destravados implicitamente no final do bloco travado. Contudo, os cadeados explícitos permanecem travados até serem explicitamente destravados, o que pode nunca acontecer.

Como mencionado, cada objeto tem uma fila de condição intrínseca, a qual armazena linhas de execução na espera de uma condição no objeto. Os métodos `wait`, `notify` e `notifyAll` são a API para uma fila de condição intrínseca. Como cada objeto pode ter apenas uma fila de condição, uma fila pode conter linhas execução esperando por diferentes condições. Por exemplo, a fila de nosso exemplo de *buffer* `Queue` pode ter linhas de execução esperando por uma de duas condições (`filled == queSize` ou `filled == 0`). É por isso que o *buffer* usa `notifyAll`. (Se usasse `notify`, apenas uma linha de execução seria acordada, e poderia ser uma que esperava uma condição diferente daquela que realmente se tornou verdadeira.) No entanto, é dispendioso usar `notifyAll`, pois ele acorda todas as linhas de execução que estão esperando um objeto e todas precisam verificar sua condição para determinar qual vai ser executada. Além disso, para verificar sua condição, elas precisam primeiro adquirir o cadeado no objeto.

Uma alternativa ao uso da fila de condição intrínseca é a interface `Condition`, que utiliza uma fila de condição associada a um objeto `Lock`. Ela também declara alternativas a `wait`, `notify` e `notifyAll`, chamadas `await`, `signal` e `signalAll`. Pode haver qualquer número de objetos `Condition` em um objeto `Lock`. Com `Condition`, `signal`, em vez de `signalAll`, pode ser usado; ele é mais fácil de entender e mais eficiente, em parte porque resulta em menos trocas de contexto.

13.7.8 Avaliação

O suporte de Java para concorrência é relativamente simples, mas eficaz. Todos os métodos `run` Java são tarefas atrizes e não há nenhum mecanismo de comunicação, exceto por meio de dados compartilhados, como o existente entre tarefas de Ada. Como são linhas de execução pesadas, as tarefas de Ada podem facilmente ser distribuídas para processadores distintos, em particular com memórias diferentes, que poderiam estar em computadores em lugares diversos. Tais tipos de sistemas não são possíveis com as linhas de execução de Java.

13.8 LINHAS DE EXECUÇÃO EM C#

Apesar de as linhas de execução em C# serem levemente baseadas nas de Java, existem diferenças significativas. A seguir, está um breve panorama das linhas de execução em C#.

13.8.1 Operações básicas de linhas de execução

Em vez de apenas métodos chamados `run`, como em Java, qualquer método C# pode executar em sua própria linha de execução. Quando são criadas linhas de execução C#, elas são associadas a uma instância de um representante predefinido, `ThreadStart`. Quando a execução de uma linha inicia, seu representante tem o endereço do método que deve executar. Assim, a execução de uma linha é controlada por meio de seu representante associado.

Uma linha de execução C# é criada por meio de um objeto `Thread`. O construtor de `Thread` deve receber uma instanciação de `ThreadStart`, à qual deve ser enviado o nome do método a ser executado na linha de execução. Por exemplo, podemos ter

```
public void MyRun1() { . . . }
. . .
Thread myThread = new Thread(new ThreadStart(MyRun1));
```

Nesse exemplo, criamos uma linha de execução chamada `myThread`, cujo representante aponta para o método `MyRun1`. Assim, quando a linha de execução começa, ela chama o método cujo endereço está em seu representante. Nesse exemplo, `myThread` é o representante e `MyRun1` é o método.

Assim como em Java, em C# existem duas categorias de linhas de execução: atrizes e servidoras. As linhas de execução atrizes não são chamadas especificamente; em vez disso, são iniciadas. Além disso, os métodos que executam não recebem parâmetros nem valores de retorno. Como em Java, criar uma linha de execução não inicia sua execução concorrente. Para linhas de execução atrizes, a execução deve ser solicitada por meio de um método da classe `Thread`, nesse caso chamado `Start`, como em

```
myThread.Start();
```

Como em Java, é possível fazer uma linha de execução esperar pelo término da execução de outra linha de execução antes de continuar, usando-se o método similarmente chamado de `Join`. Por exemplo, suponha que a linha de execução `A` tenha a seguinte chamada:

```
B.Join();
```

A linha de execução `A` será bloqueada até que a linha de execução `B` termine.

O método `Join` pode receber um parâmetro **`int`**, o qual especifica um limite de tempo, em milissegundos, que o chamador esperará para que a linha de execução termine.

Uma linha de execução pode ser suspensa por uma quantidade de tempo específica por meio do `Sleep`, um método estático público de `Thread`. O parâmetro para `Sleep` é um número inteiro de milissegundos. Diferentemente do respectivo método similar em Java, o `Sleep` de C# não lança nenhuma exceção, por isso não precisa ser chamado em um bloco **`try`**.

Uma linha de execução pode ser finalizada com o método `Abort`, apesar de ele não matá-la. Em vez disso, ele lança a exceção `ThreadAbortException`, que a linha de execução pode capturar. Quando ocorre a captura, a linha de execução normalmente libera qualquer recurso alocado e então termina (ao chegar ao final de seu código).

Uma linha de execução servidora só pode ser executada quando chamada por meio de seu representante. Essas linhas de execução são denominadas servidoras porque fornecem algum serviço quando requisitadas. As linhas de execução servidoras são mais interessantes que as atrizes, pois normalmente interagem com outras linhas de execução e frequentemente devem ter sua execução sincronizada com outras linhas.

Lembre-se, do Capítulo 9, de que qualquer método C# pode ser chamado indiretamente por meio de um representante. Tais chamadas podem ser feitas tratando-se o objeto representante como se fosse o nome do método. Na verdade, essa era uma abreviação para uma chamada a um método representante denominado `Invoke`. Assim, se o nome de um objeto representante fosse `chgfun1` e o método referenciado por ele recebesse um parâmetro **`int`**, poderíamos chamar esse método com uma das seguintes sentenças:

```
chgfun1(7);
chgfun1.Invoke(7);
```

Essas chamadas são síncronas, isto é, quando o método é chamado, o chamador é bloqueado até que o método finalize sua execução. C# também suporta chamadas assíncronas a métodos que executam em linhas de execução. Quando uma linha de execução é chamada assincronamente, ela e a linha de execução chamadora executam de forma concorrente, pois a chamadora não é bloqueada durante a execução da linha.

Uma linha de execução é chamada de forma assíncrona por meio do método de instância representante `BeginInvoke`, ao qual são enviados os parâmetros para o método do representante, junto com dois parâmetros adicionais, um de tipo `AsyncCallback` e outro de tipo **`object`**. `BeginInvoke` retorna um objeto que implementa a interface `IAsyncResult`. A classe representante também define o método de instância `EndInvoke`, o qual recebe um parâmetro de tipo `IAsyncResult` e retorna o mesmo tipo retornado pelo método encapsulado no objeto representante. Para chamar uma linha de execução de forma assíncrona, a chamamos com `BeginInvoke`. Por enquanto, usaremos **`null`** para os dois últimos parâmetros. Suponha que tenhamos a declaração de método e a definição de linha de execução a seguir:

```
public float MyMethod1(int  x);
. . .
Thread myThread = new Thread(new ThreadStart(MyMethod1));
```

A seguinte sentença chama `MyMethod` de forma assíncrona:

```
IAsyncResult result = myThread.BeginInvoke(10, null, null);
```

O valor de retorno da linha de execução chamada é obtido com o método `EndInvoke`, que recebe como parâmetro o objeto (de tipo `IAsyncResult`) retornado por `BeginInvoke`. `EndInvoke` retorna o valor de retorno da linha de execução chamada. Por exemplo, para obtermos o resultado **`float`** da chamada a `MyMethod`, usaríamos a seguinte sentença:

```
float  returnValue = EndInvoke(result);
```

Se o chamador precisa continuar algum trabalho enquanto a linha de execução chamada executa, ele deve ter um meio de determinar quando ela terminou. Para isso, a in-

terface IAsyncResult define a propriedade IsCompleted. Enquanto a linha de execução chamada executa, o chamador pode incluir código que ela pode executar em um laço **while** que dependa de IsCompleted. Por exemplo, poderíamos ter o seguinte:

```
IAsyncResult result = myThread.BeginInvoke(10, null, null);
while(!result.IsCompleted) {
  // Faz alguma computação
}
```

Essa é uma maneira eficiente de fazer algo na linha de execução chamadora enquanto se espera que a linha de execução chamada conclua seu trabalho. Contudo, se o volume de computação no laço **while** for relativamente pequeno, essa é uma maneira ineficiente de usar esse tempo (devido ao tempo exigido para testar IsCompleted). Uma alternativa é fornecer à linha de execução chamada um representante com o endereço de um método de callback e fazê-la chamar esse método quando tiver terminado. O representante é enviado como penúltimo parâmetro para BeginInvoke. Por exemplo, considere a seguinte chamada a BeginInvoke:

```
IAsyncResult result = myThread.BeginInvoke(10,
              new AsyncCallback(MyMethodComplete), null);
```

O método de callback é definido no chamador. Muitas vezes, tais métodos simplesmente definem uma variável booleana, como a denominada isDone, para **true**. Independentemente de quanto a linha de execução chamada demore, o método de callback será chamado apenas uma vez.

13.8.2 Sincronia de linhas de execução

Existem três maneiras diferentes de sincronizar linhas de execução C#: a classe Interlocked, a classe Monitor do espaço de nomes System.Threading e a sentença lock. Cada um desses mecanismos é projetado para uma necessidade específica. A classe Interlocked é usada quando as únicas operações que precisam ser sincronizadas são o incremento e o decremento de um inteiro. Essas operações são efetuadas atomicamente com os dois métodos de Interlocked, Increment e Decrement, ambos os quais recebem uma referência a um inteiro como parâmetro. Por exemplo, para incrementar um inteiro chamado counter em uma linha de execução, poderíamos usar

```
Interlocked.Increment(ref  counter);
```

A sentença lock é usada para marcar uma seção crítica de código em uma linha de execução. A sintaxe disso é:

```
lock(token) {
   // A seção crítica
}
```

Se o código a ser sincronizado está em um método de instância privado, o token é o objeto atual, de modo que **this** é usado como token para lock. Se o código a ser sin-

cronizado está em um método de instância público, uma instância de **`object`** é criada (na classe do método com o código a ser sincronizado) e uma referência a ela é usada como token para `lock`.

A classe `Monitor` define cinco métodos, `Enter`, `Wait`, `Pulse`, `PulseAll` e `Exit`, que podem ser usados para oferecer mais controle sobre a sincronização de linhas de execução. O método `Enter`, que recebe uma referência a um objeto como parâmetro, marca o início da sincronização da linha de execução desse objeto. O método `Wait` suspende a execução da linha e informa à Linguagem Comum de Tempo de Execução (CLR – Common Language Runtime) do .NET que ela quer continuar sua execução na próxima vez em que existir uma oportunidade. O método `Pulse`, que também recebe uma referência a um objeto como parâmetro, notifica uma linha de execução em espera que agora tem uma chance de executar novamente. `PulseAll` é similar ao `notifyAll` de Java. Linhas de execução que estavam esperando são executadas na ordem em que chamaram o método `Wait`. O método `Exit` termina a seção crítica da linha de execução.

A sentença é compilada em um monitor, de modo que `lock` é a abreviatura para um monitor. Um monitor é usado quando o controle adicional (por exemplo, com `Wait` e `PulseAll`) é necessário.

O .NET 4.0 adicionou uma coleção de estruturas de dados concorrentes genéricas, incluindo estruturas para filas, pilhas e bags.[8] Essas novas classes têm linhas de execução seguras, o que significa que podem ser usadas em um programa com múltiplas linhas de execução sem exigir que o programador se preocupe com sincronização de competição. O espaço de nomes `System.Collections.Concurrent` define essas classes, cujos nomes são `ConcurrentQueue<T>`, `ConcurrentStack<T>` e `ConcurrentBag<T>`. Assim, nosso programa de fila produtor-consumidor poderia ser escrito em C# com uma classe `ConcurrentQueue<T>` para a estrutura de dados e não haveria necessidade de programar a sincronização de competição para ele. Como essas coleções concorrentes são definidas no .NET, também estão disponíveis em todas as outras linguagens .NET.

13.8.3 Avaliação

As linhas de execução em C# fornecem uma ligeira melhoria em relação às de sua antecessora, Java. Por exemplo, qualquer método pode ser executado em sua própria linha de execução. Lembre-se de que, em Java, apenas métodos chamados `run` podem executar em suas próprias linhas de execução. Java suporta somente linhas de execução atrizes, mas C# suporta linhas de execução atrizes e servidoras. O término das linhas de execução também é mais limpo em C# (chamar um método `[Abort]` é mais elegante do que configurar o ponteiro da linha de execução como **`null`**). A sincronização da execução das linhas é mais sofisticada em C#, porque a linguagem tem diversos mecanismos, cada um para uma aplicação específica. As variáveis `Lock` de Java são semelhantes aos cadeados de C#, exceto que, em Java, um cadeado deve ser destravado explicitamente

[8] Bags são coleções desordenadas de objetos.

com uma chamada a `unlock`. Isso origina mais uma maneira de criar código errado. Linhas de execução em C#, como aquelas de Java, são leves; portanto, embora sejam mais eficientes, não podem ser tão versáteis quanto as tarefas de Ada. A disponibilidade das classes de coleção concorrentes é outra vantagem de C# em relação às outras linguagens não funcionais discutidas neste capítulo.

13.9 CONCORRÊNCIA EM LINGUAGENS FUNCIONAIS

Esta seção fornece um breve panorama do suporte para concorrência em várias linguagens de programação funcionais.

13.9.1 Multi-LISP

Multi-LISP (Halstead, 1985) é uma extensão de Scheme que permite ao programador especificar partes de um programa que podem ser executadas concorrentemente. Essas formas de concorrência são implícitas; o programador simplesmente informa ao compilador (ou interpretador) sobre algumas partes do programa que podem ser executadas concorrentemente.

Uma das maneiras pelas quais o programador pode informar ao sistema sobre uma possível concorrência é a construção `pcall`. Se uma chamada de função é incorporada a uma construção `pcall`, os parâmetros da função podem ser avaliados concorrentemente. Por exemplo, considere a seguinte construção `pcall`:

```
(pcall f a b c d)
```

A função é `f`, com parâmetros `a`, `b`, `c` e `d`. O efeito de `pcall` é fazer com que os parâmetros da função possam ser avaliados concorrentemente (qualquer um dos parâmetros ou todos eles podem ser expressões complicadas). Infelizmente, é responsabilidade do programador que esse processo seja usado com segurança, isto é, sem afetar a semântica da avaliação da função. Na verdade, isso é uma questão simples se a linguagem não permite efeitos colaterais, ou se o programador projetou a função de modo a não tê-los, ou, pelo menos, a tê-los de forma limitada. Contudo, Multi-LISP permite alguns efeitos colaterais. Se a função não foi escrita para evitá-los, pode ser difícil para o programador determinar se `pcall` pode ser usada com segurança.

A construção `future` de Multi-LISP é uma fonte de concorrência mais interessante e potencialmente mais produtiva. Como em `pcall`, uma chamada de função é empacotada em uma construção `future`. Tal função é avaliada em uma linha de execução separada, com a linha de execução pai continuando sua execução até que precise usar o valor de retorno da função. Se a função não tiver concluído sua execução quando seu resultado for necessário, a linha de execução pai esperará até que ela termine, antes de continuar.

Se uma função tem dois ou mais parâmetros, eles também podem ser empacotados em construções `future`, no caso em que suas avaliações poderão ser feitas concorrentemente em linhas de execução separadas.

Essas são as únicas adições a Scheme em Multi-LISP.

13.9.2 Concurrent ML

Concurrent ML (CML) é uma extensão de ML que inclui uma forma de linhas de execução e uma forma de passagem de mensagens síncrona para suportar concorrência. A linguagem é completamente descrita em Reppy (1999).

Uma linha de execução é criada em CML com a primitiva `spawn`, a qual recebe a função como parâmetro. Em muitos casos, a função é especificada como anônima. Assim que a linha de execução é criada, a função inicia sua execução nela. O valor de retorno da função é descartado. Os efeitos da função são a saída produzida ou por meio de comunicações com outras linhas de execução. A linha de execução pai (aquela que gerou a nova linha de execução) ou a filho (a nova) poderia terminar primeiro e isso não afetaria a execução da outra.

Canais fornecem os meios de comunicação entre linhas de execução. Um canal é criado com o construtor `channel`. Por exemplo, a sentença a seguir cria um canal de tipo arbitrário chamado `mychannel`:

```
let val mychannel = channel()
```

As duas principais operações (funções) em canais são para envio (`send`) e recebimento (`recv`) de mensagens. O tipo da mensagem é deduzido da operação de envio. Por exemplo, a chamada de função a seguir envia o valor inteiro `7` e, portanto, deduz-se que o tipo do canal é inteiro:

```
send(mychannel, 7)
```

A função `recv` nomeia o canal como seu parâmetro. Seu valor de retorno é aquele que recebeu.

Como as comunicações em CML são síncronas, uma mensagem é enviada e recebida somente se o remetente e o destinatário estiverem prontos. Se uma linha de execução envia uma mensagem em um canal e nenhuma outra linha de execução está pronta para receber nele, o remetente é bloqueado e espera que outra linha de execução execute um `recv` no canal. Do mesmo modo, se um `recv` é executado em um canal por uma linha de execução, mas nenhuma outra linha de execução enviou uma mensagem nesse canal, a que executou `recv` é bloqueada e espera por uma mensagem no canal.

Como os canais são tipos, as funções podem recebê-los como parâmetros.

Como acontecia na passagem de mensagens síncrona de Ada, uma questão na passagem de mensagens síncrona de CML é decidir qual mensagem escolher quando mais de um canal tiver recebido uma. E a mesma solução é usada: a construção de comando protegido **`do`**–**`od`**, que escolhe aleatoriamente entre as mensagens para diferentes canais.

O mecanismo de sincronização de CML é o *evento*. Uma explicação desse complicado mecanismo está além dos objetivos deste capítulo (e deste livro).

13.9.3 F#

Parte do suporte de F# para concorrência é baseada nas mesmas classes .NET utilizadas por C#, especificamente `System.Threading.Thread`. Por exemplo, suponha

que queiramos executar a função `myConMethod` em sua própria linha de execução. A função a seguir, quando chamada, criará a linha de execução e iniciará a execução da função nela:

```
let createThread() =
    let newThread = new Thread(myConMethod)
    newThread.Start()
```

Lembre-se de que, em C#, para criar uma instância de um representante predefinido, `ThreadStart`, é necessário enviar o nome do subprograma ao seu construtor e enviar a nova instância do representante como parâmetro para o construtor de `Thread`. Em F#, se uma função espera um representante como parâmetro, uma expressão ou função lambda pode ser enviada e o compilador se comportará como se você tivesse enviado o representante. Assim, no código acima, a função `myConMethod` é enviada como parâmetro para o construtor de `Thread`, mas o que é realmente enviado é uma nova instância de `ThreadStart` (para a qual `myConMethod` foi enviada).

A classe `Thread` define o método `Sleep`, o qual faz dormir, pelo número de milissegundos enviado a ele como parâmetro, a linha de execução a partir da qual foi chamado.

Dados imutáveis compartilhados não exigem sincronização entre as linhas de execução que os acessam. No entanto, se os dados compartilhados forem mutáveis, o que é possível em F#, será exigido um cadeado para evitar sua corrupção por parte das várias linhas de execução que tentam alterá-los. Uma variável mutável pode ser trancada enquanto uma função opera nela, a fim de fornecer acesso sincronizado ao objeto com a função `lock`. Essa função recebe dois parâmetros, o primeiro dos quais é a variável a ser alterada. O segundo é a expressão lambda que altera a variável.

Uma variável mutável alocada no monte é de tipo `ref`. Por exemplo, a seguinte declaração cria uma variável assim, chamada `sum`, com o valor inicial `0`:

```
let sum = ref 0
```

Uma variável tipo `ref` pode ser alterada em uma expressão lambda que utilize o operador de atribuição ALGOL/Pascal/Ada, `:=`. A variável `ref` deve ser prefixada com um ponto de exclamação (`!`) para receber seu valor. No seguinte, a variável mutável `sum` é trancada enquanto a expressão lambda adiciona o valor de `x` a ela:

```
lock(sum) (fun () -> sum := !sum + x)
```

Linhas de execução podem ser chamadas de forma assíncrona, exatamente como em C#, por meio dos mesmos subprogramas, `BeginInvoke` e `EndInvoke`, assim como a interface `IAsyncResult`, para facilitar a determinação da conclusão da execução da linha chamada de forma assíncrona.

Como mencionado, F# tem as coleções genéricas concorrentes de .NET disponíveis para seus programas. Isso pode economizar muito trabalho de programação ao se construir programas com múltiplas linhas de execução que precisam de uma estrutura de dados compartilhados na forma de uma fila, pilha ou bag.

13.10 CONCORRÊNCIA EM NÍVEL DE SENTENÇA

Nesta seção, abordamos brevemente o projeto de linguagem para concorrência em nível de sentença. Do ponto de vista do projeto de linguagem, o objetivo é fornecer um mecanismo para que o programador possa informar o compilador sobre as maneiras pelas quais ele pode mapear o programa em uma arquitetura multiprocessada.[9]

Nesta seção é discutida somente uma coleção de construções linguísticas de uma linguagem para concorrência em nível de sentença: Fortran de alto desempenho.

13.10.1 Fortran de alto desempenho

Fortran de alto desempenho (HPF – High-Performance Fortran) (HPF; ACM, 1993b) é uma coleção de extensões a Fortran 90 feitas para permitir que os programadores especifiquem informações ao compilador a fim de ajudá-lo a otimizar a execução de programas em computadores multiprocessados. HPF inclui tanto novas sentenças de especificação quanto subprogramas intrínsecos, predefinidos. Esta seção discute apenas algumas das sentenças de HPF.

As principais sentenças de especificação de HPF são usadas para descrever o número de processadores, a distribuição dos dados nas memórias desses processadores e o alinhamento dos dados com outros em termos de localização de memória. Essas sentenças aparecem como comentários especiais em um programa Fortran. Cada uma é introduzida pelo prefixo `!HPF$`, em que `!` é o caractere usado para iniciar linhas de comentários em Fortran 90. Esse prefixo as torna invisíveis para os compiladores Fortran 90, mas facilita seu reconhecimento pelos compiladores HPF.

A especificação `PROCESSORS` tem a seguinte forma

```
!HPF$ PROCESSORS procs (n)
```

Essa sentença é usada para especificar ao compilador o número de processadores que podem ser usados pelo código gerado para esse programa. Essa informação é usada em conjunto com outras especificações para dizer ao compilador como os dados devem ser distribuídos para as memórias associadas aos processadores.

As especificações `DISTRIBUTE` e `ALIGN` são usadas para fornecer informações ao compilador sobre máquinas que não compartilham memória — isto é, cada processador tem sua própria memória. A suposição é que um acesso feito por um processador em sua própria memória é mais rápido que um acesso à memória de outro processador.

A sentença `DISTRIBUTE` especifica quais dados serão distribuídos e o tipo de distribuição a ser usada. Sua forma é a seguinte:

```
!HPF$ DISTRIBUTE (tipo) ONTO procs :: lista_de_identificadores
```

Nessa sentença, o tipo pode ser bloco (`BLOCK`) ou cíclica (`CYCLIC`). A lista de identificadores são os nomes das variáveis de matrizes que serão distribuídas. Uma variável especificada para ser distribuída em bloco (`BLOCK`) é dividida em *n* grupos iguais; cada um consiste em coleções contíguas de elementos de matriz igualmente distribuídos nas

[9] Apesar de ALGOL 68 incluir um tipo semáforo desenvolvido para lidar com concorrência em nível de sentença, não discutimos essa aplicação de semáforos aqui.

memórias de todos os processadores. Por exemplo, se uma matriz com 500 elementos chamada `LIST` é distribuída em bloco em cinco processadores, seus primeiros 100 elementos serão armazenados na memória do primeiro processador, outros 100 na do segundo e assim por diante. Uma distribuição cíclica (`CYCLIC`) especifica quais elementos individuais da matriz são ciclicamente armazenados nas memórias dos processadores. Por exemplo, se `LIST` é ciclicamente distribuída, mais uma vez em cinco processadores, seu primeiro elemento será armazenado na memória do primeiro processador, o segundo elemento na memória do segundo processador e assim por diante.

A forma da sentença `ALIGN` é

`ALIGN` elemento_matriz1 `WITH` elemento_matriz2

`ALIGN` é usada para relacionar a distribuição de uma matriz com a de outra. Por exemplo,

```
ALIGN list1(index) WITH list2(index+1)
```

especifica que o elemento `index` de `list1` será armazenado na memória do mesmo processador que o elemento `index+1` de `list2` para todos os valores de `index`. As duas referências a matrizes em um `ALIGN` aparecem juntas em alguma sentença do programa. Colocá-las na mesma memória (o que significa no mesmo processador) garante que as referências a elas serão as mais próximas possíveis.

Considere o seguinte segmento de código de exemplo:

```
     REAL list_1 (1000), list_2 (1000)
      INTEGER list_3 (500), list_4 (501)
!HPF$ PROCESSORS proc (10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: list_1, list_2
!HPF$ ALIGN list_3 (index) WITH list_4 (index+1)
      . . .
      list_1 (index) = list_2 (index)
      list_3 (index) = list_4 (index+1)
```

Em cada execução dessas sentenças de atribuição, os dois elementos de matrizes referenciados serão armazenados na memória do mesmo processador.

As sentenças de especificação de HPF fornecem informações para o compilador que podem ou não serem usadas para otimizar o código que ele produz. O que o compilador faz depende de seu nível de sofisticação e da arquitetura específica da máquina alvo.

A sentença `FORALL` especifica uma sequência de sentenças de atribuição que podem ser executadas concorrentemente. Por exemplo,

```
FORALL (index = 1:1000)
  list_1(index) = list_2(index)
END FORALL
```

especifica a atribuição dos elementos de `list_2` aos elementos correspondentes de `list_1`. Contudo, as atribuições estão restritas à seguinte ordem: o lado direito de todas as 1.000 atribuições deve ser avaliado primeiro, antes que quaisquer atribuições ocorram. Isso permite a execução concorrente de todas as sentenças de atribuição. Além das sentenças de atribuição, sentenças `FORALL` podem aparecer no corpo de uma construção `FORALL`. A sentença `FORALL` casa bem com máquinas vetoriais, pois a mesma instrução

é aplicada a muitos valores de dados, normalmente em uma ou mais matrizes. A sentença HPF `FORALL` está incluída em Fortran 95 e nas versões subsequentes dessa linguagem.

Discutimos brevemente apenas uma pequena parte das capacidades de HPF. Entretanto, deve ser o suficiente para dar ao leitor uma ideia dos tipos de extensões de linguagem úteis para programar computadores com muitos processadores.

C# 4.0 (e as outras linguagens .NET) inclui dois métodos que se comportam de forma parecida com `FORALL`. São sentenças de controle de laço nas quais as iterações podem ser expandidas e os corpos executados concorrentemente. São elas: `Parallel.For` e `Parallel.ForEach`.

RESUMO

A execução concorrente pode ser nos níveis de instrução, sentença ou subprograma. Usamos *concorrência física* quando múltiplos processadores são usados para executar unidades concorrentes. Se as unidades concorrentes são executadas em um processador, usamos o termo *concorrência lógica*. O modelo conceitual subjacente de toda a concorrência pode ser referenciado como *concorrência lógica*.

A maioria dos computadores multiprocessados cai em uma de duas categorias amplas – SIMD ou MIMD. Os computadores MIMD podem ser distribuídos.

As linguagens que suportam concorrência em nível de subprograma devem fornecer duas capacidades fundamentais: acesso mutuamente exclusivo às estruturas de dados compartilhados (sincronização de competição) e cooperação entre as tarefas (sincronização de cooperação).

As tarefas podem estar em um de cinco estados diferentes: nova, pronta, executando, bloqueada ou morta.

Em vez de serem projetadas construções de linguagem para suportar concorrência, às vezes são usadas bibliotecas, como a OpenMP.

As questões de projeto para suporte de linguagem para concorrência são: como são fornecidas as sincronizações de competição e cooperação, como uma aplicação pode influenciar o escalonamento de tarefas, como e quando as tarefas começam e finalizam suas execuções e como e quando elas são criadas.

Um semáforo é uma estrutura de dados que consiste em um inteiro e em uma fila de descritores de tarefas. Podem ser usados para fornecer tanto sincronização de competição quanto de cooperação entre tarefas concorrentes. É comum que os semáforos sejam usados incorretamente, o que resulta em erros que não podem ser detectados pelo compilador, pelo ligador ou pelo sistema de tempo de execução.

Os monitores são abstrações de dados que fornecem uma maneira natural de conceder acesso mutuamente exclusivo a estruturas de dados compartilhadas entre tarefas. Eles são suportados por diversas linguagens de programação, entre elas Ada, Java e C#. A sincronização de cooperação em linguagens com monitores deve ser fornecida com alguma forma de semáforos.

O conceito subjacente ao modelo de concorrência de passagem de mensagens é o de que as tarefas enviam mensagens entre si para sincronizar sua execução.

Ada fornece construções complexas, mas efetivas, para concorrência, baseadas no modelo de passagem de mensagens. As tarefas de Ada são pesadas. Elas se comunicam umas com as outras pelo mecanismo de *rendezvous*, ou seja, pela passagem síncrona de

mensagens. Um *rendezvous* é a ação de uma tarefa aceitar uma mensagem enviada por outra. Ada conta com métodos simples e também complicados de controlar a ocorrência de *rendezvous* entre as tarefas.

Ada 95+ inclui capacidades adicionais para o suporte à concorrência, principalmente objetos protegidos. Essa linguagem suporta monitores de duas maneiras: com tarefas e objetos protegidos.

Java suporta unidades concorrentes leves de uma maneira relativamente simples, mas efetiva. Qualquer classe que herde de `Thread` ou implemente `Runnable` pode sobrescrever um método chamado `run` e ter o código desse método executado concorrentemente com outros métodos similares e com o programa principal. A sincronização de competição é especificada pela definição de métodos que acessam os dados compartilhados a serem implicitamente sincronizados. Pequenas seções de código também podem ser implicitamente sincronizadas. Uma classe cujos métodos são todos sincronizados é um monitor. A sincronização de cooperação é implementada com os métodos `wait`, `notify` e `notifyAll`. A classe `Thread` também fornece os métodos `sleep`, `yield`, `join` e `interrupt`.

Java tem suporte direto para semáforos de contagem por meio de sua classe `Semaphore` e de seus métodos `acquire` e `release`. Tinha também algumas classes para fornecer operações atômicas sem bloqueio, como adição, incremento e decremento de inteiros. Java fornece ainda cadeados explícitos com a interface `Lock` e a classe `ReentrantLock`, além de seus métodos `lock` e `unlock`. Além da sincronização implícita com **`synchronized`**, essa linguagem fornece sincronização implícita sem bloqueio de variáveis de tipo **`int`**, **`long`** e **`boolean`**, assim como referências e vetores. Nesses casos, são fornecidas operações atômicas de leitura, escrita, adição, incremento e decremento.

O suporte de C# para concorrência é baseado no de Java, mas é ligeiramente mais sofisticado. Qualquer método pode ser executado em uma linha de execução. São suportadas linhas de execução atrizes e servidoras. Todas as linhas de execução são controladas por meio de representantes associados. As linhas de execução servidoras podem ser chamadas de forma síncrona com `Invoke` ou de forma assíncrona com `BeginInvoke`. O endereço de um método de `callback` pode ser enviado para a linha de execução chamada. Três tipos de sincronização de linhas de execução são suportados com a classe `Interlocked`, a qual fornece operações atômicas de incremento e decremento, a classe `Monitor` e a sentença `lock`.

Todas as linguagens .NET têm o uso de estruturas de dados concorrentes genéricas para pilhas, filas e bags, para as quais a sincronização de competição é implícita.

Multi-LISP estende Scheme um pouco para permitir que o programador informe a implementação sobre partes do programa que podem ser executadas concorrentemente. Concurrent ML estende ML para suportar uma forma de linhas de execução e uma forma de passagem de mensagens síncrona entre essas linhas. Essa passagem de mensagens é feita com canais. Os programas F# têm acesso a todas as classes de suporte .NET para concorrência. O compartilhamento de dados mutáveis entre linhas de execução pode ter acesso sincronizado.

Fortran de Alto Desempenho (HPF) inclui sentenças para especificar como os dados serão distribuídos nas unidades de memória conectadas a múltiplos processadores. Também são incluídas sentenças para especificar coleções de sentenças que podem ser executadas concorrentemente.

NOTAS BIBLIOGRÁFICAS

A concorrência é discutida a fundo em Andrews e Schneider (1983), Holt et al. (1978) e Ben-Ari (1982).

O conceito de monitores é desenvolvido e sua implementação em Pascal Concorrente (Concurrent Pascal) é descrita por Brinch Hansen (1977).

O desenvolvimento inicial do modelo de passagem de mensagens de controle de unidades concorrentes é discutido por Hoare (1978) e por Brinch Hansen (1978). Uma discussão aprofundada sobre o desenvolvimento do modelo de tarefas de Ada pode ser encontrada em Ichbiah et al. (1979). Ada 95 é descrita em detalhes em ARM (1995). Fortran de Alto Desempenho é descrita em ACM (1993b).

QUESTÕES DE REVISÃO

1. Quais são os três níveis possíveis de concorrência em programas?
2. Descreva a arquitetura lógica de um computador SIMD.
3. Descreva a arquitetura lógica de um computador MIMD.
4. Qual nível de concorrência de programa é mais bem suportado por computadores SIMD?
5. Qual nível de concorrência de programa é mais bem suportado por computadores MIMD?
6. Descreva a arquitetura lógica de um processador vetorial.
7. Qual é a diferença entre a concorrência física e a lógica?
8. O que é uma linha de execução de controle em um programa?
9. Por que as corrotinas são chamadas de quasi-concorrentes?
10. O que é um programa com múltiplas linhas de execução?
11. Quais são as quatro razões para estudar o suporte linguístico para concorrência?
12. O que é uma tarefa pesada? O que é uma tarefa leve?
13. Defina *tarefa*, *sincronização*, *sincronização de competição* e *de cooperação*, *vivacidade*, *condição de corrida* e *impasse*.
14. Que tipo de tarefas não requer qualquer tipo de sincronização?
15. Descreva os cinco estados diferentes nos quais uma tarefa pode estar.
16. O que é um descritor de tarefa?
17. No contexto do suporte linguístico para concorrência, o que é uma guarda?
18. Qual é o propósito de uma fila de tarefas prontas?
19. Quais são as duas principais questões de projeto para o suporte linguístico para concorrência?

20. Descreva as ações das operações esperar (`wait`) e liberar (`release`) para semáforos.
21. O que é um semáforo binário? O que é um semáforo de contagem?
22. Quais são os principais problemas no uso de semáforos para fornecer sincronização?
23. Qual é a vantagem dos monitores em relação aos semáforos?
24. Cite três linguagens comuns nas quais os monitores podem ser implementados.
25. Defina *rendezvous*, *cláusula* **`accept`**, *cláusula* **`entry`**, *tarefa atriz*, *tarefa servidora*, *cláusula* **`accept`** *estendida*, *cláusula* **`accept`** *aberta*, *cláusula* **`accept`** *fechada* e *tarefa concluída*.
26. O que é mais geral, concorrência por monitores ou concorrência por passagem de mensagens?
27. As tarefas em Ada são criadas estática ou dinamicamente?
28. Para que serve uma cláusula **`accept`** estendida?
29. Como a sincronização de cooperação é fornecida para tarefas de Ada?
30. Qual é a vantagem dos objetos protegidos em Ada 95 em relação às tarefas para fornecer acesso a objetos de dados compartilhados?
31. Especificamente, que unidade de programa Java pode executar concorrentemente com o método principal em um programa aplicativo?
32. As linhas de execução Java são tarefas leves ou pesadas?
33. O que o método Java `sleep` faz?
34. O que o método Java `yield` faz?
35. O que o método Java `join` faz?
36. O que o método Java `interrupt` faz?
37. Quais são as duas construções Java que podem ser declaradas como sincronizadas?
38. Como a prioridade de uma linha de execução pode ser configurada em Java?
39. As linhas de execução Java podem ser atrizes, servidoras ou ambas?
40. Descreva as ações de três métodos Java usados para suportar sincronização de cooperação.
41. Que tipo de objeto Java é um monitor?
42. Explique por que Java inclui a interface `Runnable`.
43. Quais são os dois métodos usados com objetos Java `Semaphore`?
44. Qual é a vantagem da sincronização sem bloqueio em Java?
45. Quais são os métodos da classe Java `AtomicInteger` e qual é a finalidade dessa classe?

46. Como os cadeados explícitos são suportados em Java?
47. Que tipos de métodos podem ser executados em uma linha de execução C#?
48. As linhas de execução C# podem ser atrizes, servidoras ou ambas?
49. Quais são as duas maneiras de chamar uma linha de execução C# de forma síncrona?
50. Como uma linha de execução C# pode ser chamada de forma assíncrona?
51. Como é recuperado o valor retornado de uma linha de execução chamada de forma assíncrona em C#?
52. Qual é a diferença entre o método `Sleep` de C# e o `sleep` de Java?
53. O que exatamente faz o método `Abort` de C#?
54. Qual é o propósito da classe `Interlocked` de C#?
55. O que a sentença `lock` de C# faz?
56. Em que linguagem Multi-LISP foi baseada?
57. Qual é a semântica da construção `pcall` de Multi-LISP?
58. Como uma linha de execução é criada em CML?
59. Qual é o tipo de uma variável mutável alocada no monte de F#?
60. Por que as variáveis imutáveis de F# não exigem acesso sincronizado em um programa com múltiplas linhas de execução?
61. Qual é o objetivo das sentenças de especificação de HPF?
62. Qual é o propósito da sentença `FORALL` de HPF e Fortran?

PROBLEMAS

1. Explique por que a sincronização de competição não é um problema em um ambiente de programação que suporta corrotinas, mas não suporta concorrência.
2. Qual é a melhor ação que um sistema pode realizar quando um impasse é detectado?
3. Espera ociosa, ou espera ocupada, é um método no qual uma tarefa espera um evento ao continuamente verificar a ocorrência dele. Qual é o principal problema dessa estratégia?
4. No exemplo produtor-consumidor da Seção 13.3, suponha que incorretamente substituíssemos `release(access)` no processo consumidor por `wait(access)`. Qual seria o resultado desse erro na execução do sistema?
5. A partir de um livro de linguagem de programação de montagem (assembly) para um computador que usa um processador Pentium da Intel, determine quais instruções são fornecidas para suportar a construção de semáforos.
6. Suponha que duas tarefas, `A` e `B`, devem usar a variável compartilhada `Buf_Size`. A tarefa `A` adiciona 2 à `Buf_Size` e a tarefa `B` subtrai 1 dela. Suponha que tais ope-

rações aritméticas são efetuadas pelo seguinte processo de três passos: obter o valor atual, efetuar a aritmética e colocar o novo valor de volta. Na ausência de sincronização de competição, quais sequências de eventos são possíveis e quais são os valores resultantes dessas operações? Suponha que o valor inicial de `Buf_Size` é 6.

7. Compare o mecanismo de sincronização de competição de Java com o de Ada.
8. Compare o mecanismo de sincronização de cooperação de Java com o de Ada.
9. O que acontece se um procedimento monitor chama outro no mesmo monitor?
10. Explique a segurança relativa da sincronização de cooperação usando semáforos e as cláusulas **`when`** de Ada em tarefas.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Escreva uma tarefa em Ada para implementar semáforos gerais.
2. Escreva uma tarefa em Ada para gerenciar um *buffer* compartilhado como aquele de nosso exemplo, mas use a tarefa semáforo do Exercício de programação 1.
3. Defina semáforos em Ada e use-os para fornecer tanto sincronização de cooperação quanto de competição no exemplo do *buffer* compartilhado.
4. Escreva o Exercício de programação 3 usando Java.
5. Escreva o exemplo do *buffer* compartilhado apresentado neste capítulo em C#.
6. O problema do leitor-escritor pode ser enunciado como segue: uma posição de memória compartilhada pode ser concorrentemente lida por qualquer número de tarefas, mas quando uma tarefa precisa escrever na posição de memória compartilhada, ela deve ter acesso exclusivo. Escreva um programa em Java para o problema do leitor-escritor.
7. Escreva o Exercício de programação 6 usando Ada.
8. Escreva o Exercício de programação 6 usando C#.

Esta página foi deixada em branco intencionalmente.

14

Tratamento de exceções e tratamento de eventos

14.1 Introdução ao tratamento de exceções

14.2 Tratamento de exceções em C++

14.3 Tratamento de exceções em Java

14.4 Tratamento de exceções em Python e Ruby

14.5 Introdução ao tratamento de eventos

14.6 Tratamento de eventos com Java

14.7 Tratamento de eventos em C++

Este capítulo discute o suporte das linguagens de programação para duas partes relacionadas de muitos programas contemporâneos: o tratamento de exceções e o tratamento de eventos. Tanto exceções quanto eventos podem ocorrer em momentos não predeterminados, e ambos são mais bem tratados com construções e processos de linguagem especiais. Algumas dessas construções e processos – por exemplo, a propagação – são similares para o tratamento de exceções e o de eventos.

Primeiro, descrevemos os conceitos fundamentais do tratamento de exceções, incluindo exceções detectáveis por hardware e por software, tratadores de exceções e levantamento de exceções. Então, as questões de projeto para tratamento de exceções são apresentadas e discutidas, incluindo desde a vinculação até os tratadores de exceções, a continuação e os tratadores padrão. Essa seção é seguida por descrições e avaliações dos recursos de tratamento de exceções de duas linguagens de programação: C++ e Java. Em seguida, são apresentadas breves introduções ao tratamento de exceções em Python e em Ruby.

A parte final deste capítulo aborda tratamento de eventos. Primeiro apresentamos uma introdução aos conceitos básicos do tema, que é seguida por discussões sobre as estratégias de tratamento de eventos de Java e C#.

14.1 INTRODUÇÃO AO TRATAMENTO DE EXCEÇÕES

A maioria dos sistemas de hardware de computadores é capaz de detectar certas condições de erro em tempo de execução, como o transbordamento (*overflow*) de ponto flutuante. As primeiras linguagens de programação foram projetadas e implementadas de forma que o programa de usuário não pudesse nem detectar nem tentar tratar esses erros. Nessas linguagens, a ocorrência de tal tipo de erro simplesmente fazia o programa ser terminado e o controle ser transferido para o sistema operacional. A reação típica do sistema operacional para um erro em tempo de execução é mostrar uma mensagem de diagnóstico, a qual pode ser significativa e útil ou altamente enigmática. Após mostrar a mensagem, o programa é terminado.

No caso de operações de entrada e saída, entretanto, a situação é um pouco diferente. Por exemplo, uma sentença `Read` de Fortran pode interceptar erros de entrada e condições de término de arquivo, ambas detectadas pelo dispositivo de hardware de entrada. Nos dois casos, a sentença `Read` pode especificar o rótulo de alguma sentença no programa do usuário que trata dessa condição. No caso do término de arquivo, obviamente a condição nem sempre é considerada um erro. Na maioria das vezes, não passa de um sinal de que um tipo de processamento foi concluído e outro tipo precisa iniciar. A despeito da diferença óbvia entre o término de um arquivo e os eventos que são sempre erros, como um processo de entrada malsucedido, Fortran trata ambas as situações com o mesmo mecanismo. Considere a seguinte sentença `Read` em Fortran:

```
Read(Unit=5, Fmt=1000, Err=100, End=999) Weight
```

A cláusula `Err` especifica que o controle deve ser transferido para a sentença rotulada `100` se um erro ocorrer na operação de leitura. A cláusula `End` especifica que o controle deve ser transferido para a sentença rotulada `999` se a operação de leitura encontrar o final do arquivo. Então, Fortran usa desvios simples tanto para erros de entrada quanto para término de arquivo.

Existe uma categoria de erros sérios que não são detectáveis por hardware, mas que podem ser percebidos pelo código gerado pelo compilador. Por exemplo, erros de faixa de índice de matriz quase nunca são detectados por hardware,[1] mas causam erros sérios, frequentemente não percebidos até mais adiante na execução do programa.

A detecção de erros de faixas de índices algumas vezes é exigida pelo projeto da linguagem. Por exemplo, a especificação da linguagem Java exige que seus compiladores gerem código para verificar a corretude de cada uma das expressões de índices (eles não geram tal código quando, em tempo de compilação, é possível determinar que uma expressão de índice não pode ter um valor fora da faixa, por exemplo, se o índice é um literal). Em C, as faixas de índices não são verificadas porque acreditava-se (e ainda se acredita) que o custo da verificação não compensava o benefício de detectar tais erros. Em alguns compiladores para algumas linguagens, a verificação de faixa de índices pode ser selecionada (se não for ativada por padrão) ou desabilitada (se for ativada por padrão), conforme desejado no programa ou no comando que executa o compilador.

Os projetistas da maioria das linguagens contemporâneas incluíram mecanismos que permitem aos programas reagir de maneira padrão a certos erros em tempo de execução, assim como a outros eventos não usuais detectados por eles. Os programas também podem ser notificados quando certos eventos são detectados por hardware ou por software de sistema, de forma que também possam reagir a esses eventos.

Uma razão pela qual algumas linguagens não incluem tratamento de exceções é a complexidade que ele adiciona à linguagem.

14.1.1 Conceitos básicos

Consideramos os erros detectados por hardware, tanto erros de leitura de discos e condições não usuais quanto término de arquivos (o qual também é detectado por hardware), como exceções. Estendemos um pouco mais o conceito de exceção para incluir erros ou condições não usuais detectáveis por software (seja por um interpretador de software ou pelo próprio código do usuário). Dessa maneira, definimos uma **exceção** como qualquer evento não usual, errôneo ou não, detectável por hardware ou por software que possa exigir processamento especial.

O processamento especial que pode ser exigido quando uma exceção é detectada é chamado de **tratamento de exceção**. Ele é feito por uma unidade de código ou por um segmento chamado **tratador de exceção** (ou **manipulador de exceção**). Uma exceção é **levantada** (*raised*) quando seu evento associado ocorre. Em algumas linguagens baseadas em C, diz-se que as exceções são *lançadas*, em vez de levantadas.[2] Diferentes tipos de exceções exigem diferentes tratadores de exceção. A detecção de fim de arquivo praticamente sempre requer alguma ação específica do programa. Mas, claramente, tal ação também não seria apropriada para uma exceção de erro de faixa de índices de uma matriz. Em alguns casos, a única ação é a geração de uma mensagem de erro e o término organizado do programa.

[1] Nos anos 1970, havia alguns computadores que detectavam erros de faixa de índice no hardware.

[2] C++ foi a primeira linguagem baseada em C a incluir tratamento de exceções. A palavra *lançar* foi usada, em vez de *levantar*, porque a biblioteca C padrão contém uma função chamada `raise` (levantar).

A ausência de recursos de tratamento de exceções separados ou específicos em uma linguagem não impede o tratamento de exceções definidas pelo usuário, detectadas por software. Uma exceção detectada dentro de uma unidade de programa é geralmente manipulada pelo chamador da unidade. Um projeto possível é enviar um parâmetro auxiliar, usado como variável de estado. É atribuído um valor à variável de estado no subprograma chamado, de acordo com a corretude e/ou a normalidade dos resultados de suas computações. Imediatamente após o retorno da unidade chamada, o chamador testa a variável de estado. Se o valor indica a ocorrência de uma exceção, o tratador, que pode residir na unidade chamadora, pode ser ativado. Muitas das funções da biblioteca padrão de C usam uma variante dessa estratégia. Os valores de retorno são usados como indicadores de erro.

Outra possibilidade é passar um rótulo como parâmetro para o subprograma. É claro, essa estratégia é possível apenas em linguagens que permitem rótulos como parâmetros. Passar um rótulo permite que a unidade chamada retorne a um ponto diferente no chamador se uma exceção ocorrer. Como na primeira alternativa, o tratador é geralmente um segmento de código da unidade chamadora. Esse é um uso comum de rótulos como parâmetros em Fortran.

Uma terceira possibilidade é ter o tratador definido como um subprograma separado, cujo nome é passado como parâmetro para a unidade chamada. Nesse caso, o subprograma tratador é fornecido pelo chamador, mas a unidade chamada o aciona quando uma exceção é levantada. Um problema dessa estratégia é que ela exige o envio de um subprograma tratador a *todas* as chamadas a *todos* os subprogramas que recebem um subprograma tratador como parâmetro, independentemente de ele ser necessário ou não. Além disso, para lidar com diferentes tipos de exceções, diversas rotinas de tratamento precisariam ser passadas, complicando o código.

Se for desejável tratar uma exceção na unidade na qual ela é detectada, o tratador é incluído como um segmento de código nessa unidade.

Existem algumas vantagens definitivas em ter tratamento de exceções definidos em uma linguagem. Primeiro, sem tratamento de exceções, o código necessário para detectar condições de erro pode se tornar consideravelmente confuso. Por exemplo, suponha um subprograma que inclua expressões com 10 referências a elementos de uma matriz chamada `mat`, e qualquer uma delas pode ter um erro de índice fora de faixa. Além disso, suponha que a linguagem não exija verificação de faixas de índice. Sem essa verificação predefinida, cada uma dessas operações talvez precise ser precedida por código para detectar um possível erro de faixa de índice. Por exemplo, considere a seguinte referência a um elemento de `mat`, a qual tem 10 linhas e 20 colunas:

```
if (row >= 0 && row < 10 && col >= 0 && col < 20)
  sum += mat[row][col];
else
  System.out.println("Index range error on mat, row = " +
                     row + " col = " + col);
```

A presença de tratamento de exceções na linguagem permite que o compilador insira código de máquina para tais verificações antes de cada um dos acessos a elementos de matrizes, diminuindo e simplificando enormemente o programa-fonte.

Outra vantagem do suporte linguístico para o tratamento de exceções resulta da propagação de exceções. Ela permite que uma exceção levantada em uma unidade de programa seja tratada em outra unidade em seu ancestral dinâmico ou estático. Isso permite que um único tratador de exceções seja usado por qualquer número de unidades de programa. Esse reuso pode resultar em economias significativas em custo de desenvolvimento, tamanho e complexidade dos programas.

Uma linguagem que suporta o tratamento de exceções encoraja seus usuários a considerarem todos os eventos que podem ocorrer durante a execução dos programas e a analisarem como eles podem ser tratados. Essa estratégia é muito melhor do que não considerar tais possibilidades e simplesmente esperar que nada dê errado.

Por fim, existem programas nos quais lidar com situações não errôneas, mas não usuais, pode ser mais simples com o tratamento de exceções e nos quais a estrutura se tornaria muito confusa sem tal tratamento.

14.1.2 Questões de projeto

Agora, exploramos algumas das questões de projeto para um sistema de tratamento de exceções que faz parte de uma linguagem de programação. Tal sistema deve permitir tanto exceções predefinidas quanto exceções definidas pelo usuário, assim como tratadores de exceções. Note que exceções predefinidas são implicitamente levantadas, enquanto exceções definidas pelo usuário devem ser explicitamente levantadas pelo código de usuário. Considere o seguinte esqueleto de subprograma que inclui um mecanismo de tratamento de exceção para uma exceção implicitamente levantada:

```
void example() {
  . . .
  average = sum / total;
  . . .
  return;
/* tratadores de exceção */
  when zero_divide {
    average = 0;
    printf("Error-divisor (total) é zero\n");
  }
  . . .
}
```

A exceção de divisão por zero, implicitamente levantada, faz o controle ser transferido para o tratador apropriado, que é então executado.

A primeira questão de projeto para o tratamento de exceções é como a ocorrência de uma exceção é vinculada a um tratador. Essa questão ocorre em dois níveis. Em nível de unidade, existe a questão de como a mesma exceção levantada em diferentes pontos de uma unidade pode ser vinculada a diferentes tratadores dentro da unidade. Por exemplo, no subprograma anterior, existe um manipulador para uma exceção de divisão por zero que parece ser escrito para lidar com a ocorrência de uma divisão por zero em determinada sentença (aquela mostrada). Mas suponha que a função inclua diversas outras expressões com operadores de divisão. Para eles, esse tratador provavelmente não seria apropriado. Então, deve ser possível vincular as exceções que podem ser levantadas por

sentenças específicas a tratadores específicos, mesmo que essas exceções possam ser levantadas por muitas sentenças diferentes.

> **» *nota histórica***
>
> PL/I (ANSI, 1976) foi pioneira no conceito de permitir que os programas de usuário estivessem diretamente envolvidos com o tratamento de exceções. A linguagem permitia ao usuário escrever tratadores de exceções para uma longa lista de exceções definidas por ela. Além disso, PL/I introduziu o conceito de exceções definidas pelo usuário, que permitia aos programas criar exceções detectadas por software. Essas exceções usam os mesmos mecanismos empregados para exceções predefinidas.
>
> Desde que PL/I foi projetada, uma quantidade substancial de trabalho foi feita para projetar métodos alternativos de tratamento de exceções, e mecanismos de tratamento de exceção foram incluídos em uma longa lista de linguagens de programação subsequentes.

Em um nível mais alto, a questão da vinculação surge quando não existe um tratador de exceções local à unidade na qual a exceção é levantada. Nesse caso, o projetista da linguagem deve decidir se propaga a exceção para alguma outra unidade e, se esse for o caso, para onde. Como essa propagação ocorre e o quão longe ela vai têm um impacto importante na facilidade de escrita de tratadores de exceções. Por exemplo, se os tratadores devem ser locais, muitos deles devem ser escritos, o que complica tanto a escrita quanto a leitura do programa. Por outro lado, se as exceções são propagadas, um único tratador pode tratar a mesma exceção lançada em diversas unidades de programa, o que pode exigir que ele seja mais geral que o desejado.

Uma questão relacionada à vinculação de uma exceção a um tratador de exceção é se a informação acerca da exceção é disponibilizada para o tratador.

Após um tratador de exceção ser executado, o controle pode ser transferido para algum lugar no programa fora do código do tratador, ou a execução do programa pode simplesmente terminar. Chamamos isso de questão da continuação de controle após a execução do tratador ou simplesmente de **continuação**. O término é obviamente a escolha mais simples, e em muitas condições de exceções de erro, a melhor delas. Entretanto, em outras situações, particularmente naquelas associadas a eventos não usuais, mas não errôneos, a escolha de continuar a execução é a melhor. Esse projeto é chamado de **reinício**. Nesses casos, algumas convenções devem ser escolhidas para se determinar onde a execução deve continuar. Pode ser na sentença que levantou a exceção, na sentença após a que levantou a exceção ou em alguma outra unidade. A escolha de retornar para a sentença que levantou a exceção pode parecer adequada, mas, no caso de uma exceção de erro, ela é útil apenas se o tratador de alguma forma é capaz de modificar os valores ou as operações que causaram o levantamento. Caso contrário, a exceção será simplesmente levantada novamente. A modificação necessária para uma exceção de erro é geralmente muito difícil de prever. Mesmo quando possível, entretanto, pode não ser uma prática viável. Ela permite que o programa remova o sintoma de um problema sem remover a causa.

As duas questões de vinculação de exceções a manipuladores e continuação são ilustradas na Figura 14.1.

Quando o tratamento de exceções é incluído, a execução de um subprograma pode terminar de duas maneiras: quando sua execução estiver completa ou quando encontrar uma exceção.[3] Em algumas situações, é necessário completar alguma

[3]Evidentemente, mesmo que a linguagem não suporte tratamento de exceção, um subprograma pode terminar devido a um erro detectado pelo sistema.

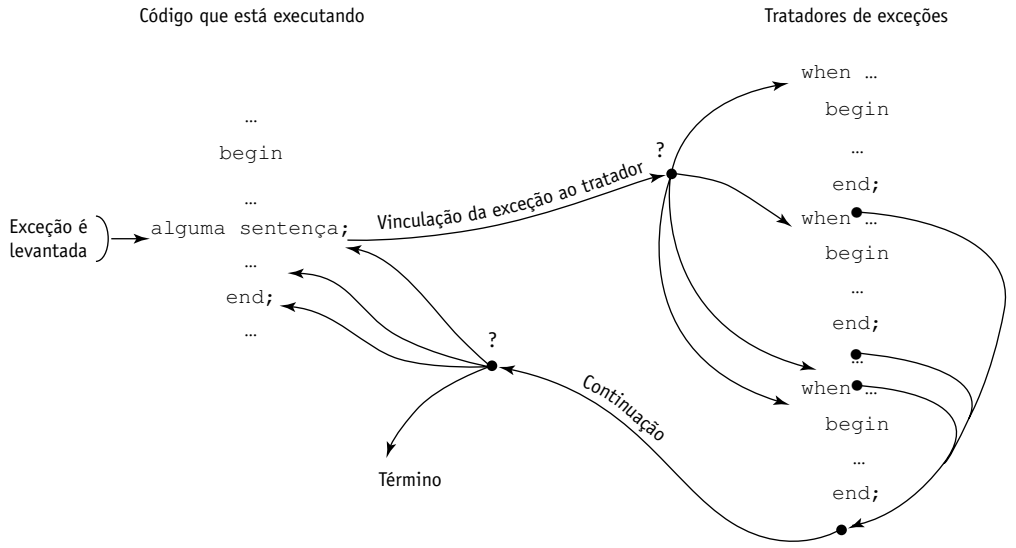


FIGURA 14.1
Fluxo de controle de tratamento de exceções.

computação, independentemente de como a execução do subprograma termina. A capacidade de especificar tal computação é chamada de *finalização*. A escolha entre suportar ou não a finalização é obviamente uma questão de projeto para o tratamento de exceções.

Outra questão de projeto é a seguinte: se é permitido aos usuários definir exceções, como elas são especificadas? A resposta usual é requerer que elas sejam declaradas nas partes de especificação das unidades de programa nas quais podem ser levantadas. O escopo de uma exceção declarada normalmente é o escopo da unidade de programa que contém a declaração.

No caso em que uma linguagem fornece exceções predefinidas surgem diversas outras questões de projeto. Por exemplo, o sistema de tempo de execução da linguagem deve fornecer tratadores padrão para as exceções predefinidas ou deve ser exigido que o usuário escreva tratadores para todas as exceções? Outra questão é se as exceções predefinidas podem ser levantadas explicitamente pelo programa do usuário. Esse uso pode ser conveniente se existirem situações detectáveis por software, nas quais o usuário gostaria de usar um tratador predefinido.

Outra questão é se erros detectáveis por hardware podem ser tratados por programas de usuário. Se não puderem, todas as exceções obviamente são detectáveis por software. Uma questão relacionada é se devem existir quaisquer exceções predefinidas. As exceções predefinidas são implicitamente levantadas por hardware ou por software de sistema.

- As questões de projeto para tratamento de exceções podem ser resumidas assim:
- Como e onde os tratadores de exceções são especificados e qual o seu escopo?
- Como uma ocorrência de uma exceção é vinculada a um tratador de exceção?

- A informação acerca de uma exceção pode ser passada para o tratador?
- Onde a execução continua, se é que continua, após um tratador de exceção completar sua execução? (Essa é a questão da continuação ou reinício.)
- Alguma forma de finalização é fornecida?
- Como as exceções definidas pelo usuário são especificadas?
- Se existem exceções predefinidas, devem existir tratadores de exceção padronizados para programas que não forneçam os seus próprios?
- As exceções predefinidas podem ser explicitamente levantadas?
- Os erros detectáveis por hardware são considerados exceções que devem ser tratadas?
- Existe alguma exceção predefinida?
- Estamos agora em uma posição que nos permite examinar os recursos de tratamento de exceções de várias linguagens de programação contemporâneas.

14.2 TRATAMENTO DE EXCEÇÕES EM C++

O tratamento de exceções de C++ foi aceito pelo comitê de padronização dessa linguagem em 1990 e foi utilizado nas suas implementações. O projeto é, em parte, baseado no tratamento de exceções de CLU, Ada e ML. Em C++, as exceções são definidas pelo usuário ou em bibliotecas e explicitamente levantadas.

14.2.1 Tratadores de exceção

C++ usa uma construção especial, introduzida com a palavra reservada **`try`**, para especificar o escopo dos tratadores de exceção. Uma construção **`try`** inclui uma sentença composta chamada **cláusula `try`** e uma lista de tratadores de exceções. A sentença composta define o escopo dos tratadores seguintes. A forma geral dessa construção é a seguinte:

```
try {
//** Código que pode levantar uma exceção
} catch (formal parameter) {
//** Corpo de um tratador
}
. . .
Catch (formal parameter) {
//** Corpo de um tratador
}
```

Cada função **`catch`** é um tratador de exceção. Uma função **`catch`** pode ter apenas um parâmetro formal, similar a um parâmetro formal em uma definição de função em C++, que pode ser reticências `(. . .)`. Um tratador com um parâmetro formal que é um sinal de reticências captura todas as exceções; ele é usado por qualquer exceção

levantada se nenhum tratador apropriado for encontrado. O parâmetro formal também pode ser um especificador de tipo, como **float**, como no protótipo da função. Nesse caso, o único objetivo do parâmetro formal é tornar o tratador unicamente identificável. Quando informações acerca da exceção precisam ser passadas para o tratador, o parâmetro formal inclui um nome de variável, usado para esse propósito. Como a classe do parâmetro pode ser qualquer uma definida pelo usuário, o parâmetro pode incluir quantos membros de dados forem necessários. A vinculação de exceções a tratadores é discutida na Seção 14.3.2.

Em C++, os tratadores de exceções podem incluir qualquer código C++.

14.2.2 Vinculação de exceções a tratadores

Exceções em C++ são levantadas apenas por meio da sentença explícita **throw**, cuja forma geral em EBNF é a seguinte:

```
throw [expressão];
```

Os colchetes aqui são metassímbolos usados para especificar que a expressão é opcional. Um **throw** sem um operando pode aparecer apenas em um tratador. Nesse caso, ele relança a exceção, então tratada em outro lugar.

O tipo da expressão **throw** seleciona o tratador específico, que deve ter um parâmetro de tipo formal que "case" com ele. Nesse caso, *casar* significa o seguinte: um tratador com um parâmetro formal do tipo `T`, **const** `T`, `T&` (uma referência a um objeto do tipo `T`) ou **const** `T&` casa um **throw** com uma expressão do tipo `T`. No caso em que `T` é uma classe, um tratador cujo parâmetro é do tipo `T` ou de qualquer classe ancestral de `T` casa com a expressão. Existem situações mais complicadas, nas quais uma expressão **throw** casa com um parâmetro formal, mas elas não são descritas aqui.

Uma exceção levantada em uma cláusula **try** causa o término imediato da execução do código em tal cláusula. A busca por um tratador que case com a expressão começa com os tratadores que seguem imediatamente a cláusula **try**. O processo de casamento é feito sequencialmente nos tratadores, até que um casamento seja encontrado. Isso significa que, se qualquer outro casamento preceder um tratador que casa perfeitamente, este não será usado. Logo, tratadores para exceções específicas são colocados no topo da lista, seguidos por tratadores mais genéricos. O último tratador geralmente é um com um parâmetro formal de reticências (. . .) que casa com qualquer exceção. Isso garantirá que todas as exceções sejam capturadas.

Se uma exceção é levantada em uma cláusula **try** e não existe um tratador que case associado a essa cláusula, a exceção é propagada. Se a cláusula **try** estiver aninhada dentro de outra, a exceção será propagada para os tratadores associados à cláusula mais externa. Se nenhuma das cláusulas **try** externas levar a um tratador que case, a exceção será propagada para o chamador da função na qual ela foi levantada. Se a chamada à função não estiver em uma cláusula **try**, a exceção será propagada para o chamador dessa função. Se nenhum tratador que case for encontrado no programa durante esse processo de propagação, o tratador padrão será chamado. Esse tratador é discutido em mais detalhes na Seção 14.2.4.

14.2.3 Continuação

Após um tratador ter completado sua execução, o controle flui para a primeira sentença que segue a construção **try** (a sentença imediatamente após o último tratador na sequência da qual ela é um elemento). Um tratador pode relançar uma exceção, usando um **throw** sem uma expressão, o que faz a exceção ser propagada.

14.2.4 Outras escolhas de projeto

Se comparado às outras questões de projeto resumidas na Seção 14.1.2, o tratamento de exceções de C++ é simples. Existem *apenas* exceções definidas pelo usuário e elas não são especificadas (apesar de poderem ser declaradas como novas classes). Existe um tratador de exceções padrão, `unexpected`, cuja única ação é terminar o programa. Esse tratador captura todas as exceções que não foram capturadas pelo programa. Ele pode ser substituído por um tratador definido pelo usuário. O tratador substituto deve ser uma função que retorna **void** e não recebe parâmetros. A função substituta é configurada com a atribuição de seu nome como `set_terminate`.

Uma função C++ pode listar os tipos das exceções (os tipos das expressões **throw**) que pode levantar. Isso é feito anexando-se a palavra reservada **throw**, seguida por uma lista desses tipos entre parênteses, ao cabeçalho da função. Por exemplo,

```
int fun() throw (int, char *) { . . . }
```

especifica que a função `fun` pode levantar exceções do tipo **int** e **char***, mas não de outros tipos. O propósito da cláusula **throw** é notificar aos usuários da função que ela pode levantar exceções. Essa cláusula pode ser vista como um contrato entre a função e suas chamadoras. Ela garante que nenhuma outra exceção será levantada na função. Se a função levantar algumas exceções não listadas, o programa será encerrado. Note que o compilador ignora cláusulas **throw**.

Se os tipos na cláusula **throw** são classes, a função pode levantar qualquer exceção que seja derivada das classes listadas. Se um cabeçalho de função tem uma cláusula **throw** e levanta uma exceção não listada nessa cláusula e não derivada de uma classe listada lá, o tratador padrão é chamado. Note que esse erro não pode ser detectado em tempo de compilação. A lista de tipos na lista pode ser vazia, o que significa que a função não levantará qualquer exceção. Se não existir uma especificação **throw** no cabeçalho, a função poderá levantar qualquer exceção. A lista não faz parte do tipo da função.

Se uma função sobrescreve outra que tem uma cláusula **throw**, a que sobrescreve não pode ter uma cláusula **throw** com mais exceções que a sobrescrita.

Apesar de C++ não ter exceções predefinidas, as bibliotecas padrão definem e lançam exceções, como `out_of_range`, que pode ser lançada por classes de biblioteca contêiner, e `overflow_error`, que pode ser lançada por funções de biblioteca matemáticas.

14.2.5 Um exemplo

O programa de exemplo a seguir ilustra alguns usos simples dos tratadores de exceção em C++. Ele computa e imprime uma distribuição de notas de entrada usando uma matriz de contadores. A entrada é uma sequência de notas terminada por um número negativo. Este levanta uma exceção `NegativeInputException`, pois as notas devem ser inteiros não negativos. Existem 10 categorias de notas (0 - 9, 10 - 19, . . . , 90 - 100). As notas propriamente ditas são usadas para computar índices em uma matriz de contadores, um para cada categoria. Notas de entrada inválidas são detectadas ao se capturar erros de indexação na matriz de contadores. Uma nota 100 é especial na computação da distribuição de notas, porque todas as categorias possuem 10 valores possíveis, exceto a última, que tem 11 (90, 91, . . . , 100). (O fato de existirem mais notas A possíveis do que B ou C é uma evidência conclusiva da generosidade dos professores.) A nota 100 também é tratada no mesmo tratador de exceções usado para dados de entrada inválidos.

Isso produz uma distribuição de notas de entrada por meio de um vetor de contadores para 10 categorias. Notas inválidas são detectadas pela verificação de índices inválidos usados no incremento do contador selecionado.

```
// Distribuição de Notas
//  Entrada: uma lista de valores inteiros que representam
//           notas, seguida de um número negativo
// Saída: uma distribuição de notas, como um percentual para
//          cada uma das categorias 0-9, 10-19, . . .,
//          90-100.
#include <iostream>
int main() {   //* Qualquer exceção pode ser levantada
  int new_grade,
      index,
      limit_1,
      limit_2,
      freq[10] = {0,0,0,0,0,0,0,0,0,0};
// A definição da exceção para lidar com o fim dos dados
class NegativeInputException {
  public:
   NegativeInputException() {  //* Construtor
    cout << "End of input data reached" << endl;
  }  //** fim do construtor
}  //** fim da classe NegativeInputException
  try {
     while (true) {
       cout << "Please input a grade" << endl;
       if ((cin >> new_grade) < 0)  //* Fim dos dados
         throw  NegativeInputException();
      index = new_grade / 10;
      {try {
        if (index > 9)
          throw  new_grade;
```

```
        freq[index]++;
        }  //* fim do try composto interno
      catch(int  grade) {  //* Tratador para erros de índices
        if (grade == 100)
          freq[9]++;
        else
          cout << "Error -- new grade: " << grade
               << " is out of range" << endl;
        }  //* fim de catch(int grade)
      }  //*  fim do bloco para o par try-catch interno
    }  //* fim do while (1)
  }  //* fim do bloco try externo
  catch(NegativeInputException& e) {  //**Tratador para
                                      //** entrada negativa
    cout << "Limits   Frequency" << endl;
    for (index = 0; index < 10; index++) {
      limit_1 = 10 * index;
      limit_2 = limit_1 + 9;
      if (index == 9)
        limit_2 = 100;
      cout << limit_1 << limit_2 << freq[index] << endl;
    }  //* fim do for (index = 0)
  }  //* fim do catch (NegativeInputException& e)
}  //* fim do main
```

Esse programa se propõe a ilustrar os mecanismos do tratamento de exceções em C++. Note que a exceção de faixa de índice é tratada por meio da sobrecarga do operador de indexação, o que pode então levantar a exceção, em vez da detecção direta do operador de indexação com a construção de seleção usada em nosso exemplo.

14.2.6 Avaliação

Uma deficiência do tratamento de exceções em C++ é que não existem exceções predefinidas detectáveis pelo hardware que possam ser tratadas pelo usuário. Exceções são conectadas aos tratadores por um tipo de parâmetro no qual o parâmetro formal pode ser omitido. O tipo do parâmetro formal de um tratador determina as condições nas quais ele é chamado, mas pode não ter relação com a natureza da exceção levantada. Logo, o uso de tipos predefinidos de exceções certamente não promove a legibilidade. É muito melhor estabelecer classes para exceções com nomes significativos em uma hierarquia significativa que possa ser usada para definir exceções. O parâmetro de exceção fornece uma maneira de passar informações acerca de uma exceção para o tratador dela.

14.3 TRATAMENTO DE EXCEÇÕES EM JAVA

No Capítulo 13, o programa de exemplo em Java incluía o uso de tratamento de exceções, mas não trazia muitas explicações sobre ele. Esta seção descreve os detalhes das capacidades de Java com relação ao tratamento de exceções.

Nessa linguagem, o tratamento de exceções é baseado no de C++, mas é projetado para estar mais alinhado com o paradigma de linguagem orientada a objetos. Além disso, Java inclui uma coleção de exceções predefinidas que podem ser implicitamente levantadas pelo seu sistema de tempo de execução.

14.3.1 Classes de exceções

Todas as exceções em Java são objetos de classes descendentes da classe `Throwable`. O sistema Java inclui duas classes de exceção predefinidas que são subclasses de `Throwable`: `Error` e `Exception`. A primeira e suas descendentes estão relacionadas a erros lançados pelo sistema de tempo de execução Java, como não ter mais memória no monte. Tais exceções nunca são lançadas por programas de usuário e nunca devem ser tratadas lá. Existem duas descendentes diretas de `Exception`, definidas pelo sistema: `RuntimeException` e `IOException`. Como seu nome indica, `IOException` é lançada quando ocorre um erro em uma operação de entrada ou de saída; ambas são descritas como métodos nas várias classes definidas no pacote `java.io`.

Existem classes predefinidas descendentes de `RuntimeException`. Na maioria dos casos, `RuntimeException` é lançada quando um programa de usuário causa um erro. Por exemplo, `ArrayIndexOutOfBoundsException`, definida em `java.util`, é uma exceção comumente lançada que descende de `RuntimeException`. Outra exceção frequentemente lançada que descende de `RuntimeException` é `NullPointerException`.

Os programas de usuário podem definir suas próprias classes de exceção. A convenção em Java é que as exceções definidas pelo usuário são subclasses de `Exception`.

14.3.2 Tratadores de exceção

Os tratadores de exceção de Java têm a mesma forma dos de C++, exceto que cada **`catch`** deve ter um parâmetro e a classe do parâmetro deve ser um descendente da classe predefinida `Throwable`.

A sintaxe da construção **`try`** em Java é exatamente igual à de C++, exceto pela cláusula **`finally`**, descrita na Seção 14.3.6.

14.3.3 Vinculação de exceções a tratadores

Lançar uma exceção é simples. Uma instância da classe de exceção é fornecida como o operando da sentença **`throw`**. Por exemplo, suponha que definíssemos uma exceção chamada `MyException` como

```
class MyException extends Exception {
  public MyException() {}
  public MyException(String message) {
    super (message);
  }
}
```

Essa exceção pode ser lançada com a seguinte sentença:

```
throw new MyException();
```

Um dos dois construtores que incluímos em nossa nova classe não tem parâmetros e o outro tem um parâmetro – um objeto da classe `String` enviado à superclasse (`Exception`), que o mostra. Portanto, nossa nova exceção pode ser lançada com

```
throw new MyException
       ("a message to specify the location of the error");
```

A vinculação de exceções a tratadores em Java é similar à de C++. Se uma exceção é lançada na sentença composta de uma construção **`try`**, ela é vinculada ao primeiro tratador (função **`catch`**) que imediatamente segue a cláusula **`try`** cujo parâmetro é a mesma classe do objeto lançado ou um ancestral dela. Se um tratador casado é encontrado, o **`throw`** é vinculado a ele e executado.

Exceções podem ser tratadas e depois relançadas ao se incluir uma sentença **`throw`** sem um operando no final do tratador. A nova exceção lançada não será tratada no mesmo **`try`** em que foi originalmente lançada; assim, repetições infinitas não são uma preocupação. Esse relançamento normalmente é feito quando alguma ação local é útil, mas é necessário um tratamento adicional por uma cláusula **`try`** que engloba o try em que a exceção ocorreu ou por uma cláusula **`try`** no chamador. Uma sentença **`throw`** em um tratador também pode lançar outra exceção que não aquela que transferiu o controle para ele.

Para garantir que exceções capazes de serem lançadas em uma cláusula **`try`** sejam sempre tratadas em um método, um tratador especial pode ser escrito para casar com todas as exceções derivadas de `Exception`; isso é feito simplesmente definindo-se o tratador com um parâmetro de tipo `Exception`, como em

```
catch (Exception genericObject) {
  . . .
}
```

Como um nome de classe sempre casa consigo mesmo ou com uma classe ancestral, qualquer classe derivada de `Exception` casa com ela. É claro, tal tratador de exceções deve ser sempre colocado no final da lista de tratadores, senão ele bloquearia qualquer tratador que o seguisse na construção **`try`** em que aparece. Isso ocorre porque a busca por um tratador que casa é sequencial e termina quando um casamento é encontrado.

14.3.4 Outras escolhas de projeto

Como parte de seus recursos de reflexão, o sistema de tempo de execução de Java armazena o nome da classe de cada um dos objetos no programa. O método `getClass` pode ser usado para obter um objeto que armazena o nome da classe, que por sua vez pode ser obtido com o método `getName`. Então, podemos buscar o nome da classe do parâmetro

real a partir da sentença **throw** que causou a execução do tratador. Para o tratador mostrado anteriormente, isso é feito com

```
genericObject.getClass().getName()
```

Além disso, a mensagem associada ao objeto parâmetro, criado pelo construtor, pode ser obtida com

```
genericObject.getMessage()
```

No caso de exceções definidas pelo usuário, o objeto lançado pode incluir qualquer número de campos de dados que sejam úteis no tratador.

A cláusula **throws** de Java tem aparência e posicionamento (em um programa) similares à da especificação **throw** de C++. Entretanto, sua semântica é um pouco diferente.

A aparição de um nome de classe de exceção na cláusula **throws** de um método Java especifica que essa classe ou qualquer uma de suas descendentes podem ser lançadas, mas não tratadas pelo método. Por exemplo, quando um método especifica que pode lançar `IOException`, isso significa que pode lançar um objeto de `IOException` ou um objeto de qualquer uma de suas classes descendentes, como `EOFException`, e que não trata a exceção que lança.

Exceções das classes `Error` e `RuntimeException` e suas descendentes são chamadas de **exceções não verificadas**. Todas as outras exceções são chamadas de **exceções verificadas**. Exceções não verificadas nunca são preocupações do compilador. Entretanto, ele garante que todas as exceções verificadas que um método pode lançar sejam listadas em sua cláusula **throws** ou tratadas no método. Note que verificar isso em tempo de compilação difere essa situação daquela de C++, em que isso é feito em tempo de execução. As exceções das classes `Error` e `RuntimeException` e suas descendentes não são verificadas porque qualquer método pode lançá-las. Um programa pode capturar exceções não verificadas, mas ele não é obrigado a isso.

Como no caso de C++, um método não pode declarar mais exceções em sua cláusula **throws** do que o método que sobrescreve, apesar de poder declarar menos. Então, se um método não tem uma cláusula **throws**, também não podem tê-la quaisquer métodos que o sobrescrevam. Um método pode lançar qualquer exceção listada em sua cláusula **throws**, com qualquer das classes descendentes dessas exceções.

Um método que não lança uma exceção específica diretamente, mas chama outro método que pode lançá-la, deve listar a exceção em sua cláusula **throws**. Essa é a razão pela qual o método `buildDist` (no exemplo da próxima subseção), que usa o método `readLine`, deve especificar `IOException` na cláusula **throws** de seu cabeçalho.

Um método que não inclui uma cláusula **throws** não pode propagar quaisquer exceções verificadas. Lembre-se de que, em C++, uma função sem uma cláusula **throws** pode lançar qualquer exceção.

Um método que chama outro capaz de listar uma exceção verificada em particular em sua cláusula **throws** tem três alternativas para lidar com essa exceção. Primeiro, pode capturar a exceção e tratá-la. Segundo, pode capturar a exceção e lançar uma exce-

ção listada em sua própria cláusula **throws**. Terceiro, pode declarar a exceção em sua cláusula **throws** e não tratá-la, o que propaga a exceção para uma cláusula **try** externa, se existir uma, ou para o chamador do método, se não existir.

Não existem tratadores de exceção padronizados e não é possível desabilitar exceções. Continuações em Java ocorrem exatamente como em C++.

14.3.5 Um exemplo

A seguir, temos o programa Java com as capacidades do programa C++ da Seção 14.2.5:

```
// Distribuição de Notas
//  Entrada: uma lista de valores inteiros que representam
//         notas, seguida de um número negativo
// Saída: uma distribuição de notas, como um percentual para
//         cada uma das categorias 0-9, 10-19, . . .,
//         90-100.
import java.io.*;
// A definição da exceção para lidar com o fim dos dados
class NegativeInputException extends Exception {
  public NegativeInputException() {
    System.out.println("End of input data reached");
  }  //** fim do construtor
}  //** fim da classe NegativeInputException

class GradeDist {
  int newGrade,
      index,
       limit_1,
       limit_2;
  int [] freq = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

void buildDist() throws  IOException {
  DataInputStream in =  new  DataInputStream(System.in);
  try {
    while (true) {
      System.out.println("Please input a grade");
      newGrade = Integer.parseInt(in.readLine());
      if (newGrade < 0)
        throw new NegativeInputException();
      index = newGrade / 10;
      try {
        freq[index]++;
      }  //** fim da cláusula try interna
      catch(ArrayIndexOutOfBoundsException e) {
        if (newGrade == 100)
          freq [9]++;
        else
          System.out.println("Error - new grade: " +
                     newGrade + " is out of range");
```

```
      }  //** fim do catch (ArrayIndex. . .
    }  //** fim do while (true) . . .
  }  //** fim do bloco try externo
  catch(NegativeInputException e) {
    System.out.println ("\nLimits    Frequency\n");
    for (index = 0; index < 10; index++) {
      limit_1 = 10 * index;
      limit_2 = limit_1 + 9;
      if (index == 9)
        limit_2 = 100;
      System.out.println("" + limit_1 + " - " +
                         limit_2 + "      " + freq [index]);
    }  //** fim do for (index = 0; . . .
  }  //** fim do catch (NegativeInputException . . .
}  //** fim do método buildDist
```

A exceção para uma entrada negativa, `NegativeInputException`, é definida no programa. Seu construtor mostra uma mensagem quando um objeto da classe é criado. Seu tratador produz a saída do método. `ArrayIndexOutOfBoundsException` é uma exceção não verificada, lançada pelo sistema de tempo de execução Java. Em ambos os casos, o tratador não inclui um nome de objeto em seus parâmetros. Em nenhum dos casos um nome serviria a algum propósito. Apesar de todos os tratadores receberem objetos como parâmetros, eles geralmente não são úteis.

14.3.6 A cláusula `finally`

Existem algumas situações nas quais um processo deve ser executado independentemente de uma cláusula **`try`** lançar uma exceção ou de a exceção ser tratada em um método. Um exemplo de tal situação é um arquivo que deve ser fechado. Outro ocorre se o método tiver algum recurso externo que deve ser liberado nele independentemente de como sua execução termine. A cláusula **`finally`** foi projetada necessidades como essas. Ela é colocada no final da lista de tratadores, imediatamente após uma construção **`try`** completa. Em geral, a construção **`try`** e sua cláusula **`finally`** aparecem como

```
try {
  . . .
}
catch (. . .) {
  . . .
}
. . . //** Mais tratadores
finally  {
  . . .
}
```

A semântica dessa construção é a seguinte: se a cláusula **`try`** não lançar exceções, a cláusula **`finally`** será executada antes que a execução continue após a construção **`try`**. Se a cláusula **`try`** lançar uma exceção e ela for capturada por um tratador que

a segue, a cláusula **finally** será executada após este completar sua execução. Se a cláusula **try** lançar uma exceção, mas ela não for capturada por um tratador que segue a construção **try**, a cláusula **finally** será executada antes de a exceção ser propagada.

Uma construção **try** sem tratadores de exceção pode ser seguida por uma cláusula **finally**. Isso faz sentido, é claro, apenas se a sentença composta tiver uma sentença **return**, **throw**, **break** ou **continue**. Seu propósito nesses casos é o mesmo de quando ela é usada com tratamento de exceções. Por exemplo, considere o seguinte:

```
try {
  for (index = 0; index < 100; index++) {
    . . .
    if (. . . ) {
      return;
    }  //** fim do if
    . . .
  }  //** fim do for
}  //** fim da cláusula try
finally  {
 . . .
}  //** fim da construção try
```

A cláusula **finally** será executada aqui independentemente de o **return** terminar o laço, ou de ele terminar normalmente.

14.3.7 Asserções

Na discussão sobre Plankalkül no Capítulo 2, mencionamos que ela incluía asserções, que também foram adicionadas a Java na versão 1.4. Para usá-las, é necessário habilitá-las, executando o programa com a opção `enableassertions` (ou `ea`), como em

```
java –enableassertions MyProgram
```

Existem duas formas possíveis da sentença **assert**:

```
assert condição;
assert condição : expressão;
```

No primeiro caso, a condição é testada quando a execução alcança o **assert**. Se for avaliada como verdadeira, nada acontecerá. Se for avaliada como falsa, a exceção `AssertionError` será lançada. No segundo caso, a ação é a mesma, exceto que o valor da expressão é passado para o construtor `AssertionError` como uma cadeia e se torna uma saída de depuração.

A sentença **assert** é usada para programação defensiva. Um programa pode ser escrito com muitas sentenças **assert**, o que garante que sua computação esteja pronta para produzir resultados corretos. Muitos programadores incluem tais verificações quando escrevem um programa, como uma ajuda para a depuração, mesmo que a linguagem usada não suporte asserções. Quando o programa está suficientemente testado, essas

verificações são removidas. A vantagem das sentenças **assert**, que têm o mesmo propósito, é poderem ser desabilitadas sem a necessidade de serem removidas do programa. Isso economiza o esforço de removê-las e permite o seu uso durante subsequentes manutenções de programa.

14.3.8 Avaliação

Os mecanismos de Java para tratamento de exceções são melhorias em relação à versão de C++, na qual eles são baseados.

Primeiro, um programa C++ pode lançar qualquer tipo definido no programa ou pelo sistema. Em Java, apenas objetos que são instâncias de `Throwable` ou de alguma classe que descende dela podem ser lançados. Isso separa os objetos que podem ser lançados de todos os outros objetos (e não objetos) que habitam um programa. Que importância pode ser dada a uma exceção que faz um valor inteiro ser lançado?

Segundo, uma unidade de programa C++ que não inclui uma cláusula **throw** pode lançar qualquer exceção, o que não diz nada para o leitor. Um método Java que não inclui uma cláusula **throws** não pode lançar uma exceção verificada que não trate. Logo, o leitor de um método Java sabe pelo seu cabeçalho quais exceções podem ser lançadas, mas não tratadas. Um compilador C++ ignora cláusulas **throw**, mas um compilador Java garante a listagem de todas as exceções que um método pode lançar em sua cláusula **throws**.

Terceiro, a cláusula **finally** é uma adição útil. Ela permite que certas ações de limpeza ocorram, independentemente de como uma sentença composta termine.

Por fim, o sistema de tempo de execução Java lança implicitamente uma variedade de exceções predefinidas, como para índices de matrizes fora de faixa e desreferenciamento de referências nulas, que podem ser tratadas por qualquer programa de usuário. Um programa C++ pode tratar apenas as exceções que lança explicitamente (ou que sejam lançadas por classes de biblioteca).

C# inclui construções de tratamento de exceções muito parecidas com as de Java, exceto que não tem uma cláusula **throws**.

14.4 TRATAMENTO DE EXCEÇÕES EM PYTHON E RUBY

Esta seção apresenta breves panoramas dos mecanismos de tratamento de exceção de Python e Ruby.

14.4.1 Python

Em Python, as exceções são objetos. A classe base de todas as classes de exceção é `BaseException`, da qual a classe `Exception` é derivada. `BaseException` fornece alguns serviços úteis para todas as classes de exceção, mas normalmente não se torna uma subclasse diretamente. Todas as classes de exceção predefinidas são derivadas de `Exception`, e as classes de exceção definidas pelo usuário também derivam dela. As subclasses predefinidas de `Exception` mais comumente usa-

das são: `ArithmeticError`, cujas principais subclasses são `OverflowError`, `ZeroDivisionError` e `FloatingPointError`; e `LookupError`, cujas subclasses principais são `IndexError` e `KeyError`.

As sentenças para lidar com exceções são semelhantes às de Java. A forma geral de uma construção **`try`** é:

```
try:
  O bloco try (a faixa de sentenças onde exceções vão ser observadas)
except Exception1:
  Tratador para Exception1
except Exception2:
  Tratador para Exception2
...
else:
  O bloco else (o que fazer quando nenhuma exceção é levantada)
finally:
  O bloco finally (o que deve ser feito independentemente do que aconteceu)
```

As cláusulas **`else`** e **`finally`** são opcionais.

Uma diferença entre os tratadores em Java e Python é que Python usa **`except`** para introduzi-los, em vez de **`catch`**. A cláusula **`else`** é executada se nenhuma exceção é levantada no bloco **`try`**. A cláusula **`finally`** tem a mesma semântica de sua equivalente em Java: se uma exceção é levantada no bloco **`try`**, mas não tratada por um tratador imediatamente a seguir, ela é propagada depois que o bloco **`finally`** é executado. Como um tratador manipula sua exceção nomeada e todas as subclasses dessa exceção, um que seja denominado `Exception` trata de todas as exceções predefinidas e definidas pelo usuário.

Uma exceção não tratada é propagada para construções **`try`** progressivamente maiores, em busca de um tratador apropriado. Se nenhum for encontrado, a exceção será propagada para o chamador de função, novamente procurando um tratador em uma construção **`try`** aninhada. Se nenhum tratador é encontrado nesse nível, os tratadores padrão são chamados; eles produzem uma mensagem de erro e um rastreamento de pilha e terminam o programa.

A sentença **`raise`** de Python é semelhante à sentença **`throw`** de Java e C++. O parâmetro de **`raise`** é o nome de classe da exceção a ser levantada. Por exemplo, poderíamos ter o seguinte:

```
raise IndexError
```

Essa sentença cria implicitamente uma instância da classe nomeada, `IndexError`.

Um tratador de exceções pode obter acesso ao objeto da exceção levantada pelo fornecimento de uma cláusula **`as`** e um nome de variável, como no seguinte:

```
except Exception as ex_obj:
```

Esse é um tratador universal, pois trata todas as exceções. O objeto exceção pode ser impresso com uma sentença **`print`** no tratador, a qual produz a mensagem do objeto.

Por exemplo, se a exceção fosse `ZeroDivisionError`, a mensagem seria divisão por zero (`division by zero`).

A sentença **`assert`** de Python fornece um mecanismo para tornar algum tratamento de exceção opcional. A forma geral de **`assert`** é a seguinte:

```
assert test, data
```

Nessa sentença, `test` é um flag ou uma expressão booleana, e `data` é o valor enviado ao construtor do objeto exceção a ser levantado. O significado dessa sentença, que opcionalmente levanta a exceção `AssertionError`, pode ser descrito com o seguinte código:

```
if __debug__:
  if not test:
    raise AssertionError(data)
```

`__debug__` é um flag predefinido, configurado como `True`, a menos que o flag `-0` seja usado no comando que executa o programa. Isso permite desabilitar todas as sentenças **`assert`** para determinada execução do programa. Se uma exceção `AssertionError` não é tratada pelo programa, assim como outras exceções não tratadas, ela termina o programa após usar o tratador padrão.

Python não tem equivalente para a cláusula **`throws`** de Java.

14.4.2 Ruby

Assim como as de Python, as exceções de Ruby são objetos, e essa linguagem tem uma grande coleção de classes de exceção predefinidas. Todas as exceções tratadas pelos programas aplicativos são objetos da classe `StandardError` ou de uma classe descendente dela. `StandardError` deriva de `Exception`, a qual fornece dois métodos úteis para todos os seus descendentes. São eles: `message`, que retorna a mensagem de erro legível para humanos, e `backtrace`, que retorna um rastreamento de pilha a partir do método onde a exceção foi levantada. Algumas das subclasses predefinidas de `StandardError` são: `ArgumentError`, `IndexError`, `IOError` e `ZeroDivisionError`.

Exceções são explicitamente levantadas com o método `raise`. Frequentemente, ele é chamado com um parâmetro do tipo cadeia. Nesse caso, levanta um novo objeto `RuntimeError`, com a cadeia como sua mensagem. Por exemplo, poderíamos ter o seguinte:

```
raise "bad parameter"  if  count == 0
```

`raise` também poderia ter dois parâmetros, o primeiro dos quais seria um objeto de uma classe de exceção. O método `exception` desse objeto é chamado e o objeto `Exception` retornado é levantado. Nesse caso, o segundo parâmetro seria a mensagem da cadeia a ser exibida. Por exemplo, poderíamos ter o seguinte:

```
raise TypeError, "Float parameter expected"
if not param.is_a? Float
```

Um tratador de exceções é especificado com uma cláusula **rescue**, a qual é anexada a uma sentença. Para se anexar um tratador de exceções a um segmento de código, o código é colocado em um bloco **begin–end**. A cláusula **rescue** é posicionada após o código do bloco. Em geral, isso aparece como no seguinte:

```
begin
  A sequência de sentenças no bloco
rescue
  O tratador
end
```

Um bloco **begin–end** pode incluir uma cláusula **else** e/ou uma cláusula **ensure**. A cláusula **else** é exatamente como a de Python. A cláusula **ensure** é exatamente como uma cláusula `finally`. Um método pode atuar como contêiner para tratamento de exceções no lugar de um bloco **begin–end**.

Em um claro distanciamento da maioria das outras linguagens, Ruby permite que um segmento de código que levantou uma exceção seja executado novamente depois que a exceção estiver tratada. Isso é especificado com uma sentença **retry** no final do tratador.

14.5 INTRODUÇÃO AO TRATAMENTO DE EVENTOS

O tratamento de eventos é similar ao tratamento de exceções. Em ambos os casos, os tratadores são implicitamente chamados pela ocorrência de algo, seja uma exceção ou um evento. Enquanto exceções podem ser levantadas explicitamente pelo código do usuário ou implicitamente por hardware ou por um interpretador de software, os eventos são criados por ações externas, como interações de usuário realizadas por meio de uma interface gráfica de usuário (GUI). Nesta seção, são apresentados os fundamentos do tratamento de eventos, que são menos complexos que os do tratamento de exceções.

Na programação convencional (não dirigida por eventos), o código de programa propriamente dito especifica a ordem na qual o código é executado, apesar de a ordem ser normalmente afetada pelos dados de entrada do programa. Na programação dirigida por eventos, partes do programa são executadas em momentos imprevisíveis, geralmente disparadas por interações de usuários com o programa em execução.

O tipo específico de tratamento de eventos discutido neste capítulo está relacionado às GUIs. Logo, a maioria dos eventos é causada por interações de usuários por meio de objetos ou de componentes gráficos, chamados de *widgets*. Os *widgets* mais comuns são os botões. A implementação de interações de usuário com componentes de GUI é a forma mais comum de tratamento de eventos.

Um **evento** é uma notificação de que algo ocorreu, como um clique de mouse em um botão gráfico. Em síntese, um evento é um objeto implicitamente criado pelo sistema de tempo de execução em resposta a uma ação de usuário, ao menos no contexto no qual o tratamento de eventos é discutido aqui.

Um **tratador de evento** é um segmento de código executado em resposta à aparição de um evento. Tratadores de evento permitem a um programa responder às ações do usuário.

Apesar de a programação dirigida por eventos ser empregada por muito tempo antes do aparecimento das GUIs, ela se tornou uma metodologia de programação bastante usada apenas em resposta à popularidade dessas interfaces. Como exemplo, considere as GUIs apresentadas aos usuários de navegadores Web. Atualmente, muitos documentos Web apresentados aos usuários são dinâmicos. Eles podem exibir um formulário de pedido ao usuário, que escolhe a mercadoria clicando em botões. As computações internas necessárias associadas a esses cliques são realizadas por tratadores de eventos.

Outro uso comum de tratadores de eventos é na verificação de erros simples e omissões nos elementos de um formulário, seja quando são mudados ou quando o formulário é submetido ao servidor Web para ser processado. O uso de tratamento de eventos no navegador para verificar a validade de dados de formulário economiza o tempo de envio dos dados para o servidor, onde, antes de serem enviados, esses dados têm sua corretude verificada por um programa ou *script* residente. Esse tipo de programação dirigida por eventos é feito por meio de uma linguagem de *scripting* do lado cliente, como JavaScript.

14.6 TRATAMENTO DE EVENTOS COM JAVA

Além das aplicações Web, aplicações que não são para a Web podem apresentar interfaces gráficas para os usuários. Nesta seção são discutidas as interfaces gráficas em aplicações Java.

A versão inicial de Java fornecia um suporte de certa forma primitivo para componentes de GUI. Na versão 1.2 da linguagem, lançada no final de 1998, foi adicionada uma nova coleção de componentes, coletivamente chamados de Swing.

14.6.1 Componentes de GUI de Java Swing[4]

A coleção Swing de classes e interfaces, definida em `javax.swing`, inclui componentes de GUI, ou *widgets*. Como nosso interesse aqui é o tratamento de eventos, não os componentes de GUI, discutiremos apenas dois tipos de *widgets*: caixas de texto e botões de rádio.

Uma caixa de texto é um objeto da classe `JTextField`. O construtor mais simples de `JTextField` recebe um parâmetro, o tamanho da caixa em caracteres. Por exemplo,

```
JTextField name = new JTextField(32);
```

O construtor de `JTextField` também pode receber uma cadeia literal como primeiro parâmetro opcional. O parâmetro do tipo cadeia, quando presente, é mostrado como o conteúdo inicial da caixa de texto.

Os botões de rádio são botões especiais colocados em um contêiner de grupos. Um grupo de botões é um objeto da classe `ButtonGroup`, cujo construtor não recebe parâmetros. Em um grupo de botões de rádio, apenas um pode ser pressionado de cada vez. Se qualquer botão no grupo for pressionado, o botão previamente pressionado implici-

[4]Nos últimos anos, o Swing está lentamente sendo substituído por um novo conjunto de ferramentas de GUI, chamado JAVAFX.

tamente passa a ser não pressionado. O construtor de `JRadioButton`, usado para criar os botões de rádio, recebe dois parâmetros: um rótulo e o estado inicial do botão de rádio (`true` ou `false`, para pressionado e não pressionado, respectivamente). Se um botão de rádio em um grupo está configurado inicialmente como pressionado, os outros botões do grupo são não pressionados por padrão. Após os botões de rádio serem criados, eles são colocados em seu grupo de botões com o método `add` do objeto grupo. Considere o seguinte exemplo:

```
ButtonGroup payment = new ButtonGroup();
JRadioButton box1 = new JRadioButton("Visa", true);
JRadioButton box2 = new JRadioButton("Master Charge");
JRadioButton box3 = new JRadioButton("Discover");
payment.add(box1);
payment.add(box2);
payment.add(box3);
```

Um objeto `JFrame` é um frame exibido como uma janela separada. A classe `JFrame` define os dados e métodos necessários para frames. Assim, uma classe que utiliza um frame pode ser uma subclasse de `JFrame`. Este tem várias camadas, chamadas **painéis**. Estamos interessados em apenas uma dessas camadas, o painel de conteúdo. Os componentes de uma GUI são colocados em um objeto `JPanel` (um painel), o qual é usado para organizar e definir o layout dos componentes. Um frame é criado e o painel que contém os componentes é adicionado ao painel de conteúdo desse frame.

Objetos gráficos predefinidos, como os componentes de GUI, são colocados diretamente em um painel. O seguinte código cria o objeto painel usado na discussão a seguir sobre componentes:

```
JPanel myPanel = new  JPanel();
```

Após os componentes terem sido criados com construtores, eles são colocados no painel com o método `add`, como em

```
myPanel.add(button1);
```

14.6.2 O modelo de eventos de Java

Quando um usuário interage com um componente de GUI, por exemplo, clicando em um botão, o componente cria um objeto evento e chama um tratador de evento por meio de um objeto denominado escutador de evento, passando o objeto evento. O tratador de evento fornece as ações associadas. Os componentes de GUI são geradores de evento. Em Java, os eventos são conectados a tratadores de evento por meio de **escutadores de evento**. Estes são conectados aos geradores de evento por meio do registro de escutador de evento, o qual é feito com um método da classe que implementa a interface escutadora, como descrito posteriormente. Apenas escutadores de eventos registrados para um evento especificado são notificados quando tal evento ocorre.

O método escutador que recebe a mensagem implementa um tratador de evento. Para fazer os métodos de tratamento de evento obedecerem a um protocolo padrão, é

usada uma interface. Esta prescreve protocolos de método padrão, mas não fornece implementações deles.

Uma classe que precisa implementar um tratador de evento deve implementar uma interface para o escutador desse tratador. Existem várias classes de eventos e interfaces escutadoras. Uma classe de eventos é `ItemEvent`, associada ao evento de clicar em uma caixa de verificação ou em um botão de rádio, ou selecionar um item de lista. A interface `ItemListener` inclui o protocolo de um método, `itemStateChanged`, que é o tratador de eventos de `ItemEvent`. Logo, para fornecer uma ação disparada pelo clique de um botão de rádio, a interface `ItemListener` deve ser implementada, o que requer a definição do método `itemStateChanged`.

Conforme mencionado, a conexão de um componente a um escutador de evento é feita com um método da classe que implementa a interface escutadora. Por exemplo, como `ItemEvent` é o nome da classe de objetos evento criados por ações de usuários em botões de rádio, o método `addItemListener` é usado para registrar um escutador para botões de rádio. O escutador para eventos de botão criados em um painel poderia ser implementado no painel ou em uma subclasse de `JPanel`. Então, para um botão de rádio chamado `button1` em um painel denominado `myPanel` que implementa o tratador de eventos para botões `ItemEvent`, poderíamos registrar o escutador com a sentença:

```
button1.addItemListener(this);
```

Cada método tratador de evento recebe um parâmetro de evento, o qual fornece informações acerca do evento. Classes de evento têm métodos para acessar tais informações. Por exemplo, quando chamado por um botão de rádio, o método `isSelected` retorna **`true`** ou **`false`**, dependendo se o botão estava ligado ou desligado (pressionado ou não pressionado), respectivamente.

Todas as classes relacionadas a eventos estão no pacote `java.awt.event`; então, elas normalmente são importadas por qualquer classe que use eventos.

A aplicação de exemplo a seguir, `RadioB`, ilustra o uso de eventos e de tratamento de eventos. Essa aplicação constrói botões de rádio que controlam o estilo da fonte do conteúdo de uma caixa de texto. Ela cria um objeto `Font` para cada um dos quatro estilos de fonte. Cada um deles tem um botão de rádio para permitir que o usuário selecione o estilo de fonte.

A finalidade desse exemplo é mostrar como eventos disparados pelos componentes de GUI podem ser tratados para mudar a saída do programa dinamicamente. Como nosso enfoque é restrito ao tratamento de eventos, algumas partes desse programa não são explicadas aqui.

```
/* RadioB.java
   Um exemplo para ilustrar o tratamento de eventos com botões
   de rádio interativos que controlam o estilo de fonte de uma caixa
de texto
  */
package radiob;
import java.awt.*;
```

```
import java.awt.event.*;
import javax.swing.*;
public class RadioB extends JPanel implements
        ItemListener {
    private JTextField text;
    private Font plainFont, boldFont, italicFont,
                 boldItalicFont;
    private JRadioButton plain, bold, italic, boldItalic;
    private ButtonGroup radioButtons;
// O método construtor é onde a tela é inicialmente
//  construída
    public  RadioB() {

// Cria a cadeia de texto de teste e define sua fonte
        text = new JTextField(
             "In what font style should I appear?", 25);
        text.setFont(plainFont);

// Cria os botões de rádio para as fontes e os adiciona
//  a um novo grupo de botões
        plain = new JRadioButton("Plain", true);
        bold = new JRadioButton("Bold");
        italic = new JRadioButton("Italic");
        boldItalic = new JRadioButton("Bold Italic");
        radioButtons = new ButtonGroup();
        radioButtons.add(plain);
        radioButtons.add(bold);
        radioButtons.add(italic);
        radioButtons.add(boldItalic);

        // Cria um painel e coloca nele o texto e os
        //  botões de rádio; então, adiciona o painel ao frame
        JPanel radioPanel =  new  JPanel();
        radioPanel.add(text);
        radioPanel.add(plain);
        radioPanel.add(bold);
        radioPanel.add(italic);
        radioPanel.add(boldItalic);
        add(radioPanel, BorderLayout.LINE_START);

// Registra os tratadores de evento
        plain.addItemListener(this);
        bold.addItemListener(this);
        italic.addItemListener(this);
        boldItalic.addItemListener(this);

// Cria as fontes
        plainFont = new Font("Serif", Font.PLAIN, 16);
```

FIGURA 14.2
Saída de `RadioB.java`.
Fonte: Captura de tela de applet de botão rádio Java.

```
        boldFont = new Font("Serif", Font.BOLD, 16);
        italicFont = new Font("Serif", Font.ITALIC, 16);
        boldItalicFont = new  Font("Serif", Font.BOLD +
                                   Font.ITALIC, 16);
    }  // Fim do construtor de RadioB
// O tratador de evento
    public void itemStateChanged (ItemEvent e) {

// Determina qual botão está ligado e configura a fonte de
//  acordo
        if (plain.isSelected())
           text.setFont(plainFont);
        else if (bold.isSelected())
           text.setFont(boldFont);
        else if (italic.isSelected())
           text.setFont(italicFont);
        else if (boldItalic.isSelected())
           text.setFont(boldItalicFont);
    } // Fim de itemStateChanged

// O método main
     public static void main(String[] args) {
// Cria o frame da janela
        JFrame myFrame = new JFrame(" Radio button
                                     example");

// Cria o painel de conteúdo e o configura como o frame
        JComponent myContentPane = new RadioB();
        myContentPane.setOpaque(true);
        myFrame.setContentPane(myContentPane);

// Exibe a janela.
        myFrame.pack();
        myFrame.setVisible(true);
    }
} // Fim de RadioB
```

A aplicação `RadioB` produz a tela mostrada na Figura 14.2.

14.7 TRATAMENTO DE EVENTOS EM C#

O tratamento de eventos em C# (e nas outras linguagens .NET) é semelhante ao de Java. .NET fornece duas estratégias para criar GUIs em aplicações, o Windows Forms original e o Windows Presentation Foundation, mais recente, sofisticado e complexo. Como estamos interessados apenas no tratamento de eventos, vamos usar o mais simples, Windows Forms, para discutirmos o assunto.

Com o Windows Forms, uma aplicação C# que constrói uma GUI é criada fazendo-se uma subclasse da classe predefinida `Form`, definida no espaço de nomes `System.Windows.Forms`. Essa classe fornece implicitamente uma janela para conter nossos componentes. Não há necessidade de construir frames ou painéis explicitamente.

Texto pode ser colocado em um objeto `Label` e botões de rádio são objetos da classe `RadioButton`. O tamanho de um objeto `Label` não é especificado explicitamente no construtor; em vez disso, pode ser especificado configurando-se o membro de dados `AutoSize` do objeto `Label` como **`true`**, o que ajusta o tamanho de acordo com o que é colocado nele.

Componentes podem ser dispostos em uma posição específica na janela, atribuindo-se um novo objeto `Point` à propriedade `Location` do componente. A classe `Point` é definida no espaço de nomes `System.Drawing`. O construtor de `Point` recebe dois parâmetros, os quais são as coordenadas do objeto em pixels. Por exemplo, `Point(100, 200)` é uma posição que está a `100` pixels da margem esquerda da janela e a `200` pixels da parte superior. O rótulo de um componente é definido atribuindo-se uma cadeia literal à propriedade `Text` dele. Após a criação de um componente, ele é adicionado à janela de formulário por meio de seu envio ao método `Add` da subclasse `Controls` do formulário. Portanto, o código a seguir cria um botão de rádio com o rótulo `Plain` na posição `(100, 300)` na janela de saída:

```
private RadioButton plain = new RadioButton();
plain.Location = new Point(100, 300);
plain.Text = "Plain";
Controls.Add(plain);
```

Todos os tratadores de evento C# têm o mesmo protocolo: o tipo de retorno é **`void`** e os dois parâmetros são de tipos **`object`** e `EventArgs`. Para uma situação simples, nenhum dos parâmetros precisa ser usado. Um método tratador de eventos pode ter qualquer nome. Um botão de rádio é testado para determinar se é clicado com sua propriedade booleana `Checked`. Considere o seguinte exemplo de esqueleto de tratador de eventos:

```
private void rb_CheckedChanged (object o, EventArgs e){
  if (plain.Checked) . . .
  . . .
}
```

Para se registrar um evento, deve ser criado um objeto `EventHandler`. O construtor dessa classe recebe o nome do método tratador. O novo objeto é adicionado ao representante predefinido do evento no objeto componente (usando-se o operador de atribuição +=). Por exemplo, quando um botão de rádio muda de desmarcado para marcado, o evento `CheckedChanged` é disparado e os tratadores registrados no representante associado, que é referenciado pelo nome do evento, são chamados. Se o tratador

de evento se chamasse `rb_CheckedChanged`, a seguinte sentença o registraria para o evento `CheckedChanged` no botão de rádio `plain`:

```
plain. CheckedChanged +=
       new EventHandler(rb_CheckedChanged);
```

A seguir está o exemplo `RadioB` da Seção 14.6 reescrito em C#. Mais uma vez, como nosso enfoque é o tratamento de eventos, não explicamos todos os detalhes do programa.

```
// RadioB.cs
// Um exemplo para ilustrar o tratamento de eventos com
//   botões de rádio interativos que controlam o estilo
//  de fonte de uma cadeia de texto

namespace RadioB {

    using System;
    using System.Drawing;
    using System.Windows.Forms;
  public class RadioB : Form {
    private Label text = new Label();
    private RadioButton plain = new RadioButton();
    private RadioButton bold = new RadioButton();
    private RadioButton italic = new RadioButton();
    private RadioButton boldItalic = new RadioButton();

    // Construtor de RadioB
    public RadioB() {

      // Inicializa os atributos do texto e dos
      //  botões de rádio
      text.AutoSize = true;
      text.Text = "In what font style should I appear?";
      plain.Location = new  Point(220,0);
      plain.Text = "Plain";
      plain.Checked = true;
      bold.Location = new Point(350, 0);
      bold.Text = "Bold";
      italic.Location = new Point(480, 0);
      italic.Text = "Italics";
      boldItalic.Location = new Point(610, 0);
      boldItalic.Text = "Bold/Italics";

      // Adiciona o texto e os botões de rádio ao formulário
      Controls.Add(text);
      Controls.Add(plain);
      Controls.Add(bold);
      Controls.Add(italic);
      Controls.Add(boldItalic);

      // Registra o tratador de evento para o botão de rádio
```

```
      plain.CheckedChanged +=
           new EventHandler(rb_CheckedChanged);
      bold.CheckedChanged +=
           new EventHandler(rb_CheckedChanged);
      italic.CheckedChanged +=
           new EventHandler(rb_CheckedChanged);
      boldItalic.CheckedChanged +=
           new EventHandler(rb_CheckedChanged);
    }

    // O método main é onde a execução começa
    static void Main() {
      Application.EnableVisualStyles();
      Application.SetCompatibleTextRenderingDefault (false);
      Application.Run(new RadioB());
    }
    // O tratador de evento

    private void rb_CheckedChanged (object o,
                                    EventArgs e) {

    // Determina qual botão está ligado e configura a fonte de
    //  acordo
     if (plain.Checked)
         text.Font =
              new Font( text.Font.Name, text.Font.Size,
                        FontStyle.Regular);
     if (bold.Checked)
         text.Font =
              new Font(text.Font.Name, text.Font.Size,
                       FontStyle.Bold);
     if (italic.Checked)
         text.Font =
              new Font(text.Font.Name, text.Font.Size,
                       FontStyle.Italic);
     if (boldItalic.Checked)
         text.Font =
              new  Font(text.Font.Name, text.Font.Size,
                        FontStyle.Italic ^ FontStyle.Bold);
    } // Fim de radioButton_CheckedChanged

  } // Fim de RadioB
}
```

A saída desse programa é exatamente como aquela mostrada na Figura 14.2.

RESUMO

C++ não inclui exceções predefinidas (exceto aquelas definidas na biblioteca padrão). Exceções C++ são objetos de um tipo primitivo, de uma classe predefinida ou de uma classe definida pelo usuário. As exceções são vinculadas aos tratadores ao conectarmos o tipo da expressão na sentença **`throw`** com o tipo do parâmetro formal do tratador. Todos os tratadores têm o mesmo nome – **`catch`**. A cláusula **`throw`** de um método lista os tipos de exceções que ele pode levantar.

As exceções Java são objetos cujos ancestrais devem ser rastreados para uma classe que descenda da classe `Throwable`. Existem duas categorias de exceções – verificadas e não verificadas. Exceções verificadas são preocupações do programa de usuário e do compilador. Exceções não verificadas podem ocorrer em qualquer lugar e geralmente são ignoradas por programas de usuário.

A cláusula Java **`throws`** de um método lista as exceções verificadas que ele pode lançar e que não trata. Ela deve incluir as exceções que métodos chamados podem lançar e propagá-las de volta para seu chamador.

A cláusula Java **`finally`** fornece um mecanismo para garantir que algum código será executado independentemente de como a execução de uma composição **`try`** terminar.

Java agora inclui uma sentença **`assert`**, que facilita a programação defensiva.

O tratamento de exceções de Python é semelhante ao de Java, embora adicione a cláusula **`else`** à construção **`try`**. Além disso, utiliza cláusulas **`except`**, em vez de cláusulas **`catch`**, para definir tratadores, e **`raise`**, em vez de **`throw`**. O acesso aos dados de um objeto exceção é obtido pela sua atribuição a uma variável com uma cláusula **`as`**. A sentença **`assert`** de Python é uma **`raise`** condicional.

O tratamento de exceções em Ruby é semelhante ao de Python. Toda classe de exceção tem dois métodos: `message` e `backtrace`. As exceções frequentemente são levantadas com uma sentença **`raise`**, com um único parâmetro de cadeia. Isso cria um objeto `RuntimeError` com a cadeia como sua mensagem. A sentença **`raise`** pode se tornar condicional pela adição de uma expressão condicional a ela. O escopo dos tratadores de exceção normalmente é especificado com um bloco **`begin-end`**. Tratadores são definidos em cláusulas **`rescue`**. Um bloco **`begin-end`** pode incluir uma cláusula **`else`** e uma cláusula **`ensure`**, que é como a cláusula **`finally`** de Python e Java.

Um evento é uma notificação de que ocorreu algo que precisa de processamento especial. Os eventos são criados por interações dos usuários com um programa, por meio de uma interface gráfica. Os tratadores de evento em Java são chamados por meio de escutadores de eventos. Um escutador de eventos deve ser registrado para um evento caso deva ser notificado de sua ocorrência. Duas das interfaces escutadoras de evento mais comumente usadas são `actionPerformed` e `itemStateChanged`.

Windows Forms é a estratégia original para se construir componentes de GUI e tratar eventos em linguagens .NET. Uma aplicação em C# constrói uma GUI com essa estratégia, fazendo uma subclasse da classe `Form`. Todos os tratadores de evento .NET usam o mesmo protocolo. Os tratadores de evento são registrados por meio da criação de um objeto **`EventHandler`** e de sua atribuição ao representante predefinido associado ao objeto de GUI que pode disparar o evento.

NOTAS BIBLIOGRÁFICAS

Um dos artigos mais importantes sobre tratamento de exceções que não está ligado a uma linguagem de programação específica é o trabalho de Goodenough (1975). Os problemas do projeto de PL/I para tratamento de exceções são abordados em MacLaren (1977). O tratamento de exceções em C++ é descrito por Stroustrup (1997). O tratamento de exceções em Java é descrito por Campione et al. (2001).

QUESTÕES DE REVISÃO

1. Defina *exceção*, *tratador de exceção*, *levantar uma exceção*, *continuação*, *finalização* e *exceção predefinida*.
2. Quais são as duas alternativas para projetar continuações?
3. Quais são as vantagens de ter suporte predefinido para tratamento de exceções em uma linguagem?
4. Quais são as questões de projeto relativas ao tratamento de exceções?
5. O que significa para uma exceção ser vinculada a um tratador de exceção?
6. Qual é o nome de todos os tratadores de exceção em C++?
7. Como as exceções podem ser explicitamente levantadas em C++?
8. Como as exceções são vinculadas aos tratadores em C++?
9. Como um tratador de exceção pode ser escrito em C++ de forma que trate qualquer exceção?
10. Para onde vai o controle de execução quando um tratador de exceção C++ completa sua execução?
11. C++ inclui exceções predefinidas?
12. Por que o levantamento de uma exceção em C++ não é chamado de raise?
13. Qual é a classe raiz de todas as classes de exceção em Java?
14. Qual é a classe pai da maioria das classes de exceção definidas pelo usuário em Java?
15. Como um tratador de exceção pode ser escrito em Java de forma que trate qualquer exceção?
16. Quais são as diferenças entre uma especificação **`throw`** em C++ e uma cláusula **`throws`** em Java?
17. Qual é a diferença entre exceções verificadas e não verificadas em Java?
18. Como um tratador de exceção pode ser escrito em Java de forma que trate qualquer exceção?
19. Qual é o propósito da cláusula **`finally`** em Java?

20. Que vantagem as asserções definidas pela linguagem possuem sobre construções de seleção simples (**`if`**–`write`)?
21. Explique o que faz um bloco **`else`** em Python.
22. Qual é a finalidade de uma cláusula **`as`** em Python?
23. Explique o que faz uma sentença **`assert`** em Python.
24. O que faz o método `message` da classe `StandardError` de Ruby?
25. O que acontece exatamente quando uma sentença **`raise`** com um parâmetro de cadeia é executado?
26. O que faz exatamente uma cláusula **`ensure`** em Ruby?
27. De que maneiras o tratamento de exceções e o tratamento de eventos são relacionados?
28. Defina evento e tratador de evento.
29. O que é programação dirigida por eventos?
30. Qual é o propósito de um `JFrame` Java?
31. Qual é o propósito de um `JPanel` Java?
32. Qual objeto é frequentemente usado como escutador de eventos em aplicações Java com GUI?
33. Qual é a origem do protocolo de um tratador de evento em Java?
34. Que método é usado para registrar um tratador de evento em Java?
35. Usando-se Windows Forms do .NET, qual espaço de nomes é exigido para se construir uma GUI para uma aplicação em C#?
36. Como um componente é posicionado em um formulário usando-se Windows Forms?
37. Qual é o protocolo de um tratador de evento .NET?
38. Qual classe de objetos deve ser criada para se registrar um tratador de evento .NET?
39. Que função os representantes exercem no processo de registro de tratadores de evento?

PROBLEMAS

1. O que os projetistas de C obtiveram ao não exigirem a verificação de faixas de índices?
2. Descreva três estratégias usadas para o tratamento de exceções em linguagens que não fornecem suporte direto para tal.

3. A partir de livros-texto sobre as linguagens de programação PL/I e Ada, busque os respectivos conjuntos de exceções predefinidas. Faça uma avaliação comparativa das duas, considerando tanto a completude quanto a flexibilidade.
4. A partir de um livro-texto sobre COBOL, determine como o tratamento de exceções é feito em programas dessa linguagem.
5. Em linguagens sem recursos para tratamento de exceções, é comum fazer a maioria dos subprogramas incluir um parâmetro de "erro", o qual pode ser configurado com algum valor representando "OK" ou outro representando "erro no procedimento". Que vantagem um recurso de tratamento de exceções linguístico como o de Java tem em relação a esse método?
6. Em uma linguagem sem recursos de tratamento de exceções, poderíamos enviar um procedimento de tratamento de erros como parâmetro para cada procedimento capaz de detectar erros a serem tratados. Quais são as desvantagens existentes nesse método?
7. Compare os métodos sugeridos nos Problemas 5 e 6. Qual deles você acha melhor? Por quê?
8. Escreva uma análise comparativa sobre a cláusula **`throw`** de C++ e a cláusula **`throws`** de Java.
9. Considere o seguinte esqueleto de programa C++:

```
class Big {
  int i;
  float f;
  void fun1() throw int {
       . . .
       try {
         . . .
         throw i;
       . . .
       throw f;
       . . .
     }
     catch(float) { . . . }
     . . .
  }
}
class Small {
  int j;
  float g;
  void fun2() throw float {
    . . .
    try {
      . . .
      try {
        Big.fun1();
        . . .
```

```
            throw j;
            . . .
            throw g;
            . . .
        }
        catch(int) { . . . }
        . . .
      }
      catch(float) { . . . }
    }
}
```

10. Em cada uma das quatro sentenças **throw**, onde a exceção é tratada? Note que `fun1` é chamado a partir de `fun2` na classe `Small`.
11. Escreva uma análise comparativa detalhada sobre as capacidades de tratamento de exceções de C++ e as de Java.
12. Com a ajuda de um livro sobre ML, escreva uma análise comparativa detalhada sobre as capacidades de tratamento de exceções de ML e as de Java.
13. Resuma os argumentos a favor dos modelos de continuação término e reinício.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Suponha que você esteja escrevendo uma função C++ com três estratégias alternativas para implementar seus requisitos. Escreva uma versão esqueleto dessa função de forma que, se a primeira alternativa levantar qualquer exceção, a segunda será tentada, e, se a segunda alternativa levantar qualquer exceção, a terceira será executada. Escreva o código como se existissem três métodos, em que os procedimentos fossem chamados de `alt1`, `alt2` e `alt3`.
2. Escreva um programa em Java que leia do teclado uma lista de valores inteiros na faixa de -100 a 100 e calcule a soma dos quadrados dos valores de entrada. Esse programa deve usar tratamento de exceções para garantir que os valores de entrada estejam na faixa e sejam valores inteiros válidos, para tratar o erro que ocorre quando a soma dos quadrados se torna maior do que uma variável `Integer` padrão pode armazenar, e para detectar o fim de um arquivo e usá-lo para mostrar o resultado na saída. No caso de transbordamento da soma, uma mensagem de erro deve ser impressa e o programa deve ser finalizado.
3. Escreva um programa em C++ para a especificação do Exercício de programação 2.
4. Revise o programa Java da Seção 14.3.5 usando `EOFException` para detectar o final da entrada.
5. Reescreva o código Java da Seção 14.3.6 para usar uma cláusula **finally** em C++.

Esta página foi deixada em branco intencionalmente.

15

Linguagens de programação funcional

15.1 Introdução

15.2 Funções matemáticas

15.3 Fundamentos das linguagens de programação funcional

15.4 A primeira linguagem de programação funcional: Lisp

15.5 Uma introdução a Scheme

15.6 Common Lisp

15.7 ML

15.8 Haskell

15.9 F#

15.10 Suporte para programação funcional em linguagens basicamente imperativas

15.11 Uma comparação entre linguagens funcionais e imperativas

Este capítulo apresenta a programação funcional e algumas das linguagens de programação que foram projetadas para essa estratégia de desenvolvimento de software. Começamos revisando os fundamentos das funções matemáticas, já que as linguagens funcionais são baseadas nelas. Depois, apresentamos o conceito de uma linguagem de programação funcional, seguido de um panorama da primeira linguagem funcional, Lisp. A seção seguinte, um tanto extensa, é dedicada a uma introdução a Scheme, incluindo algumas de suas funções primitivas, formas especiais, formas funcionais e alguns exemplos de funções simples escritas nessa linguagem. A seguir, fornecemos breves introduções a Common Lisp, ML, Haskell e F#. Então, discutimos o suporte para programação funcional incluído em algumas linguagens imperativas. A seção seguinte descreve algumas das aplicações das linguagens de programação funcional. Por fim, apresentamos um breve comparativo entre linguagens funcionais e imperativas.

15.1 INTRODUÇÃO

A maioria dos capítulos anteriores deste livro abordou principalmente as linguagens imperativas. O alto grau de similaridade entre as linguagens imperativas surge em parte de uma das bases comuns de seu projeto: a arquitetura de von Neumann, conforme discutido no Capítulo 1. As linguagens imperativas podem ser compreendidas coletivamente como uma progressão de desenvolvimentos realizados para melhorar o modelo básico, Fortran I. Todas foram projetadas para usar eficientemente a arquitetura de computadores de von Neumann. Apesar de o estilo imperativo de programação ser considerado aceitável pela maioria dos programadores, sua forte dependência da arquitetura subjacente é vista por alguns como uma restrição desnecessária nas estratégias alternativas de desenvolvimento de software.

Existem outras bases para o projeto de linguagens, algumas orientadas mais a paradigmas de programação em particular ou a metodologias para execução eficiente em determinada arquitetura de computadores. Até agora, entretanto, apenas uma minoria relativamente pequena de programas foi escrita em linguagens não imperativas.

O paradigma de programação funcional, baseado em funções matemáticas, é a base de projeto dos estilos de linguagem não imperativos mais importantes. Esse estilo de programação é suportado por linguagens de programação funcional.

O Prêmio Turing da ACM de 1977 foi concedido a John Backus por seu trabalho no desenvolvimento de Fortran. Cada ganhador realiza uma palestra quando recebe formalmente o prêmio, e esta é publicada na *Communications of the ACM*. Em sua palestra, Backus (1978) argumentou que linguagens de programação puramente funcionais são melhores que linguagens imperativas porque resultam em programas mais legíveis, mais confiáveis e mais propensos a serem corretos. O cerne de seu argumento é que os programas puramente funcionais são mais fáceis de entender, tanto durante quanto após o desenvolvimento, em grande parte porque os significados das expressões são independentes de seu contexto (uma característica de uma linguagem de programação funcional pura é que nem expressões nem funções têm efeitos colaterais).

Em sua palestra, Backus também propôs uma linguagem funcional pura, FP (*f*unctional *p*rogramming), que usou como base para sua argumentação. Apesar de a linguagem não ter sido bem-sucedida, ao menos em termos de atingir uso disseminado, ela

motivou o debate e a pesquisa em linguagens de programação puramente funcionais. O ponto-chave aqui é que alguns cientistas da computação bastante conhecidos têm tentado promover a noção de que as linguagens de programação funcional são superiores às linguagens imperativas tradicionais, mas esses esforços têm fracassado em seus objetivos. Contudo, no decorrer da última década, estimulado em parte pela maturidade das linguagens funcionais tipadas, como ML, Haskell, OCaml e F#, houve um aumento no interesse e no uso de linguagens de programação funcional.

Uma das características fundamentais dos programas escritos em linguagens imperativas é o fato de terem estado, o qual muda ao longo do processo de execução. Esse estado é representado pelas variáveis do programa. O autor e todos os leitores do programa devem entender os usos de suas variáveis e compreender como o estado do programa muda durante a execução. Para um programa grande, é uma tarefa intimidante. Esse é um problema dos programas escritos em uma linguagem imperativa que não está presente nos programas escritos em uma linguagem funcional pura, pois estes não têm variáveis nem estado.

Lisp começou como uma linguagem funcional pura, mas rapidamente adquiriu alguns recursos imperativos importantes que aumentaram sua eficiência de execução. Ela ainda é a mais importante das linguagens funcionais, ao menos no sentido de ser a única a atingir uso disseminado. A linguagem é a predominante nas áreas de representação de conhecimento, aprendizagem automática, sistemas de treinamento inteligentes e modelagem de fala. Common Lisp é um amálgama de diversos outros dialetos de Lisp do início dos anos 1980.

Scheme é um pequeno dialeto de Lisp, de escopo estático; ela é amplamente usada no ensino de programação funcional. É utilizada também em algumas universidades, em cursos introdutórios de programação.

O desenvolvimento de linguagens de programação funcional tipadas, principalmente ML, Haskell, OCaml e F#, levou a uma expansão significativa das áreas da computação nas quais agora são usadas linguagens funcionais. À medida que essas linguagens são atualizadas, seu uso prático aumenta. Atualmente, são utilizadas em áreas como processamento de bancos de dados, modelagem financeira, análise estatística e bioinformática.

Um objetivo deste capítulo é fornecer uma introdução à programação funcional por meio dos elementos básicos de Scheme, intencionalmente deixando de fora seus recursos imperativos. Material suficiente sobre essa linguagem é incluído para permitir que o leitor escreva alguns programas simples, mas interessantes. É difícil assimilar a programação funcional sem alguma experiência de programação prática, então isso é bastante incentivado.

15.2 FUNÇÕES MATEMÁTICAS

Uma função matemática é um mapeamento de membros de um conjunto, chamado de **conjunto domínio**, para outro, chamado de **conjunto imagem**. Uma definição de função especifica, explícita ou implicitamente, os conjuntos domínio e imagem junto com o mapeamento. Este é descrito por uma expressão ou, em alguns casos, por uma tabela. As funções são geralmente aplicadas a um elemento específico do conjunto domínio, fornecido como parâmetro para a função. Note que o conjunto domínio pode ser o pro-

duto vetorial de diversos conjuntos (refletindo o fato de que pode existir mais de um parâmetro). Uma função leva a um elemento do conjunto imagem.

Uma das características fundamentais das funções matemáticas é que a ordem de avaliação de suas expressões de mapeamento é controlada por recursão e por expressões condicionais, não por sequência e repetição iterativa, comuns nos programas escritos nas linguagens de programação imperativas.

Outra característica importante delas é que, como não possuem efeitos colaterais e não podem depender de valores externos, sempre mapeiam um elemento específico do domínio no mesmo elemento da imagem. Contudo, um subprograma em uma linguagem imperativa pode depender dos valores atuais de diversas variáveis não locais ou globais. Isso torna difícil determinar estaticamente os valores que o subprograma produzirá e os efeitos colaterais que terá em determinada execução.

Na matemática não existe algo como uma variável que modela uma posição de memória. Variáveis locais em funções em linguagens de programação imperativas mantêm o estado da função. A computação é realizada por meio da avaliação de expressões em sentenças de atribuição que alteram o estado do programa. Na matemática não existe o conceito de estado de uma função.

Uma função matemática mapeia seu parâmetro (ou parâmetros) em um valor (ou valores), em vez de especificar uma sequência de operações sobre valores na memória para produzir um valor.

15.2.1 Funções simples

Definições de funções são geralmente escritas como um nome de função, seguido de uma lista de parâmetros entre parênteses, seguidos pela expressão de mapeamento. Por exemplo,

cube(x) ≡ x * x * x, onde x é um número real.

Nessa definição, os conjuntos domínio e imagem são os números reais. O símbolo ≡ é usado para significar "é definido como". O parâmetro x pode representar qualquer membro do conjunto domínio, mas é fixado para representar um elemento específico durante a avaliação da expressão da função. Essa é uma das maneiras pelas quais os parâmetros das funções matemáticas diferem das variáveis em linguagens imperativas.

Aplicações de funções são especificadas por um par que contém o nome da função com um elemento particular do conjunto domínio. O elemento da imagem é obtido ao avaliarmos a expressão de mapeamento da função com o elemento do domínio substituído para as ocorrências do parâmetro. Mais uma vez, é importante notar que, durante a avaliação, o mapeamento de uma função não contém nenhum parâmetro desvinculado, em que um parâmetro vinculado é um nome para um valor específico. Cada ocorrência de um parâmetro é vinculada a um valor do conjunto domínio e é uma constante durante a avaliação. Por exemplo, considere a seguinte avaliação de cube(x):

cube (2.0) = 2.0 * 2.0 * 2.0 = 8

O parâmetro x é vinculado a 2.0 durante a avaliação e não há nenhum parâmetro desvinculado. Além disso, x é uma constante (seu valor não pode ser alterado) durante a avaliação.

Os primeiros trabalhos teóricos acerca de funções separaram a tarefa de defini-las da de nomeá-las. A notação lambda, definida por Alonzo Church (1941), fornece um método para definir funções não nomeadas. Uma **expressão lambda** especifica os parâmetros e o mapeamento de uma função. Ela é a função propriamente dita, que é não nomeada. Por exemplo, considere a seguinte expressão lambda:

λ(x)x * x * x

Church definiu um modelo de computação formal (um sistema formal para definição de funções, aplicação de funções e recursão) usando expressões lambda. Ele se chama **cálculo lambda**, que pode ser tipado ou não tipado. O cálculo lambda não tipado serve de inspiração para as linguagens de programação funcionais.

Conforme dito anteriormente, antes da avaliação, um parâmetro representa qualquer membro do conjunto domínio, mas durante a avaliação ele é vinculado a um membro específico. Quando uma expressão lambda é avaliada para um parâmetro, diz-se que ela é aplicada a esse parâmetro. A mecânica de tal aplicação é a mesma para qualquer avaliação de função. A aplicação da expressão lambda do exemplo é denotada como a seguir:

(λ(x)x * x * x)(2)

que resulta no valor 8.

Expressões lambda, como outras definições de função, podem ter mais de um parâmetro.

15.2.2 Formas funcionais

Uma **função de ordem superior**, ou **forma funcional**, é aquela que recebe uma ou mais funções como parâmetros, ou que leva a uma função como resultado, ou ambos. Um tipo comum de forma funcional é a **composição funcional**, que tem dois parâmetros funcionais e leva a uma função cujo valor é o primeiro parâmetro de função real aplicado ao resultado do segundo. A composição funcional é escrita como uma expressão, usando ° como um operador, como em

$h \equiv f \circ g$

Por exemplo, se

$f(x) \equiv x + 2$
$g(x) \equiv 3*x$

então h é definida como

$h(x) \equiv f(g(x))$, ou $h(x) = (3*x) + 2$

Aplicar-para-todos (apply-to-all) é uma forma funcional que recebe uma única função como parâmetro.[1] Se usada em uma lista de parâmetros, aplicar-para-todos aplica seu parâmetro funcional a cada um dos valores na lista de parâmetros e coleta os resultados em uma lista ou em uma sequência. Aplicar-para-todos é denotada por α.

Seja

$h(x) \equiv x*x$

então

$\alpha(h,(2,3,4))$ resulta em (4, 9, 16)

Existem outras formas funcionais, mas esses dois exemplos ilustram as características básicas de todas elas.

15.3 FUNDAMENTOS DAS LINGUAGENS DE PROGRAMAÇÃO FUNCIONAL

O objetivo do projeto de uma linguagem de programação funcional é imitar as funções matemáticas ao máximo possível. Isso resulta em uma estratégia para a solução de problemas fundamentalmente diferente das usadas com linguagens imperativas. Em uma linguagem imperativa, uma expressão é avaliada e o resultado é armazenado em uma posição de memória, representada como uma variável em um programa. Essa é a finalidade das sentenças de atribuição. Essa atenção necessária às células da memória, cujos valores representam o estado do programa, resulta em uma metodologia de programação de nível relativamente baixo.

Um programa em uma linguagem de montagem geralmente também armazena os resultados de avaliações parciais de expressões. Por exemplo, para avaliar

$(x + y)/(a - b)$

o valor de $(x + y)$ é calculado primeiro. Esse valor deve então ser armazenado enquanto $(a - b)$ é avaliada. Em linguagens de alto nível, o compilador manipula o armazenamento dos resultados intermediários de avaliações de expressões. O armazenamento de resultados intermediários ainda é necessário, mas os detalhes não podem ser vistos pelo programador.

Uma linguagem de programação puramente funcional não usa variáveis nem sentenças de atribuição, liberando o programador de preocupações relacionadas às células de memória, ou ao estado, do programa. Sem variáveis, as construções de iteração não são possíveis, já que são controladas por variáveis. As repetições devem ser especificadas com recursão, e não com iteração. Os programas são definições de funções e especificações de aplicações de funções, e as execuções consistem em avaliar a aplicação de funções. Sem variáveis, a execução de um programa puramente funcional não tem estado, no sentido de semântica operacional e denotacional. A execução de uma função sempre produz o mesmo resultado quando fornecidos os mesmos parâmetros. Esse recurso é chamado de **transparência referencial**. Ele torna a semântica de linguagens puramente funcionais

[1] Em linguagens de programação, são geralmente denominadas funções *mapa*.

muito mais simples que a semântica das linguagens imperativas (e que as linguagens funcionais que incluem recursos imperativos). Também torna os testes mais fáceis, pois cada função pode ser testada separadamente, sem qualquer preocupação com seu contexto.

Uma linguagem funcional fornece um conjunto de funções primitivas, um conjunto de formas funcionais para construir funções complexas a partir dessas funções primitivas, uma operação de aplicação de função e alguma estrutura ou estruturas para representar dados. Essas estruturas são usadas para representar os parâmetros e os valores computados pelas funções. Se uma linguagem funcional é bem projetada, ela requer apenas um número relativamente pequeno de funções primitivas.

Como vimos em capítulos anteriores, a primeira linguagem de programação funcional, Lisp, usa uma forma sintática para dados e para código muito diferente da de linguagens imperativas. Contudo, muitas linguagens funcionais projetadas depois utilizam para seu código uma sintaxe semelhante à das linguagens imperativas.

Embora existam algumas linguagens puramente funcionais, por exemplo, Haskell, a maior parte das que são chamadas funcionais contém alguns recursos imperativos, como variáveis mutáveis e construções que atuam como sentenças de atribuição.

Alguns conceitos e construções originadas nas linguagens funcionais, como avaliação tardia e subprogramas anônimos, agora são usados em algumas linguagens consideradas imperativas.

Apesar de as primeiras linguagens funcionais terem sido muitas vezes implementadas com interpretadores, muitos programas escritos em linguagens de programação funcional agora são compilados.

15.4 A PRIMEIRA LINGUAGEM DE PROGRAMAÇÃO FUNCIONAL: LISP

Muitas linguagens de programação funcional foram desenvolvidas. A mais antiga e mais utilizada é Lisp (ou uma de suas descendentes), desenvolvida por John McCarthy no MIT, em 1959. Estudar linguagens funcionais por meio de Lisp é de certa forma equivalente a estudar as linguagens imperativas por meio de Fortran: Lisp foi a primeira linguagem funcional, mas, apesar de ter evoluído continuamente por meio século, não representa os mais recentes conceitos de projeto para linguagens funcionais. Além disso, com exceção da primeira versão, todos os dialetos de Lisp incluem recursos imperativos, como variáveis no estilo imperativo, sentenças de atribuição e iteração. (Variáveis no estilo imperativo são usadas para nomear células de memória, cujos valores podem ser modificados muitas vezes durante a execução de um programa.) Apesar disso e de suas formas um pouco não convencionais, as descendentes de Lisp original representam bem os conceitos fundamentais da programação funcional e, por isso, merecem ser analisadas aqui.

15.4.1 Tipos de dados e estruturas

Em Lisp original havia apenas duas categorias de objetos de dados: átomos e listas. Os elementos de lista são pares, em que a primeira parte é o dado do elemento, um ponteiro para um átomo ou uma lista aninhada. A segunda parte de um par pode ser um ponteiro para um átomo, um ponteiro para outro elemento ou um valor especial, nil. Os elementos são vinculados nas listas às segundas partes. Átomos e listas não são tipos, no sentido de que as

linguagens imperativas têm tipos. Na verdade, Lisp original era uma linguagem desprovida de tipos. Átomos são símbolos, na forma de identificadores ou literais numéricos.

Lembre-se, do Capítulo 2, de que Lisp originalmente usava listas como sua estrutura de dados porque se pensava que seriam essenciais para o processamento de listas. No entanto, conforme foi desenvolvido, Lisp passou a raramente exigir as operações de lista gerais de inserção e exclusão em posições que não são o início de uma lista.

As listas são especificadas em Lisp ao delimitarmos seus elementos com parênteses. Os elementos de **listas simples** são restritos aos átomos, como em

```
(A B C D)
```

Estruturas de lista aninhadas também são especificadas com parênteses. Por exemplo,

```
(A (B C) D (E (F G)))
```

é uma lista de quatro elementos. O primeiro é o átomo `A`, o segundo é a sublista `(B C)`, o terceiro é o átomo `D` e o quarto é a sublista `(E(F G))`, que tem como segundo elemento a sublista `(F G)`.

Em uma implementação de Lisp, uma lista normalmente é armazenada como uma estrutura de lista encadeada na qual cada nó tem dois ponteiros, um para referenciar os dados do nó e o outro para formar a lista encadeada. Uma lista é referenciada por um ponteiro para o seu primeiro elemento.

As representações internas de nossas duas listas de exemplo são mostradas na Figura 15.1. Note que os elementos de uma lista são mostrados horizontalmente. O último

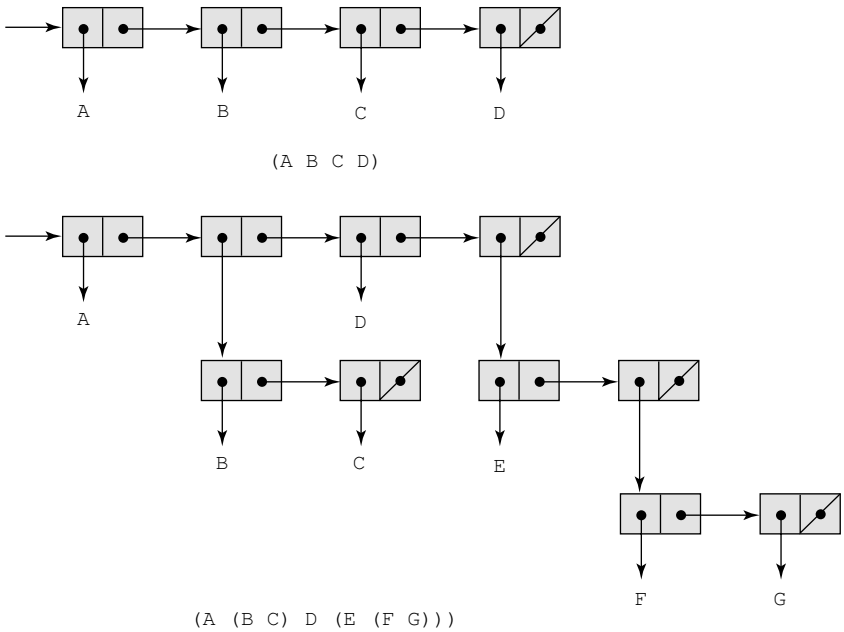


FIGURA 15.1
Representação interna de duas listas Lisp.

elemento de uma lista não tem um sucessor, então sua ligação é nil. Sublistas são mostradas com a mesma estrutura.

15.4.2 O primeiro interpretador Lisp

O objetivo original do projeto de Lisp era ter uma notação para programas o mais próxima possível de Fortran, com adições quando necessário. Essa notação era chamada de notação-M (metanotação). Deveria existir um compilador que traduziria programas escritos em notação-M para programas em código de máquina equivalentes para o IBM 704.

No início do desenvolvimento de Lisp, McCarthy escreveu um artigo para promover o processamento de listas como uma estratégia para o processamento simbólico geral. Ele acreditava que o processamento de listas poderia ser usado para estudar computabilidade, na época investigada por meio de máquinas de Turing, as quais se baseiam no modelo imperativo de computação. McCarthy achou que o processamento funcional de listas simbólicas era um modelo mais natural de computação que as máquinas de Turing, que operavam em símbolos escritos em fitas, os quais representavam o estado. Um dos requisitos mais comuns do estudo da computação é possibilitar que as pessoas provem certas características de computabilidade da classe completa de qualquer modelo de computação usado. No caso do modelo da máquina de Turing, alguém pode construir uma máquina de Turing universal capaz de imitar as operações de qualquer outra dessas máquinas. A partir desse conceito, surgiu a ideia de construir uma função Lisp universal para avaliar qualquer outra função em Lisp.

O primeiro requisito para a função Lisp universal era uma notação que permitisse às funções serem expressas da mesma maneira que os dados. A notação de listas entre parênteses da Seção 15.4.1 já havia sido adotada para os dados de Lisp, então decidiu-se criar convenções para definições de funções e chamadas de funções que também poderiam ser expressas em notação de lista. As chamadas a funções eram especificadas em uma forma de lista prefixada, chamada de **Cambridge Polonesa**,[2] como a seguir:

(nome_da_função parâmetro$_1$ . . . parâmetro$_n$)

Por exemplo, se + é uma função que recebe dois ou mais parâmetros numéricos, as duas expressões a seguir são avaliadas como `12` e `20`, respectivamente:

```
(+ 5 7)
(+ 3 4 7 6)
```

A notação lambda descrita na Seção 15.2.1 foi escolhida para especificar definições de funções. Entretanto, ela precisou ser modificada para permitir a vinculação de funções a nomes, de forma que as funções pudessem ser referenciadas por outras e por elas

[2]Esse nome foi usado pela primeira vez no desenvolvimento inicial de Lisp. Ele foi escolhido porque as listas de Lisp lembram a notação prefixada utilizada pelo especialista em teoria lógica polonês Jan Lukasiewicz e porque Lisp havia nascido no MIT, em Cambridge, Massachusetts, EUA. Atualmente, alguns preferem chamar a notação de *Cambridge prefixada*.

próprias. A vinculação de nome era especificada por uma lista composta do nome da função e por uma lista contendo a expressão lambda, como em

(nome_da_função (LAMBDA ($param_1$... $param_n$) expressão))

Se você não conhece a programação funcional, pode parecer estranho até mesmo considerar uma função sem nome. Entretanto, funções sem nomes são algumas vezes úteis em nesse tipo de programação (bem como na matemática e na programação imperativa).[3] Por exemplo, considere uma função cuja ação é produzir outra para aplicação imediata a uma lista de parâmetros. A função produzida não precisa de um nome, já que é aplicada apenas no ponto de sua construção. Um exemplo disso é dado na Seção 15.5.14.

Funções Lisp especificadas com essa nova notação eram chamadas de expressões-S (expressões simbólicas). No fim, todas as estruturas Lisp, tanto dados quanto código, eram chamadas expressões-S. Uma expressão-S pode ser uma lista ou um átomo. Em geral, vamos nos referir a elas simplesmente como expressões.

McCarthy desenvolveu de maneira bem-sucedida uma função universal que poderia avaliar qualquer outra função. Ela foi chamada EVAL e tinha a forma de uma expressão. Dois dos pesquisadores do Projeto IA que estavam desenvolvendo Lisp, Stephen B. Russell e Daniel J. Edwards, deram-se conta de que uma implementação de EVAL poderia servir como um interpretador Lisp e prontamente construíram tal implementação (McCarthy et al., 1965).

Vários resultados importantes decorreram dessa implementação rápida, fácil e inesperada. Primeiro, todas as implementações iniciais de Lisp copiaram EVAL e, dessa forma, eram interpretativas. Segundo, a definição da notação-M, a notação de programação planejada para Lisp, nunca foi concluída ou implementada; então, as expressões-S se tornaram a única notação de Lisp. O uso da mesma notação para dados e código tem consequências importantes, uma das quais será discutida na Seção 15.5.14. Terceiro, grande parte do projeto original da linguagem foi efetivamente congelada, o que manteve certos recursos incomuns nela, como a forma de expressão condicional e o uso de () tanto para a lista vazia quanto para o falso lógico.

Outro recurso dos primeiros sistemas Lisp que apareceu de maneira acidental foi o uso de escopo dinâmico. As funções eram avaliadas no ambiente de seus chamadores. Ninguém na época sabia muito sobre escopo, e pode ter sido dada pouca atenção à escolha. O escopo dinâmico foi usado pela maioria dos dialetos de Lisp antes de 1975. Os dialetos contemporâneos usam escopo estático, ou permitem que o programador escolha entre escopo estático e dinâmico.

Um interpretador para Lisp pode ser escrito em Lisp. Tal interpretador, que não é um programa grande, descreve a semântica operacional de Lisp, em Lisp. Essa é uma clara evidência da simplicidade semântica da linguagem.

[3] Também existem usos para subprogramas sem nomes na programação imperativa.

15.5 UMA INTRODUÇÃO A SCHEME

Nesta seção, descrevemos os aspectos principais de Scheme (Dybvig, 2009). Escolhemos essa linguagem porque ela é relativamente simples e popular em faculdades e universidades, e porque interpretadores Scheme estão prontamente disponíveis (gratuitamente) para uma ampla variedade de computadores. A versão de Scheme descrita nesta seção é a 4. Note que esta seção cobre apenas uma pequena parte da linguagem e não inclui nenhum dos seus recursos imperativos.

15.5.1 Origens de Scheme

A linguagem Scheme, um dialeto de Lisp, foi desenvolvida no MIT em meados dos anos 1970 (Sussman e Steele, 1975). Ela é caracterizada por ser pequena, ter uso exclusivo de escopo estático e tratamento de funções como entidades de primeira classe. Como entidades de primeira classe, as funções Scheme podem ser os valores de expressões, elementos de listas, passadas como parâmetros e retornadas de funções. As primeiras versões de Lisp não forneciam todas essas capacidades.

Como uma linguagem pequena, basicamente desprovida de tipos, com sintaxe e semântica simples, Scheme é adequada para aplicações educacionais, como cursos sobre programação funcional, e para introduções gerais à programação.

A maior parte do código Scheme nas seções a seguir exigiria apenas pequenas modificações para ser convertido em código Lisp válido.

15.5.2 O interpretador Scheme

Um interpretador Scheme no modo interativo é um laço infinito de leitura-avaliação-impressão (muitas vezes abreviado como REPL - read-evaluate-print loop). Ele repetidamente lê uma expressão digitada pelo usuário (na forma de uma lista), interpreta a expressão e mostra o valor resultante. Essa forma de interpretador também é usada por Ruby e Python. Expressões são interpretadas pela função `EVAL`. Os literais são avaliados para eles próprios. Então, se você digitar um número para o interpretador, ele simplesmente mostra o número. Expressões que são chamadas a funções primitivas são avaliadas da seguinte maneira: primeiro, cada uma das expressões parametrizadas é avaliada, sem nenhuma ordem específica. Então, a função primitiva é aplicada para os valores de parâmetros e o valor resultante é mostrado.

Evidentemente, os programas Scheme armazenados em arquivos podem ser carregados e interpretados.

Em Scheme, os comentários são compostos de qualquer texto após um ponto e vírgula em qualquer linha.

15.5.3 Funções numéricas primitivas

Scheme contém funções primitivas para as operações aritméticas básicas. São elas: `+`, `−`, `*` e `/`, para adição, subtração, multiplicação e divisão. As funções `*` e `+` podem ter zero ou mais parâmetros. Se não for passado um parâmetro para `*`, ela retorna `1`; se não for dado nenhum parâmetro para `+`, ela retorna `0`. A função `+` soma todos os seus parâmetros. A função `*` multiplica todos os seus parâmetros. As funções `/` e `−` podem ter dois

ou mais parâmetros. No caso da subtração, todos os parâmetros, exceto o primeiro, são subtraídos do primeiro. A divisão é similar à subtração. Alguns exemplos são:

Expressão	*Valor*
`42`	`42`
`(* 3 7)`	`21`
`(+ 5 7 8)`	`20`
`(- 5 6)`	`-1`
`(- 15 7 2)`	`6`
`(- 24 (* 4 3))`	`12`

Há uma grande quantidade de outras funções numéricas em Scheme, dentre elas `MODULO`, `ROUND`, `MAX`, `MIN`, `LOG`, `SIN` e `SQRT`. Esta retorna a raiz quadrada de seu parâmetro numérico se o valor do parâmetro não for negativo. Se o parâmetro for negativo, `SQRT` resultará em um número complexo.

Em Scheme, observe que usamos letras maiúsculas para todas as palavras reservadas e funções predefinidas. A definição oficial da linguagem especifica que, nelas, não há distinção entre maiúsculas e minúsculas. Contudo, algumas implementações, como as linguagens de ensino de DrRacket, exigem minúsculas para palavras reservadas e funções predefinidas.

Se uma função tem um número fixo de parâmetros, como a `SQRT`, a quantidade deles na chamada deve combinar com esse número. Caso contrário, o interpretador produzirá uma mensagem de erro.

15.5.4 Definição de funções

Um programa Scheme é uma coleção de definições de funções. Consequentemente, saber definir essas funções é um pré-requisito para escrever o programa mais simples. Em Scheme, uma função não nomeada inclui a palavra `LAMBDA` e é chamada de **expressão lambda**. Por exemplo,

```
(LAMBDA (x) (* x x))
```

é uma função não nomeada que retorna o quadrado de seu parâmetro numérico dado. Essa função pode ser aplicada da mesma maneira que as funções nomeadas: colocando-a no início de uma lista que contém os parâmetros reais. Por exemplo, a seguinte expressão retorna `49`:

```
((LAMBDA (x) (* x x)) 7)
```

Nessa expressão, `x` é chamada de **variável vinculada** dentro da expressão lambda. Durante a avaliação dessa expressão, `x` é vinculada a `7`. Uma variável vinculada nunca muda na expressão após ter sido vinculada a um valor de parâmetro real no momento em que inicia a avaliação da expressão lambda.

As expressões lambda podem ter qualquer número de parâmetros. Por exemplo, poderíamos ter o seguinte:

```
(LAMBDA (a b c x) (+ (* a x x) (* b x) c))
```

A função de forma especial DEFINE serve a duas necessidades fundamentais da programação em Scheme: vincular um nome a um valor e um nome a uma expressão lambda. A forma de DEFINE que vincula um nome a um valor pode dar a entender que DEFINE pode ser usada para criar variáveis no estilo das linguagens imperativas. Entretanto, essas vinculações de nomes criam valores nomeados, não variáveis.

DEFINE é chamada de forma especial porque é interpretada (por EVAL) de uma maneira diferente das primitivas normais, como as funções aritméticas, como mostraremos em breve.

A forma mais simples de DEFINE é aquela usada para vincular um nome ao valor de uma expressão. Essa forma é

```
(DEFINE símbolo expressão)
```

Por exemplo,

```
(DEFINE pi 3,14159)
(DEFINE two_pi (* 2 pi))
```

Se essas duas expressões fossem digitadas para o interpretador Scheme e pi fosse digitado, o número 3,14159 seria mostrado; se two_pi fosse digitada, 6,28318 seria mostrado. Em ambos os casos, os números apresentados podem ter mais dígitos do que os mostrados aqui.

Essa forma de DEFINE é análoga a uma declaração de constante nomeada em uma linguagem imperativa. Por exemplo, em Java, os equivalentes dos nomes definidos acima são os seguintes:

```
final float PI = 3.14159;
final float TWO_PI = 2.0 * PI;
```

Nomes em Scheme podem ser formados por letras, dígitos e caracteres especiais, exceto parênteses; eles não diferenciam maiúsculas e minúsculas e não podem começar com um dígito.

O segundo uso da função DEFINE é vincular uma expressão lambda a um nome. Nesse caso, a expressão lambda é abreviada ao se remover a palavra LAMBDA. Para vincular um nome a uma expressão lambda, DEFINE recebe duas listas como parâmetros. O primeiro parâmetro é o protótipo de uma chamada à função, com o nome da função seguido pelos parâmetros formais, juntos, em uma lista. A segunda lista contém uma expressão, à qual o nome será vinculado. A forma geral de tal DEFINE é[4]

```
(DEFINE (nome_da_função parâmetros)
    (expressão)
)
```

[4]Na verdade, a forma geral de DEFINE tem como corpo uma lista contendo uma sequência de uma ou mais expressões, embora, em muitos casos, somente uma seja incluída. Aqui, incluímos apenas uma pela simplicidade.

É claro que essa forma de DEFINE é a definição de uma função nomeada.

A seguinte chamada de exemplo a DEFINE vincula o nome square à expressão funcional que recebe um parâmetro:

```
(DEFINE (square number) (* number number))
```

Após o interpretador avaliar essa função, ela pode ser usada, como em

```
(square 5)
```

que mostra 25.

Para ilustrar a diferença entre funções primitivas e a forma especial DEFINE, considere o seguinte:

```
(DEFINE x 10)
```

Se DEFINE fosse uma função primitiva, a primeira ação de EVAL nessa expressão seria avaliar os dois parâmetros de DEFINE. Se x ainda não estivesse vinculado a um valor, isso seria um erro. Além disso, se x já tivesse sido definido, isso também seria um erro, porque DEFINE tentaria redefinir x, o que é ilegal. Lembre-se de que x é o nome de um valor; não é uma variável no sentido imperativo.

A seguir, temos outro exemplo de função. Ela computa o tamanho da hipotenusa (o lado maior) de um triângulo retângulo, dado o tamanho de seus dois outros lados.

```
(DEFINE (hypotenuse side1 side2)
    (SQRT(+(square side1)(square side2)))
)
```

Note que a função hypotenuse usa square, que já foi definida anteriormente.

15.5.5 Funções de saída

Scheme contém algumas funções de saída simples; mas, quando usada com o interpretador interativo, a maior parte da saída de programas Scheme é a saída normal do interpretador, que mostra os resultados da aplicação de EVAL às funções de nível superior.

Note que entrada e saída explícitas não fazem parte do modelo de programação funcional puro, pois operações de entrada alteram o estado do programa e operações de saída têm efeitos colaterais. Nada disso pode fazer parte de uma linguagem funcional pura. Portanto, este capítulo não descreve as funções de entrada e saída explícitas de Scheme.

15.5.6 Funções de predicado numérico

Uma função de predicado é aquela que retorna um valor booleano (alguma representação de verdadeiro ou falso). Scheme inclui uma coleção de funções de predicado para dados numéricos. Dentre elas, estão:

Função	*Significado*
=	Igual
<>	Diferente
>	Maior que
<	Menor que
>=	Maior ou igual a
<=	Menor ou igual a
EVEN?	É um número par?
ODD?	É um número ímpar?
ZERO?	É zero?

Note que os nomes de todas as funções de predicado predefinidas que têm palavras para nomes terminam com pontos de interrogação. Em Scheme, os dois valores booleanos são `#T` e `#F` (ou `#t` e `#f`), embora as funções de predicado predefinidas de Scheme retornem a lista vazia, `()`, para falso.

Quando uma lista é interpretada como um valor booleano, qualquer lista não vazia é avaliada como verdadeira; a lista vazia é avaliada como falsa. Isso é semelhante à interpretação de inteiros como valores booleanos em C; zero é avaliado como falso e qualquer valor diferente de zero é avaliado como verdadeiro.

Para melhorar a legibilidade, todas as nossas funções de predicado neste capítulo retornam `#F` em vez de `()`.

A função `NOT` é usada para inverter a lógica de uma expressão booleana.

15.5.7 Controle de fluxo

Scheme usa três construções para controle de fluxo, uma semelhante à construção de seleção das linguagens imperativas e duas baseadas no controle de avaliação usado em funções matemáticas.

A função seletora de dois caminhos de Scheme, chamada `IF`, tem três parâmetros: uma expressão de predicado, uma expressão então (then) e uma expressão senão (else). Uma chamada a `IF` tem a forma

`(IF` predicado expressão_então expressão_senão`)`

Por exemplo,

```
(DEFINE (factorial n)
  (IF (<= n 1)
    1
    (* n (factorial (- n 1)))
))
```

Lembre-se de que a seleção múltipla de Scheme, COND, foi discutida no Capítulo 8. A seguir, temos um exemplo de função simples que usa COND:

```
(DEFINE (leap? year)
  (COND
    ((ZERO? (MODULO year 400)) #T)
    ((ZERO? (MODULO year 100)) #F)
    (ELSE (ZERO? (MODULO year 4)))
))
```

As subseções seguintes contêm exemplos adicionais do uso de COND.

O terceiro mecanismo de controle em Scheme é a recursão, usada, como na matemática, para especificar repetição. A maioria das funções de exemplo na Seção 15.5.10 usa recursão.

15.5.8 Funções de lista

Um dos usos mais comuns das linguagens de programação baseadas em Lisp é o processamento de listas. Esta subseção apresenta as funções de Scheme utilizadas para lidar com listas. Lembre-se de que as operações de lista de Scheme foram apresentadas brevemente no Capítulo 6. A seguir está uma discussão mais detalhada sobre o processamento de listas nessa linguagem.

Os programas Scheme são interpretados pela função de aplicação de função, EVAL. Quando aplicada a uma função primitiva, EVAL avalia primeiro os parâmetros da função. Essa ação é necessária quando os parâmetros reais de uma chamada à função são, eles próprios, chamadas a funções, o que acontece com frequência. Em algumas chamadas, entretanto, os parâmetros são elementos de dados, em vez de referências a funções. Quando um parâmetro não é uma referência a uma função, obviamente não deve ser avaliado. Não estávamos preocupados com isso antes porque os literais numéricos sempre são avaliados por eles próprios e não podem ser confundidos com nomes de funções.

Suponha que tenhamos uma função com dois parâmetros, um átomo e uma lista, e que o propósito dela seja determinar se o átomo está na lista dada. Nem o átomo nem a lista devem ser avaliados; eles são dados literais a serem processados. Para evitar a avaliação de um parâmetro, ele primeiro é passado como parâmetro para a função primitiva QUOTE, que simplesmente o retorna, sem modificações. O seguinte exemplo ilustra QUOTE:

`(QUOTE A)` retorna `A`
`(QUOTE (A B C))` retorna `(A B C)`

As chamadas a QUOTE normalmente são abreviadas, precedendo-se a expressão a ser citada por um apóstrofo (') e omitindo-se os parênteses em torno da expressão. Então, em vez de `(QUOTE (A B))`, `'(A B)` é usado.

QUOTE é necessário devido à natureza fundamental de Scheme (e das outras linguagens baseadas em Lisp): dados e código têm a mesma forma. Embora isso possa parecer estranho para programadores de linguagens imperativas, resulta em alguns processos interessantes e eficientes, um dos quais está discutido na Seção 15.5.14.

As funções CAR, CDR e CONS foram apresentadas no Capítulo 6. A seguir estão mais exemplos das operações de CAR e CDR:

```
(CAR '(A B C)) retorna A
(CAR '((A B) C D)) retorna (A B)
(CAR 'A) é um erro porque A não é uma lista
(CAR '(A)) retorna A
(CAR '()) é um erro
(CDR '(A B C)) retorna (B C)
(CDR '((A B) C D)) retorna (C D)
(CDR 'A) é um erro
(CDR '(A)) retorna ()
(CDR '()) é um erro
```

Os nomes das funções CAR e CDR são, no mínimo, peculiares. A origem deles é a primeira implementação de Lisp, feita em um computador IBM 704. As palavras de memória do 704 tinham dois campos, chamados decremento e endereço, usados em várias estratégias de endereçamento de operandos. Cada um desses campos pode armazenar um endereço de memória de máquina. O 704 incluía também duas instruções de máquina, também chamadas CAR (o *c*onteúdo da parte do endereço [*a*ddress] de um *r*egistro) e CDR (o *c*onteúdo da parte do *d*ecremento de um *r*egistro), que extraíam os campos associados. Era natural usar os dois campos para armazenar os dois ponteiros de um nó de lista, de forma que uma palavra de memória podia armazenar um nó muito bem. Usando essas convenções, as instruções CAR e CDR do 704 forneceram seletores de listas eficientes. Os nomes continuaram sendo usados nas primitivas de todos os dialetos de Lisp.

Como outro exemplo de função simples, considere

```
(DEFINE (second a_list) (CAR (CDR a_list)))
```

Uma vez que essa função for avaliada, ela pode ser usada, como em

```
(second '(A B C))
```

que retorna B.

Algumas das composições funcionais mais usadas em Scheme são construídas como funções simples. Por exemplo, (CAAR x) é equivalente a (CAR (CAR x)), (CADR x) é equivalente a (CAR (CDR x)) e (CADDAR x) é equivalente a (CAR (CDR (CDR (CAR x)))). Qualquer combinação de até quatro As e Ds é válida entre o C e o R no nome da função. Como exemplo, considere a seguinte avaliação de CADDAR:

```
(CADDAR '((A B (C) D) E)) =
(CAR (CDR (CDR (CAR '((A B (C) D) E))))) =
(CAR (CDR (CDR '(A B (C) D)))) =
```

```
(CAR (CDR '(B (C) D))) =
(CAR '((C) D)) =
(C)
```

A seguir estão exemplos de chamadas a `CONS`:

`(CONS 'A '())` retorna `(A)`
`(CONS 'A '(B C))` retorna `(A B C)`
`(CONS '() '(A B))` retorna `(() A B)`
`(CONS '(A B) '(C D))` retorna `((A B) C D)`

O resultado dessas operações `CONS` é mostrado na Figura 15.2. Note que `CONS` é, em certo sentido, o inverso de `CAR` e `CDR`. Estas separam uma lista, e `CONS` constrói uma nova lista a partir de duas partes de uma lista. Os dois parâmetros de `CONS` se tornam o `CAR` e o `CDR` da nova lista. Logo, se `a_list` é uma lista, então

```
(CONS (CAR a_list) (CDR a_list))
```

retorna uma lista com a mesma estrutura e os mesmos elementos de `a_list`.

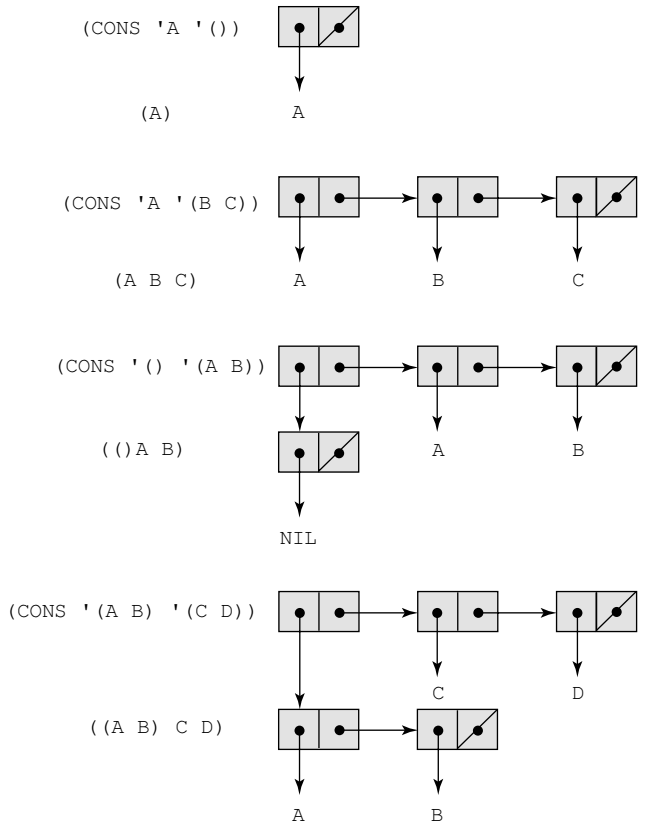


FIGURA 15.2
O resultado de diversas operações `CONS`.

Ao lidar apenas com os problemas e programas relativamente simples discutidos neste capítulo, é provável que ninguém aplique intencionalmente CONS a dois átomos, embora isso seja válido. O resultado de tal aplicação é um par assinalado com um ponto, nomeado dessa forma devido à maneira como é mostrado em Scheme. Por exemplo, considere a seguinte chamada:

```
(CONS 'A 'B)
```

Se o resultado dela for mostrado, aparecerá como

```
(A . B)
```

Esse par assinalado com um ponto indica que, em vez de um átomo e um ponteiro ou um ponteiro e um ponteiro, essa célula contém dois átomos.

LIST é uma função que constrói uma lista a partir de um número variável de parâmetros. Ela é uma versão simplificada de funções CONS aninhadas, como ilustrado a seguir:

```
(LIST 'apple 'orange 'grape)
```

retorna

```
(apple orange grape)
```

Usando CONS, a chamada a LIST acima é escrita como segue:

```
(CONS 'apple (CONS 'orange (CONS 'grape '())))
```

15.5.9 Funções de predicado para átomos simbólicos e listas

Scheme tem três funções de predicado fundamentais, EQ?, NULL? e LIST?, para átomos simbólicos e listas.

A função EQ? recebe duas expressões como parâmetros, embora seja normalmente usada com dois parâmetros de átomo simbólico. Ela retorna #T se os dois parâmetros têm o mesmo valor de ponteiro – isto é, se apontam para o mesmo átomo ou lista; caso contrário, retorna #F. Se os dois parâmetros são átomos simbólicos, EQ? retorna #T caso sejam os mesmos símbolos (pois Scheme não faz duplicatas de símbolos); caso contrário, #F. Considere os exemplos a seguir:

`(EQ? 'A 'A)` retorna `#T`
`(EQ? 'A 'B)` retorna `#F`
`(EQ? 'A '(A B))` retorna `#F`
`(EQ? '(A B) '(A B))` retorna `#F` ou `#T`
`(EQ? 3.4 (+ 3 0.4))` retorna `#F` ou `#T`

Conforme o quarto exemplo indica, o resultado de comparar listas com EQ? não é consistente. O motivo disso é que duas listas exatamente iguais frequentemente não são duplicadas na memória. No momento em que o sistema Scheme cria uma lista, ele verifica se ela já não existe. Se existir, a nova lista será nada mais que um ponteiro para

a já existente. Nesses casos, as duas listas seriam julgadas iguais por `EQ?`. Entretanto, em alguns casos, pode ser difícil detectar a presença de uma lista idêntica, o que leva à criação de uma nova lista. Nesse cenário, `EQ?` leva a `#F`.

O último caso mostra que a adição pode produzir um novo valor e, assim, não seria igual (com `EQ?`) a `3.4`, ou pode reconhecer que já tem o valor `3.4` e utilizá-lo; nesse caso, `EQ?` usará o ponteiro para o antigo `3.4` e retornará `#T`.

Como vimos, `EQ?` funciona para átomos simbólicos, mas não necessariamente para átomos numéricos. O predicado = funciona para átomos numéricos, mas não para átomos simbólicos. Como discutido anteriormente, `EQ?` também não funciona de maneira confiável para parâmetros do tipo lista.

Algumas vezes, é conveniente testar a igualdade de dois átomos quando não se sabe se eles são simbólicos ou numéricos. Para esse propósito, Scheme tem um predicado diferente, `EQV?`, que funciona tanto em átomos numéricos quanto em átomos simbólicos. Considere os exemplos a seguir:

`(EQV? 'A 'A)` retorna `#T`
`(EQV? 'A 'B)` retorna `#F`
`(EQV? 3 3)` retorna `#T`
`(EQV? 'A 3)` retorna `#F`
`(EQV? 3.4 (+ 3 0.4))` retorna `#T`
`(EQV? 3.0 3)` retorna `#F`

Note que o último exemplo demonstra que valores de ponto flutuante são diferentes de valores inteiros. `EQV?` não é uma comparação de ponteiros – é uma comparação de valores.

A principal razão para se usar `EQ?` ou = em vez de `EQV?`, quando possível, é que as duas primeiras são mais rápidas.

A função de predicado `LIST?` retorna `#T` se seu único argumento é uma lista, e `#F` em caso contrário, como nos exemplos a seguir:

`(LIST? '(X Y))` retorna `#T`
`(LIST? 'X)` retorna `#F`
`(LIST? '())` retorna `#T`

A função `NULL?` testa seu parâmetro para determinar se é a lista vazia e, se for, retorna `#T`. Considere os exemplos a seguir:

`(NULL? '(A B))` retorna `#F`
`(NULL? '())` retorna `#T`
`(NULL? 'A)` retorna `#F`
`(NULL? '(()))` retorna `#F`

A última chamada leva a `#F` porque o parâmetro não é a lista vazia. Em vez disso, é uma lista contendo um elemento, a lista vazia.

15.5.10 Exemplos de funções em Scheme

Esta seção contém diversos exemplos de definições de funções em Scheme. Esses programas resolvem problemas simples de processamento de listas.

Considere o problema de pertinência de um átomo em uma lista que não inclui sublistas. Tal lista é chamada de **lista simples**. Se a função fosse denominada `member`, ela poderia ser usada como segue:

```
(member 'B '(A B C)) retorna #T
(member 'B '(A C D E)) retorna #F
```

Em termos de iteração, o problema da pertinência é simplesmente comparar o átomo e os elementos individuais da lista, um de cada vez, em alguma ordem, até um casamento ser encontrado, ou até não existirem mais elementos na lista a serem comparados. Um processo similar pode ser realizado com recursão. A função pode comparar o átomo com o `CAR` da lista. Se eles casam, o valor `#T` é retornado. Se não, o `CAR` da lista deve ser ignorado e a busca deve continuar no `CDR` da lista. Isso pode ser feito da seguinte maneira: a função chama a ela mesma com o `CDR` como parâmetro de lista e retorna o resultado dessa chamada recursiva. Esse processo terminará se o átomo for encontrado na lista. Se ele não estiver na lista, a função será por fim chamada (recursivamente) com uma lista nula como parâmetro real. Esse evento deve forçar a função a retornar `#F`. Nesse processo, existem duas saídas da recursão: ou a lista está vazia em alguma chamada, no caso em que é retornado `#F`, ou é encontrado um casamento e `#T` é retornado.

Ao todo, existem três casos que devem ser tratados na função: uma entrada de lista vazia, um casamento entre o átomo e o `CAR` da lista ou uma diferença entre o átomo e o `CAR` da lista, o que causa a chamada recursiva. Esses são os três parâmetros para `COND`, e o último é o caso padrão, disparado por um predicado `ELSE`. A função completa é a seguinte:[5]

```
(DEFINE (member atm a_list)
  (COND
    ((NULL? a_list) #F)
    ((EQ? atm (CAR a_list)) #T)
    (ELSE (member atm (CDR a_list)))
))
```

Essa forma é típica de funções simples de processamento de listas em Scheme. Em tais funções, os dados da lista são processados um elemento de cada vez. Os elementos individuais são especificados com `CAR` e o processo prossegue usando recursão no `CDR` da lista.

Note que o teste de nulo deve preceder o teste de igualdade, porque aplicar `CAR` a uma lista vazia é um erro.

[5] A maioria dos sistemas Scheme define uma função chamada `member` e não permite que o usuário a redefina. Assim, se o leitor quiser testar essa função, ela deverá ser definida com algum outro nome.

Como outro exemplo, considere o problema de determinar se duas listas são iguais. Caso as duas listas sejam simples, a solução é relativamente fácil, apesar de envolver algumas técnicas de programação com as quais o leitor pode não estar familiarizado. Uma função de predicado, `equalsimp`, para comparar listas simples, é mostrada aqui:

```
(DEFINE (equalsimp list1 list2)
  (COND
    ((NULL? list1) (NULL? list2))
    ((NULL? list2) #F)
    ((EQ? (CAR list1) (CAR list2))
          (equalsimp (CDR list1) (CDR list2)))
    (ELSE #F)
))
```

O primeiro caso, tratado pelo primeiro parâmetro de `COND`, é para quando o primeiro parâmetro de lista é a lista vazia. Isso pode ocorrer em uma chamada externa se o primeiro parâmetro da lista for inicialmente vazio. Como uma chamada recursiva usa os `CDR`s das duas listas parametrizadas como seus parâmetros, o primeiro parâmetro da lista pode estar vazio em tal chamada (se o primeiro parâmetro da lista agora estiver vazio). Quando o primeiro parâmetro da lista está vazio, também é necessário verificar se o segundo está vazio. Se estiver, eles são iguais (seja inicialmente ou se os `CAR`s eram iguais em todas as chamadas recursivas anteriores) e `NULL?`, corretamente, retorna `#T`. Se o segundo parâmetro da lista não está vazio, ele é maior que o primeiro parâmetro da lista e `#F` deve ser retornado, como ocorre usando `NULL?`.

O próximo caso lida com o fato de a segunda lista estar vazia quando a primeira não está. Essa situação só ocorre quando a primeira lista não é maior que a segunda. Apenas a segunda lista deve ser testada, porque o primeiro caso captura todas as instâncias de a primeira lista estar vazia.

O terceiro caso é o passo recursivo que testa a igualdade de dois elementos correspondentes nas duas listas. Isso é feito comparando-se os `CAR`s das duas listas não vazias. Se eles forem iguais, as duas listas são iguais até esse ponto; então, a recursão é usada nos `CDR`s das duas. Esse caso falha quando são encontrados dois átomos desiguais. Quando isso ocorre, o processo não precisa continuar; então, o caso padrão `ELSE` é selecionado, o qual retorna `#F`.

Note que `equalsimp` espera listas como parâmetros e não funciona corretamente se um ou ambos os parâmetros são átomos.

O problema de comparar listas gerais é um pouco mais complexo, porque as sublistas podem ser rastreadas completamente no processo de comparação. Nessa situação, a recursão é especificamente apropriada, porque a forma das sublistas é a mesma das listas. Sempre que os elementos correspondentes de duas listas são listas, eles são separados em suas duas partes, `CAR` e `CDR`, e a recursão é usada nelas. Esse é um exemplo perfeito da utilidade da estratégia de dividir-e-conquistar. Se os elementos correspondentes das duas listas são átomos, eles podem ser simplesmente comparados com `EQ?`.

A definição da função completa é:

```
(DEFINE (equal list1 list2)
  (COND
    ((NOT (LIST? list1)) (EQ? list1 list2))
    ((NOT (LIST? list2)) #F)
    ((NULL? list1) (NULL? list2))
    ((NULL? list2) #F)
    ((equal (CAR list1) (CAR list2))
            (equal (CDR list1) (CDR list2)))
    (ELSE #F)
))
```

Os dois primeiros casos de `COND` tratam da situação na qual qualquer um dos parâmetros é um átomo, em vez de uma lista. O terceiro e o quarto casos são para a situação em que uma ou ambas as listas são vazias. Esses casos também evitam que casos subsequentes tentem aplicar `CAR` a uma lista vazia. O quinto caso de `COND` é o mais interessante. O predicado é uma chamada recursiva, com os `CAR`s das listas como parâmetros. Se essa chamada recursiva retorna `#T`, a recursão é usada mais uma vez nos `CDR`s das listas. Esse algoritmo permite que as duas listas incluam sublistas de qualquer profundidade.

A definição de `equal` funciona em qualquer par de expressões, não apenas em listas. `equal` é equivalente à função de predicado de sistema `EQUAL?` Note que `EQUAL?` deve ser usada apenas quando necessário (as formas dos parâmetros reais não são conhecidas), porque é muito mais lenta que `EQ?` e `EQV?`

Outra operação de listas comumente necessária é aquela para construir uma nova lista que contenha todos os elementos de duas listas passadas como argumentos. Isso é normalmente implementado como uma função Scheme chamada `append`. A lista resultante pode ser construída na segunda lista passada como argumento, que se torna a lista resultante. Para entender a ação de `append`, considere os exemplos a seguir:

`(append '(A B) '(C D R))` retorna `(A B C D R)`
`(append '((A B) C) '(D (E F)))` retorna `((A B) C D (E F))`

A definição de `append` é[6]

```
(DEFINE (append list1 list2)
  (COND
    ((NULL? list1) list2)
    (ELSE (CONS (CAR list1) (append (CDR list1) list2)))
))
```

O primeiro caso de `COND` é usado para terminar o processo recursivo quando a primeira lista passada como argumento é vazia, retornando a segunda lista. No segundo caso (o

[6]Como acontecia com `member`, um usuário normalmente não pode definir uma função chamada `append`.

ELSE), o CAR da primeira lista passada como parâmetro é construído (usando CONS) com o resultado retornado pela chamada recursiva, a qual passa o CDR da primeira lista como seu primeiro parâmetro.

Considere a seguinte função Scheme, chamada guess, que usa a função member descrita nesta seção. Tente determinar o que ela faz antes de ler a descrição que a segue. Suponha que os parâmetros são listas simples.

```
(DEFINE (guess list1 list2)
  (COND
    ((NULL? list1) '())
    ((member (CAR list1) list2)
             (CONS (CAR list1) (guess (CDR list1) list2)))
    (ELSE (guess (CDR list1) list2))
))
```

A função guess retorna uma lista simples que contém os elementos comuns às duas listas passadas como parâmetro. Se as listas representam conjuntos, guess computa uma lista que representa a interseção desses dois conjuntos.

15.5.11 LET

LET é uma função (inicialmente descrita no Capítulo 5) que cria um escopo local no qual os nomes são temporariamente vinculados a valores de expressões. Ela é geralmente usada para fatorar subexpressões comuns de expressões mais complicadas. Esses nomes podem, então, ser usados na avaliação de outra expressão, mas não podem ser revinculados a novos valores em LET. O seguinte exemplo ilustra o uso de LET. Ele calcula as raízes de uma equação quadrática dada, supondo que as raízes sejam reais.[7] As definições matemáticas das raízes reais (em contraste com complexas) da equação quadrática $ax^2 + bx + c$ são as seguintes: root1 = (–b + sqrt(b2 – 4 ac))/2a e root 2 = (–b – sqrt(b2 – 4 ac))/2 a

```
(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a
                (/ (SQRT (- (* b b) (* 4 a c))) (* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))
        )
  (LIST (+ minus_b_over_2a root_part_over_2a)
            (- minus_b_over_2a root_part_over_2a))
))
```

Esse exemplo usa LIST para criar a lista de dois valores que geram o resultado.

Como os nomes vinculados na primeira parte de uma construção LET não podem ser alterados na expressão seguinte, eles não são iguais às variáveis locais de um bloco

[7] Algumas versões de Scheme incluem "complex" como tipo de dados e calculam as raízes da equação independentemente de serem reais ou complexas.

em uma linguagem imperativa. Todos eles poderiam ser eliminados pela substituição textual de suas respectivas expressões por seus nomes na expressão LET.

LET é, na verdade, uma forma abreviada para uma expressão LAMBDA aplicada a um parâmetro. As duas expressões a seguir são equivalentes:

```
(LET ((alpha 7))(* 5 alpha))
((LAMBDA (alpha) (* 5 alpha)) 7)
```

Na primeira expressão, 7 é vinculado a alpha com LET; na segunda, 7 é vinculado a alpha por meio do parâmetro da expressão LAMBDA.

15.5.12 Recursão em cauda em Scheme

Uma função é **recursiva em cauda** se sua chamada recursiva é a última operação nela. Isso significa que o valor de retorno da chamada recursiva é o valor de retorno da chamada não recursiva à função. Por exemplo, a função member da Seção 15.5.10, repetida aqui, é recursiva em cauda.

```
(DEFINE (member atm a_list)
  (COND
    ((NULL? a_list) #F)
    ((EQ? atm (CAR a_list)) #T)
    (ELSE (member atm (CDR a_list)))
))
```

Essa função pode ser automaticamente convertida por um compilador para usar iteração, o que resulta em uma execução mais rápida do que a de sua forma recursiva.

Entretanto, muitas funções que usam recursão para repetição não são recursivas em cauda. Os programadores que se preocupavam com a eficiência descobriram maneiras de reescrever algumas dessas funções de forma que fossem recursivas em cauda. Um exemplo disso é o uso de um parâmetro acumulador e uma função auxiliar. Como exemplo dessa estratégia, considere a função factorial da Seção 15.5.7, repetida aqui.

```
(DEFINE (factorial n)
  (IF (<= n 1)
    1
    (* n (factorial (- n 1)))
))
```

A última operação dessa função é a multiplicação. A função trabalha criando a lista de números a serem multiplicados e então faz as multiplicações à medida que a recursão se desenrola, para produzir o resultado. Cada um desses números é criado por uma ativação da função e cada um é armazenado em uma instância de registro de avaliação. À medida que a recursão se desenrola, os números são multiplicados. Lembre-se de que, no Capítulo 9, a pilha é mostrada após várias chamadas recursivas a factorial. Essa função factorial pode ser reescrita com uma função auxiliar que usa um parâmetro para acumular

o fatorial parcial. A função auxiliar, recursiva em cauda, também recebe o parâmetro de `factorial`. Essas funções são:

```
(DEFINE (facthelper n factpartial)
  (IF (<= n 1)
    factpartial
    (facthelper (- n 1) (* n factpartial))
))
(DEFINE (factorial n)
  (facthelper n 1)
)
```

Com essas funções, o resultado é calculado durante as chamadas recursivas, em vez de à medida que a recursão se desenrola. Como não há nada de útil nas instâncias de registro de avaliação, elas não são necessárias. Independentemente de quantas chamadas recursivas são solicitadas, apenas uma instância de registro de avaliação é necessária. Isso torna a versão recursiva em cauda bem mais eficiente que a versão recursiva sem cauda.

A definição da linguagem Scheme requer que os sistemas de processamento de linguagem Scheme convertam todas as funções recursivas em cauda de forma a substituir tal recursão por iteração. Logo, é importante, ao menos por questões de eficiência, definir funções que usem recursão para especificar repetição de forma que elas sejam recursivas em cauda. Determinadas otimizações de compiladores para algumas linguagens funcionais podem até mesmo realizar conversões de algumas funções não recursivas em cauda para funções recursivas em cauda equivalentes e, então, codificar tais funções de forma a usarem iteração, em vez de recursão, para repetição.

15.5.13 Formas funcionais

Esta seção descreve duas formas funcionais matemáticas comuns fornecidas por Scheme: composição e aplicar-a-todos. Ambas são matematicamente definidas na Seção 15.2.2.

15.5.13.1 Composição funcional

A composição funcional é a única forma funcional primitiva fornecida por Lisp original. Todos os dialetos subsequentes de Lisp, incluindo Scheme, também a fornecem. Conforme mencionado na Seção 15.2.2, composição funcional é uma forma funcional que recebe duas funções como parâmetros e retorna uma função que primeiramente aplica a segunda função parametrizada ao seu parâmetro e, então, aplica a primeira função parametrizada ao valor de retorno da segunda função parametrizada. Em outras palavras, a função `h` é a função de composição de `f` e `g` se `h(x) = f(g(x))`. Por exemplo, considere o seguinte exemplo:

```
(DEFINE (g x) (* 3 x))
(DEFINE (f x) (+ 2 x))
```

Agora a composição funcional de `f` e `g` pode ser escrita como segue:

```
(DEFINE (h x) (+ 2 (* 3 x)))
```

Em Scheme, a função de composição funcional `compose` pode ser escrita como segue:

```
(DEFINE (compose f g) (LAMBDA (x)(f (g x))))
```

Por exemplo, poderíamos ter o seguinte:

```
((compose CAR CDR) '((a b) c d))
```

Essa chamada produziria `c`. Ela é uma forma alternativa, embora menos eficiente, de `CADR`. Agora, considere outra chamada a `compose`:

```
((compose CDR CAR) '((a b) c d))
```

Essa chamada produziria `(b)`. Ela é uma alternativa à `CDAR`.

Como outro exemplo do uso de `compose`, considere o seguinte:

```
(DEFINE (third a_list)
  ((compose CAR (compose CDR CDR)) a_list))
```

Essa é uma alternativa a `CADDR`.

15.5.13.2 Uma forma funcional aplicar-a-todos

As formas funcionais mais comuns fornecidas em linguagens de programação funcional são variações das formas matemáticas funcionais aplicar-a-todos. As mais simples delas é `map`, que tem dois parâmetros: uma função e uma lista. `map` aplica a função dada a cada elemento da lista dada e retorna uma lista do resultado dessas aplicações. Uma definição de `map` em Scheme é mostrada a seguir:[8]

```
(DEFINE (map fun a_list)
  (COND
    ((NULL? a_list) '())
    (ELSE (CONS (fun (CAR a_list)) (map fun (CDR a_list))))
))
```

Note a forma simples de `map`, que expressa uma forma funcional complexa.

Como exemplo do uso de `map`, suponha que quiséssemos que todos os elementos de uma lista fossem elevados ao cubo. Podemos realizar isso com:

```
(map (LAMBDA (num) (* num num num)) '(3 4 2 6))
```

[8] Como era o caso com member, `map` é uma função predefinida que não pode ser redefinida pelos usuários.

Essa chamada retorna `(27 64 8 216)`.

Note que, nesse exemplo, o primeiro parâmetro para `map` é uma expressão `LAMBDA`. Quando `EVAL` avalia a expressão `LAMBDA`, ela constrói uma função com a mesma forma de qualquer função predefinida, mas sem um nome. Na expressão de exemplo, essa função sem nome é imediatamente aplicada a cada elemento da lista de parâmetros e os resultados são retornados em uma lista.

15.5.14 Funções que constroem código

O fato de programas e dados possuírem a mesma estrutura pode ser explorado na construção de programas. Lembre-se de que o interpretador Scheme usa uma função chamada `EVAL`. O sistema Scheme aplica `EVAL` a cada expressão digitada, seja no prompt de Scheme, no interpretador interativo ou como parte de um programa interpretado. A função `EVAL` também pode ser chamada diretamente por programas Scheme, o que possibilita que um programa crie expressões e chame `EVAL` para avaliá-las. Isso não é exclusividade de Scheme, mas as formas simples de suas expressões facilitam criá-las durante a execução.

Um dos exemplos mais simples desse processo envolve átomos numéricos. Lembre-se de que Scheme inclui uma função chamada +, que recebe qualquer quantidade de átomos numéricos como argumentos e retorna sua soma. Por exemplo, `(+ 3 7 10 2)` retorna `22`.

Nosso problema é o seguinte: suponha que em um programa tivéssemos uma lista de átomos numéricos e precisássemos da soma. Não podemos aplicar + diretamente na lista, porque essa função pode ser aplicada apenas a parâmetros atômicos e não a uma lista de átomos numéricos. Poderíamos, é claro, escrever uma função que repetidamente adicionasse o `CAR` da lista à soma por meio de seu `CDR`, usando recursão para percorrer a lista. Tal função é mostrada a seguir:

```
(DEFINE (adder a_list)
  (COND
    ((NULL? a_list) 0)
    (ELSE (+ (CAR a_list) (adder (CDR a_list
))
```

A seguir está um exemplo de chamada a `adder`, junto com as chamadas recursivas e os retornos:

```
(adder '(3 4 5))
(+ 3 (adder (4 5)))
(+ 3 (+ 4 (adder (5))))
(+ 3 (+ 4 (+ 5 (adder ()))))
(+ 3 (+ 4 (+ 5 0)))
(+ 3 (+ 4 5))
(+ 3 9)
(12)
```

Uma solução alternativa para o problema é escrever uma função que construa uma chamada a + com as formas parametrizadas corretas. Isso pode ser feito utilizando-se CONS para construir uma nova lista idêntica à lista de parâmetros, mas com o átomo + inserido no início. Essa nova lista pode então ser submetida a EVAL para avaliação, como segue:

```
(DEFINE (adder a_list)
  (COND
    ((NULL? a_list) 0)
    (ELSE (EVAL (CONS '+ a_list)))
))
```

Note que o nome da função + aparece com uma aspa para impedir que EVAL o considere na avaliação de CONS. A seguir está um exemplo de chamada a essa nova versão de adder, junto com a chamada a EVAL e o valor de retorno:

```
(adder '(3 4 5))
(EVAL (+ 3 4 5)
(12)
```

Em todas as versões anteriores de Scheme, a função EVAL avaliava sua expressão no escopo mais externo do programa. As versões mais recentes da linguagem, desde Scheme 4, exigem um segundo parâmetro para EVAL que especifica o escopo no qual a expressão será avaliada. Para simplificar, omitimos o parâmetro de escopo em nosso exemplo – não discutimos nomes de escopo aqui.

15.6 COMMON LISP

Common Lisp (Steele, 1990) é o resultado de um esforço para combinar os recursos de diversos dialetos anteriores de Lisp, incluindo Scheme, em uma única linguagem. Já que consiste em uma espécie de união de linguagens, ela é muito extensa e complexa; similar, nesse sentido, a C++ e C#. Sua base, entretanto, é Lisp original; portanto, sua sintaxe, funções primitivas e natureza fundamental vêm dessa linguagem.

A seguir está a função factorial escrita em Common Lisp:

```
(DEFUN factorial (x)
  (IF (<= n 1)
    1
    (* n factorial (- n 1)))
))
```

Somente a primeira linha dessa função difere sintaticamente de sua versão em Scheme.

A lista de recursos de Common Lisp é longa: um grande número de tipos e estruturas de dados, incluindo registros, matrizes, números complexos e cadeias de caracteres; operações de entrada e saída muito eficientes; e uma forma de pacotes para modular coleções de funções e de dados, e também para fornecer controle de acesso. Common Lisp inclui várias construções imperativas e alguns tipos mutáveis.

Reconhecendo a flexibilidade ocasionalmente fornecida pelo escopo dinâmico, assim como a simplicidade do escopo estático, Common Lisp permite ambos. O escopo padrão para variáveis é estático, mas ao se declarar uma variável como "especial" (special), o escopo dela se torna dinâmico.

Macros são frequentemente usadas em Common Lisp para estender a linguagem. Na verdade, algumas das funções predefinidas são macros. Por exemplo, `DOLIST`, que recebe dois parâmetros, uma variável e uma lista, é uma macro. Considere o seguinte:

```
(DOLIST (x '(1 2 3)) (print x))
```

Isso produz:

```
1
2
3
NIL
```

Aqui, `NIL` é o valor de retorno de `DOLIST`.

As macros geram seus efeitos em duas etapas: primeiro, a macro é expandida. Segundo, a macro expandida, que é código Lisp, é avaliada. Os usuários podem definir suas próprias macros com `DEFMACRO`.

O operador crase (`) de Common Lisp é semelhante ao `QUOTE` de Scheme, exceto que algumas partes do parâmetro podem não ser citadas, precedendo-as com vírgulas. Por exemplo, considere os dois exemplos a seguir:

```
'(a (* 3 4) c)
```

Essa expressão é avaliada como `(a (* 3 4) c)`. Entretanto, a expressão a seguir:

```
'(a ,(* 3 4) c)
```

é avaliada como `(a 12 c)`.

As implementações de Lisp têm um *front end* chamado de *leitor* que transforma o texto de programas Lisp em uma representação em código. Então, as chamadas a macros na representação em código são expandidas nas representações em código. Então, a saída dessa etapa é interpretada ou compilada na linguagem de máquina do computador hospedeiro ou, talvez, em um código intermediário que pode ser interpretado. Existem tipos especiais de macros, chamados *macros de leitor* ou *macros de leitura*, que são expandidos durante a fase de leitor de um processador de linguagem Lisp. Uma macro de leitor expande um caractere específico em uma cadeia de código Lisp. Por exemplo, o apóstrofo em Lisp é uma macro de leitura que se expande em uma chamada a `QUOTE`. Os usuários podem definir suas próprias macros de leitura para criar outras construções abreviadas.

Common Lisp, assim como outras linguagens baseadas em Lisp, tem um tipo de dados símbolo. As palavras reservadas são símbolos que avaliam a si mesmos, como `T` e `NIL`. Tecnicamente, os símbolos são vinculados ou desvinculados. Símbolos parametrizados são vinculados enquanto a função é avaliada. Além disso, os símbolos que

são nomes de variáveis de estilo imperativo e receberam valores são vinculados. Outros símbolos são desvinculados. Por exemplo, considere a expressão:

```
(LIST '(A B C))
```

Os símbolos `A`, `B` e `C` são desvinculados. Lembre-se de que Ruby também tem um tipo de dados símbolo.

Em certo sentido, Scheme e Common Lisp são opostos. Scheme é bem menor e semanticamente mais simples, em parte devido ao seu uso exclusivo de escopo estático, mas também porque foi projetada para ser usada no ensino de programação, enquanto Common Lisp tinha o objetivo de ser uma linguagem comercial. Common Lisp conseguiu ser uma linguagem bastante utilizada para aplicações de IA, entre outras áreas. Scheme, por outro lado, é mais usada em faculdades para o ensino de programação funcional. É mais provável que ela seja estudada como uma linguagem funcional por ser pequena. Um objetivo de projeto importante de Common Lisp, que a transformou em uma linguagem muito extensa, foi o desejo de torná-la compatível com diversos dos dialetos de Lisp dos quais foi derivada.

Common Lisp Object System (CLOS) (Paepeke, 1993) foi desenvolvido, no final dos anos 1980, como uma versão de Common Lisp orientada a objetos. Essa linguagem suporta funções genéricas e herança múltipla, entre outras construções.

15.7 ML

ML (Milner et al., 1997) é uma linguagem de programação funcional com escopo estático, como Scheme. Entretanto, ela difere de Lisp e de seus dialetos, incluindo Scheme, de diversas maneiras significativas. Uma diferença importante é o fato de ML ser uma linguagem fortemente tipada, enquanto Scheme é essencialmente desprovida de tipos. ML tem declarações de tipo para os parâmetros de função e para os tipos de retorno de funções, embora muitas vezes não sejam usadas, devido à sua inferência de tipos. O tipo de cada variável e expressão pode ser determinado estaticamente. ML, como outras linguagens de programação funcional, não tem variáveis, no sentido das linguagens imperativas. Ela tem identificadores, os quais têm a aparência de nomes de variáveis das linguagens imperativas. Contudo, é melhor considerar esses identificadores como nomes de valores. Uma vez definidos, eles não podem ser alterados. São como as constantes nomeadas das linguagens imperativas, como declarações **`final`** em Java. Os identificadores de ML não têm tipos fixos – qualquer identificador pode ser o nome de um valor de qualquer tipo.

Uma tabela chamada **ambiente de avaliação** armazena os nomes de todos os identificadores declarados implícita e explicitamente em um programa, junto com seus tipos. É como uma tabela de símbolos de tempo de execução. Quando um identificador é declarado, implícita ou explicitamente, ele é colocado no ambiente de avaliação.

Outra diferença importante entre Scheme e ML é que esta usa uma sintaxe mais relacionada a uma linguagem imperativa do que a de Lisp. Por exemplo, expressões aritméticas são escritas em ML com a notação infixa (a mais usada em linguagens imperativas).

Declarações de funções em ML aparecem na forma geral

```
fun nome_da_função(parâmetros formais) = expressão;
```

Quando chamada, o valor da expressão é retornado pela função. Na verdade, a expressão pode ser uma lista de expressões, separadas por pontos e vírgulas e circundadas por parênteses. Nesse caso, o valor de retorno é o da última expressão. Evidentemente, a menos que tenham efeitos colaterais, as expressões anteriores à última não têm nenhuma finalidade. Como não estamos considerando as partes de ML que têm efeitos colaterais, examinamos apenas as definições de função com uma única expressão.

Agora podemos discutir a inferência de tipos. Considere a seguinte declaração de função ML:

```
fun circumf(r) = 3.14159 * r * r;
```

Isso especifica uma função chamada `circumf` que recebe um parâmetro de ponto flutuante (**real** em ML) e produz um resultado em ponto flutuante. Os tipos são inferidos pelo tipo do literal na expressão. Da mesma forma, na função

```
fun times10(x) = 10 * x;
```

o parâmetro e o valor funcional são inferidos como sendo do tipo **int**.

Considere a seguinte função ML:

```
fun square(x) = x * x;
```

ML determina o tipo tanto do parâmetro quanto do valor de retorno a partir do operador `*` na definição da função. Como ele é um operador aritmético, presume-se que o tipo do parâmetro e o tipo da função sejam numéricos. Em ML, o tipo numérico padrão é **int**. Então, infere-se que o tipo do parâmetro e do valor de retorno de `square` seja **int**.

Se `square` fosse chamada com um valor de ponto flutuante, como em

```
square(2.75);
```

isso causaria um erro, porque ML não realiza coerção de valores do tipo **real** para o tipo **int**. Se quiséssemos que `square` aceitasse parâmetros do tipo **real**, ela poderia ser reescrita como

```
fun square(x) : real = x * x;
```

Como ML não permite funções sobrecarregadas, essa versão não poderia coexistir com a versão anterior baseada em **int**. A última versão definida seria a única.

O fato de o valor funcional ser tipado como **real** é suficiente para inferir que o parâmetro também é do tipo **real**. Cada uma das definições a seguir também é válida:

```
fun square(x : real) = x * x;
fun square(x) = (x : real) * x;
fun square(x) = x * (x : real);
```

A inferência de tipos também é usada nas linguagens funcionais Miranda, Haskell e F#.

A construção de fluxo de controle de seleção de ML é semelhante à das linguagens imperativas. Ela tem a seguinte forma geral:

if expressão **then** expressão_então **else** expressão_senão

A primeira expressão deve avaliar para um valor booleano.

Em ML, as expressões condicionais de Scheme podem aparecer no nível de definição de função. Em Scheme, a função `COND` é usada para determinar o valor de um parâmetro que, por sua vez, especifica o valor retornado por `COND`. Em ML, a computação realizada por uma função pode ser definida para formas diferentes de um parâmetro. Esse recurso visa a imitar a forma e o significado de definições funcionais condicionais da matemática. Em ML, a expressão específica que define o valor de retorno de uma função é escolhida pelo casamento de padrões em relação ao parâmetro dado. Por exemplo, sem usar esse casamento de padrões, uma função para computar o fatorial poderia ser escrita como segue:

```
fun fact(n : int): int = if n <= 1 then 1
                         else n * fact(n - 1);
```

Múltiplas definições de uma função podem ser escritas com casamento de padrões de parâmetros. As definições de função diferentes que dependem da forma dos parâmetros são separadas por um símbolo OR (`|`). Por exemplo, usando casamento de padrões, a função fatorial poderia ser escrita como segue:

```
fun fact(0) = 1
|   fact(1) = 1
|  fact(n : int): int = n * fact(n - 1);
```

Se `fact` é chamada com o parâmetro real `0`, a primeira definição é usada; se o parâmetro real é `1`, a segunda definição é usada; se é enviado um valor `int` que não é nem `0` nem `1`, a terceira definição é usada.

Como discutido no Capítulo 6, ML suporta listas e operações com listas. Lembre-se de que `hd`, `tl` e `::` são versões de ML de `CAR`, `CDR` e `CONS` de Scheme.

Devido à disponibilidade de parâmetros de funções baseados em padrões, as funções `hd` e `tl` são muito menos usadas em ML do que `CAR` e `CDR` em Scheme. Por exemplo, em um parâmetro formal, a expressão

```
(h :: t)
```

são dois parâmetros formais, na verdade – o primeiro elemento e o restante da lista passada como parâmetro dada –, enquanto o único parâmetro real correspondente é uma lista. Por exemplo, o número de elementos em uma lista pode ser computado com a função:

```
fun length([]) = 0
|   length(h :: t) = 1 + length(t);
```

Como outro exemplo desses conceitos, considere a função `append`, que faz o mesmo que a homônima de Scheme:

```
fun append([], lis2) = lis2
|   append(h :: t, lis2) = h :: append(t, lis2);
```

O primeiro caso dessa função trata da situação em que ela é chamada com uma lista vazia como seu primeiro parâmetro. Esse caso também termina a recursão quando a chamada inicial possui um primeiro parâmetro não vazio. O segundo caso da função quebra a lista passada como primeiro parâmetro em sua cabeça e cauda (`hd` e `tl`). O primeiro elemento é anexado no resultado da função recursiva com `CONS`, que usa a cauda da lista como seu primeiro parâmetro.

Em ML, os nomes são vinculados a valores com sentenças de declarações de valores na forma

`val` new_name = expression;

Por exemplo,

```
val distance = time * speed;
```

Essa sentença não é exatamente igual às de atribuição nas linguagens imperativas. A sentença **`val`** vincula um nome a um valor, mas o nome não pode ser posteriormente revinculado a um novo valor. Bem, de certa forma, pode. Na verdade, se você revincular um nome com uma segunda sentença **`val`**, é criada uma nova entrada no ambiente de avaliação, não relacionada à versão anterior do nome. Efetivamente, após a nova vinculação, a entrada de ambiente anterior (para a vinculação anterior) não é mais visível. Além disso, o tipo da nova vinculação não precisa ser o mesmo da anterior. Sentenças **`val`** não têm efeitos colaterais. Elas simplesmente adicionam um nome ao ambiente de avaliação atual e o vinculam a um valor.

O uso normal de **`val`** é em uma expressão **`let`**.[9] Considere o seguinte exemplo:

```
let val radius = 2.7
   val pi = 3.14159
in pi * radius * radius
end;
```

ML contém diversas funções de ordem superior, comumente usadas em programação funcional. Entre elas está uma função de filtragem para listas, `filter`, a qual recebe como parâmetro uma função de predicado. Muitas vezes, a função de predicado é fornecida como uma expressão lambda, que, em ML, é definida exatamente como uma função, exceto por ser usada a palavra reservada **`fn`**, em vez de **`fun`**, e pela expressão lambda não ter nome. `filter` retorna uma função que recebe uma lista como parâmetro. Ela testa cada elemento da lista com o predicado. Cada elemento no qual o predicado retorna `true` é adicionado a uma nova lista, a qual é o valor de retorno da função. Considere o seguinte uso de `filter`:

[9] Expressões `let` em ML foram apresentadas no Capítulo 5.

```
filter(fn(x) => x < 100, [25, 1, 50, 711, 100, 150, 27,
    161, 3]);
```

Essa aplicação retornaria `[25, 1, 50, 27, 3]`.

A função `map` recebe um único parâmetro, que é uma função. A função resultante recebe uma lista como parâmetro. Ela aplica sua função a cada elemento da lista e retorna uma lista dos resultados dessas aplicações. Considere o código a seguir:

```
fun cube x = x * x * x;
val cubeList = map cube;
val newList = cubeList [1, 3, 5];
```

Após a execução, o valor de `newList` é `[1, 27, 125]`. Isso poderia ser feito de forma mais simples, definindo-se a função cube como uma expressão lambda:

```
val newList = map (fn x => x * x * x, [1, 3, 5]);
```

ML tem um operador binário para compor duas funções, `o` (um "o" minúsculo). Por exemplo, para construirmos uma função `h` que primeiro aplica a função `f` e depois a função `g` ao valor retornado de `f`, poderíamos usar o seguinte:

```
val h = g o f;
```

Rigorosamente falando, as funções ML recebem um único parâmetro. Quando uma função é definida com mais de um parâmetro, ML os considera como uma tupla, mesmo que os parênteses que normalmente delimitam um valor de tupla sejam opcionais. As vírgulas que separam os parâmetros (os elementos da tupla) são obrigatórias.

O processo de **currying** substitui uma função com mais de um parâmetro por uma função com apenas um, a qual retorna uma função que recebe os outros parâmetros da função inicial.

As funções ML que recebem mais de um parâmetro podem ser definidas em forma de currying omitindo-se as vírgulas entre os parâmetros (e os parênteses delimitadores).[10] Por exemplo, poderíamos ter o seguinte:

```
fun add a b = a + b;
```

Embora isso pareça definir uma função com dois parâmetros, na verdade apenas um parâmetro é definido. A função `add` recebe um parâmetro inteiro (`a`) e retorna uma função que também recebe um parâmetro inteiro (`b`). Uma chamada a essa função também exclui as vírgulas entre os parâmetros, como no seguinte:

```
add 3 5;
```

Conforme o esperado, essa chamada a `add` retorna `8`.

[10] Essa forma de funções é assim chamada em homenagem a Haskell Curry, matemático britânico que a estudou.

As funções que sofreram currying são interessantes e úteis, pois novas funções podem ser construídas a partir delas por meio de avaliação parcial. **Avaliação parcial** significa que a função é avaliada com os parâmetros reais para um ou mais dos parâmetros formais mais à esquerda. Por exemplo, poderíamos definir uma nova função, como segue:

```
fun add5 x = add 5 x;
```

A função `add5` recebe o parâmetro real `5` e avalia a função `add` tendo `5` como valor de seu primeiro parâmetro formal. Ela retorna uma função que soma `5` ao seu parâmetro único, como segue:

```
val num = add5 10;
```

O valor de `num` agora é `15`. Poderíamos criar qualquer número de funções a partir da função que sofreu currying `add` para somar qualquer número específico a um parâmetro dado.

Funções que sofreram currying também podem ser escritas em Scheme, Haskell e F#. Considere a função Scheme a seguir:

```
(DEFINE (add x y) (+ x y))
```

Uma versão dela com currying seria a seguinte:

```
DEFINE (add y) (LAMBDA (x) (+ y x)))
```

Isso pode ser chamado como segue:

```
((add 3) 4)
```

ML tem tipos enumerados, matrizes e tuplas. Além disso, tem tratamento de exceções e um recurso de módulo para implementar tipos de dados abstratos.

Essa linguagem teve impacto significativo na evolução das linguagens de programação. Entre pesquisadores de linguagens, ela se tornou uma das mais estudadas. Além disso, gerou diversas linguagens subsequentes, entre elas Haskell, Caml, OCaml e F#.

15.8 HASKELL

Haskell (Thompson, 1999) é similar a ML por usar uma sintaxe semelhante, ter escopo estático, ser fortemente tipada e utilizar o mesmo método de inferência. Existem três características de Haskell que a diferenciam de ML. Primeiro, as funções em Haskell podem ser sobrecarregadas (as funções em ML, não). Segundo, semânticas não estritas são usadas em Haskell, enquanto em ML (e na maioria das outras linguagens de programação) são usadas semânticas estritas. Terceiro, Haskell é uma linguagem de programação funcional pura, ou seja, não tem expressões nem sentenças com efeitos colaterais, enquanto ML permite alguns efeitos colaterais (por exemplo, ML tem vetores mutáveis).

Tanto a semântica não estrita quanto a sobrecarga de funções são discutidas em mais detalhes posteriormente nesta seção.

O código desta seção é escrito na versão 1.4 de Haskell.

Considere a seguinte definição da função fatorial, que usa casamento de padrões em seus parâmetros:

```
fact 0 = 1
fact 1 = 1
fact n = n * fact (n - 1)
```

Note as diferenças sintáticas entre essa definição e sua versão em ML da Seção 15.7. Primeiro, não existe uma palavra reservada para introduzir a definição de função (**`fun`** em ML). Segundo, todas as definições alternativas de funções (com parâmetros formais diferentes) têm a mesma aparência.

Usando casamento de padrões, podemos definir uma função para computar o enésimo número de Fibonacci com o seguinte:

```
fib 0 = 1
fib 1 = 1
fib (n + 2) = fib (n + 1) + fib n
```

Guardas podem ser adicionadas às linhas da definição da função para especificar as circunstâncias nas quais a definição pode ser aplicada. Por exemplo,

```
fact n
  | n == 0 = 1
  | n == 1 = 1
  | n > 1 = n * fact(n - 1)
```

Essa definição de fatorial é mais precisa que a anterior, visto que restringe a faixa dos valores dos parâmetros reais àqueles com os quais ela funciona. Essa forma de definição de função é chamada de *expressão condicional*, denominação devida às expressões matemáticas em que que a definição de função é baseada.

Um **`otherwise`** (senão) pode aparecer como a última condição em uma expressão condicional, com a semântica óbvia. Por exemplo,

```
sub n
| n < 10 = 0
| n > 100 = 2
| otherwise = 1
```

Note a similaridade entre as guardas aqui e os comandos protegidos discutidos no Capítulo 8.

Considere a seguinte definição de função, cujo propósito é o mesmo da função ML correspondente da Seção 15.7:

```
square x = x * x
```

Nesse caso, entretanto, devido ao suporte de Haskell para polimorfismo, essa função pode receber um parâtmetro de qualquer tipo numérico.

Como em ML, as listas são escritas com colchetes em Haskell. Por exemplo:

```
colors = ["blue", "green", "red", "yellow"]
```

Haskell inclui uma coleção de operadores de lista. Por exemplo, listas podem ser concatenadas com ++, `:` serve como uma versão infixa de `CONS` e `..` é usado para especificar séries aritméticas em uma lista. Por exemplo,

`5:[2, 7, 9]` resulta em `[5, 2, 7, 9]`
`[1, 3..11]` resulta em `[1, 3, 5, 7, 9, 11]`
`[1, 3, 5] ++ [2, 4, 6]` resulta em `[1, 3, 5, 2, 4, 6]`

Observe que o operador `:` é exatamente como o operador `::` de ML.[11] Usando `:` e casamento de padrões, podemos definir uma função simples para computar o produto de uma lista de números dada:

```
product [] = 1
product (a:x) = a * product x
```

Usando `product`, podemos escrever uma função fatorial na forma mais simples

```
fact n = product [1..n]
```

Haskell inclui uma construção **`let`** similar a **`let`** e **`val`** de ML. Por exemplo, poderíamos escrever

```
quadratic_root a b c =
    let minus_b_over_2a = - b / (2.0 * a)
     root_part_over_2a =
                sqrt(b ^ 2 - 4.0 * a * c) / (2.0 * a)
    in
     minus_b_over_2a - root_part_over_2a,
     minus_b_over_2a + root_part_over_2a
```

As compreensões de lista de Haskell foram apresentadas no Capítulo 6. Por exemplo, considere o seguinte:

```
[n * n * n | n <- [1..50]]
```

Isso define uma lista dos cubos dos números de 1 a 50. É lido como "a lista de todos `n*n*n` tal que `n` é obtido da faixa de 1 a 50". Nesse caso, o qualificador está na forma de **gerador**. Ele gera os números de 1 a 50. Em outros casos, os qualificadores estão na forma de expressões booleanas e são chamados de **testes**. Essa notação pode ser usada

[11] É interessante notar que ML usa : para anexar um nome de tipo a um nome e `::` para CONS, enquanto Haskell usa esses dois operadores de maneiras exatamente opostas.

a fim de descrever algoritmos para fazer muitas coisas, como encontrar permutações de listas e ordená-las. Por exemplo, considere a função a seguir, que, quando dado um número `n`, retorna uma lista de todos os seus fatores:

```
factors n = [ i | i <-  [1..n `div` 2], n `mod` i == 0]
```

A compreensão de lista em `factors` cria uma lista de números, cada um temporariamente vinculado ao nome `i`, variando de `1` a `n/2`, tal que `n `mod` i` é zero. Essa é, na verdade, uma definição bastante minuciosa e curta dos fatores de um número. As crases envolvendo **`div`** e **mod** são usadas para especificar o uso infixo dessas funções. Quando são chamadas em notação funcional, como em **`div`**`n 2`, as crases não são usadas.

A seguir, considere a concisão de Haskell ilustrada na seguinte implementação do algoritmo quicksort (ordenação rápida):

```
sort [] = []
sort (h:t) = sort [b | b <- t, b <- h]
             ++ [h] ++
             sort [b | b <- t, b > h]
```

Nesse programa, o conjunto de elementos menores ou iguais ao primeiro elemento da lista é ordenado e concatenado com esse primeiro elemento; então, o conjunto de elementos maiores que o primeiro é ordenado e concatenado com os resultados anteriores. Essa definição de quicksort é significativamente menor e mais simples que o mesmo algoritmo codificado em uma linguagem imperativa.

Uma linguagem de programação é **estrita** se requer que todos os parâmetros reais sejam completamente avaliados, garantindo que o valor de uma função não dependa da ordem em que os parâmetros são avaliados. Uma linguagem é **não estrita** se não tem o requisito de ser estrita. Linguagens não estritas podem ter diversas vantagens em relação às linguagens estritas. Primeiramente, em geral, elas são mais eficientes, pois alguma avaliação é evitada. Segundo, permitem algumas capacidades interessantes que não são possíveis com linguagens estritas, entre elas as listas infinitas. Além disso, podem usar uma forma de avaliação chamada de **avaliação preguiçosa** (também conhecida como avaliação atrasada, adiada ou postergada), e isso significa que as expressões são avaliadas apenas se e quando seus valores forem necessários.

Lembre-se de que em Scheme os parâmetros para uma função são completamente avaliados antes de a função ser chamada; então, ela tem semântica estrita. Na avaliação preguiçosa, um parâmetro real é avaliado apenas quando seu valor é necessário para avaliar a função. Então, se uma função tem dois parâmetros, mas em uma de suas execuções o primeiro não é usado, o parâmetro real passado para tal execução não será avaliado.[12] Além disso, se apenas uma parte de um parâmetro real deve ser avaliada para uma execução da função, o resto é deixado sem avaliação. Por fim, os parâmetros reais

[12]Note como isso está relacionado com a avaliação em curto-circuito de expressões booleanas, feita em algumas linguagens imperativas.

são avaliados apenas uma vez, se é que o são, mesmo se esse parâmetro real aparecer mais de uma vez em uma chamada à função.

Conforme mencionado, a avaliação preguiçosa permite definir estruturas de dados infinitas. Por exemplo, considere o seguinte:

```
positives = [0..]
evens = [2, 4..]
squares = [n * n | n <- [0..]]
```

É claro, nenhum computador pode representar todos os números dessas listas, mas isso não impede seu uso se a avaliação preguiçosa for empregada. Por exemplo, se quiséssemos saber se determinado número seria um quadrado perfeito, poderíamos verificar a lista `squares` com uma função de pertinência. Suponha que tivéssemos uma função de predicado chamada `member` que determinasse se um átomo está contido em uma lista dada. Então, poderíamos usá-la como em

```
member 16 squares
```

que retornaria `True`. A definição de `squares` seria avaliada até que o `16` fosse encontrado. A função `member` precisaria ser cuidadosamente escrita. Especificamente, suponha que ela fosse definida como:

```
member b [] = False
member b (a:x)= (a == b) || member b x
```

A segunda linha dessa definição quebra o primeiro parâmetro em sua cabeça e cauda. Seu valor de retorno é verdadeiro se a cabeça casa com o elemento da busca (`b`) ou se a chamada recursiva com a cauda retorna `True`.

A definição de `member` funcionaria corretamente com `squares` apenas se o número fosse um quadrado perfeito. Se não, `squares` continuaria gerando quadrados infinitamente ou até ocorrer alguma limitação de memória, buscando o número na lista. A seguinte função realiza o teste de pertinência de uma lista ordenada, abandonando a busca e retornando `False` se um número maior que o número buscado for encontrado.[13]

```
member2 n (m:x)
  | m < n      = member2 n x
  | m == n     = True
  | otherwise = False
```

A avaliação preguiçosa algumas vezes fornece uma ferramenta de modularização. Suponha que em um programa exista uma chamada a uma função `f` e que o parâmetro para `f` seja o valor de retorno de uma função `g`.[14] Então, temos `f(g(x))`. Em seguida, suponha que `g` produza uma grande quantidade de dados, um pouco de cada vez, e que `f` deva processar esses dados, um pouco de cada vez. Em uma linguagem imperativa convencional, `g` executaria na entrada inteira, produzindo um arquivo de sua saída.

[13] Nesse caso, presume-se que a lista está em ordem ascendente.

[14] Esse exemplo aparece em Hughes (1989).

Então, `f` executaria usando o arquivo como entrada. Essa estratégia exige o tempo de escrita e leitura do arquivo, bem como seu armazenamento. Com avaliação preguiçosa, as execuções de `f` e `g` seriam, de forma implícita, extremamente sincronizadas. A função `g` executaria apenas o suficiente para produzir dados para `f` iniciar seu processamento. Quando `f` estivesse pronta para mais dados, `g` seria reiniciada para produzir mais, enquanto `f` esperaria. Se `f` terminasse sem obter toda a saída de `g`, `g` seria abortada, evitando computações inúteis. Além disso, `g` não precisaria ser uma função com término, talvez porque produzisse uma quantidade de saída infinita. A função `g` seria forçada a terminar quando `f` terminasse. Então, sob avaliação preguiçosa, `g` executaria o mínimo possível. Esse processo de avaliação suporta a modularização de programas em unidades geradoras e unidades seletoras, onde o gerador produz um grande número de resultados possíveis e o seletor escolhe o subconjunto apropriado.

No entanto, a avaliação preguiçosa não ocorre sem expensas. Seria certamente surpreendente se tal poder de expressão e tal flexibilidade não tivessem custos. Nesse caso, o custo é em uma semântica muito mais complicada, que resulta em uma velocidade de execução muito mais lenta.

15.9 F#

F# é uma linguagem de programação funcional .NET cujo núcleo é baseado em OCaml, que é uma descendente de ML e Haskell. Embora seja fundamentalmente uma linguagem funcional, ela contém recursos imperativos e suporta programação orientada a objetos. Uma das características mais importantes de F# é o fato de ter um IDE completo, uma ampla biblioteca de utilitários que suportam programação imperativa, orientada a objetos e funcional, além de interoperabilidade com uma coleção de linguagens não funcionais (todas as linguagens .NET).

F# é uma linguagem .NET de primeira classe. Isso significa que os seus programas podem interagir de todas as maneiras com outras linguagens .NET. Por exemplo, classes F# podem ser usadas e estendidas (por meio de especialização) por programas em outras linguagens e vice-versa. Além disso, os programas F# têm acesso a todas as APIs do Framework .NET. A implementação de F# está disponível gratuitamente no site da Microsoft (http://research.microsoft.com/fsharp/fsharp.aspx). Ela também é suportada pelo Visual Studio.

F# contém uma variedade de tipos de dados. Entre eles estão tuplas, como aquelas de Python e das linguagens funcionais ML e Haskell, listas, uniões discriminadas, uma expansão das uniões da ML e registros, como aqueles da ML, os quais são como tuplas, exceto por seus componentes serem nomeados. F# tem vetores mutáveis e imutáveis.

Lembre-se, do Capítulo 6, de que as listas de F# são semelhantes às de ML, exceto por seus elementos serem separados por pontos e vírgulas e `hd` e `tl` deverem ser chamados como métodos de List.

F# suporta valores em sequência, que são tipos do espaço de nomes .NET `System.Collections.Generic.IEnumerable`. Em F#, as sequências são abreviadas como `seq<type>`, em que `<type>` indica o tipo do genérico. Por exemplo, o tipo `seq<int>` é uma sequência de valores inteiros. Valores em sequência podem ser criados com gera-

dores e ser iterados. As sequências mais simples são geradas com expressões de faixa, como no exemplo a seguir:

```
let x = seq {1..4};;
```

Nos exemplos de F#, supomos que é usado o interpretador interativo, o qual exige os dois pontos e vírgulas no final de cada sentença. A expressão acima gera `seq[1; 2; 3; 4]`. (Elementos de lista e de sequência são separados por pontos e vírgulas.) A geração de uma sequência é preguiçosa; por exemplo, o seguinte exemplo define `y` como uma sequência muito longa, mas são gerados apenas os elementos necessários. Para exibição, são gerados somente os quatro primeiros.

```
let y = seq {0..100000000};;
y;;
val it: seq<int> = seq[0; 1; 2; 3;. . .]
```

A primeira linha define `y`; a segunda solicita a exibição do valor de `y`; a terceira é a saída do interpretador interativo de F#.

O tamanho do passo padrão para definições de sequência de inteiros é `1`, mas pode ser configurado pela sua inclusão no meio da especificação de faixa, como no exemplo a seguir:

```
seq {1..2..7};;
```

Isso gera `seq [1; 3; 5; 7]`.

Os valores de uma sequência podem ser iterados com uma construção **`for-in`**, como no exemplo a seguir:

```
let seq1 = seq {0..3..11};;
for value in seq1 do printfn "value = %d" value;;
```

Isso produz o seguinte:

```
value = 0
value = 3
value = 6
value = 9
```

Iteradores também podem ser usados para criar sequências, como no exemplo a seguir:

```
let cubes = seq {for i in 1..5 -> (i, i * i * i)};;
```

Isso gera a seguinte lista de tuplas:

```
seq [(1, 1); (2, 8); (3, 27); (4, 64); (5, 125)]
```

Esse uso de iteradores para gerar coleções é uma forma de compreensão de lista.

O sequenciamento também pode ser usado para gerar listas e vetores, embora nesses casos a geração não seja preguiçosa. De fato, a principal diferença entre listas e sequências em F# é que as sequências são preguiçosas e, assim, podem ser infinitas, enquanto as listas não são preguiçosas. Listas são armazenadas na memória em sua totalidade. Isso não acontece com as sequências.

As funções de F# são semelhantes às de ML e Haskell. Se forem nomeadas, são definidas com sentenças **let**. Se forem não nomeadas, o que tecnicamente significa que são expressões lambda, são definidas com a palavra reservada **fun**. A seguinte expressão lambda ilustra sua sintaxe:

```
(fun  a b -> a / b)
```

Note que não há nenhuma diferença entre um nome definido com **let** e uma função sem parâmetros definida com **let**.

A endentação é usada para mostrar a extensão de uma definição de função. Por exemplo, considere a seguinte definição de função:

```
let f =
    let pi = 3.14159
    let twoPi = 2.0 * pi
    twoPi;;
```

Note que F#, assim como ML, não faz a coerção de valores numéricos; portanto, se essa função usasse `2` na penúltima linha, em vez de `2.0`, seria informado um erro.

Se uma função é recursiva, a palavra reservada **rec** deve preceder seu nome em sua definição. A seguir está uma versão em F# de factorial:

```
let rec factorial x =
    if x <= 1  then 1
    else n * factorial(n - 1)
```

Nomes definidos em funções podem estar fora do escopo, o que significa que podem ser redefinidos, fato que termina seu escopo inicial. Por exemplo, poderíamos ter o seguinte:

```
let x4 x =
    let x = x * x
    let x = x * x
    x;;
```

Nessa função, o primeiro **let** no corpo da função `x4` cria uma versão de `x`, definindo-a para que tenha o valor do quadrado do parâmetro `x`. Isso termina o escopo do parâmetro. Assim, o segundo **let** no corpo da função usa o novo `x` em seu lado direito e cria ainda outra versão de `x`, terminando com isso o escopo do `x` criado no **let** anterior.

Existem dois operadores funcionais importantes em F#, pipeline (`|>`) e composição de função (`>>`). O operador de pipeline é um operador binário que envia o valor de seu

operando esquerdo, que é uma expressão, para o último parâmetro da chamada de função, que é o operando da direita. Ele é usado para encadear chamadas de função, ao passo que faz os dados que estão sendo processados fluírem para cada chamada. Considere o exemplo de código a seguir, o qual usa as funções de ordem superior `filter` e `map`:

```
let myNums = [1; 2; 3; 4; 5]
let evensTimesFive = myNums
    |> List.filter (fun  n -> n % 2 = 0)
    |> List.map (fun  n -> 5 * n)
```

A função `evensTimesFive` começa com a lista `myNums`, filtra os números que não formam pares com `filter` e usa `map` para mapear uma expressão lambda que multiplica por cinco os números de uma lista dada. O valor de retorno de `evensTimesFive` é `[10; 20]`.

O operador de composição de função constrói uma função que aplica seu operando esquerdo a determinado parâmetro, o qual é uma função, e então passa o resultado retornado dessa função para seu operando da direita, que também é uma função. Assim, a expressão em F# `(f >> g) x` é equivalente à expressão matemática `g(f(x))`.

Como ML, F# suporta funções com currying e avaliação parcial. O exemplo em ML da Seção 15.7 poderia ser escrito em F# como segue:

```
let add a b = a + b;;
let add5 = add 5;;
```

Note que, ao contrário de ML, a sintaxe da lista de parâmetros formais em F# é a mesma para todas as funções; portanto, todas as funções com mais de um parâmetro podem sofrer currying.

F# é interessante por várias razões. Primeiro, como uma linguagem funcional, complementa as linguagens funcionais anteriores. Segundo, suporta praticamente todas as metodologias de programação em uso atualmente. Terceiro, é a primeira linguagem funcional projetada para interoperabilidade com outras linguagens bastante usadas. Quarto, parte de um IDE elaborado e bem desenvolvido e de uma biblioteca de software utilitário com .NET e seu framework.

15.10 SUPORTE PARA PROGRAMAÇÃO FUNCIONAL EM LINGUAGENS BASICAMENTE IMPERATIVAS

Linguagens de programação imperativas normalmente fornecem apenas suporte limitado para programação funcional. Esse suporte resultou no pouco uso dessas linguagens para programação funcional. A restrição mais importante, relacionada à programação funcional, das primeiras linguagens imperativas era a ausência de suporte para funções de ordem superior.

Uma indicação do crescente interesse e uso de programação funcional é o suporte parcial para ela, surgido no decorrer da década passada, em linguagens basicamente

imperativas. Por exemplo, funções anônimas, que são como expressões lambda, agora fazem parte de JavaScript, Python, Ruby, Java e C#.

Em JavaScript, as funções nomeadas são definidas com a seguinte sintaxe:

```
function nome (parâmetros-formais) {
 corpo
}
```

Uma função anônima é definida em JavaScript com a mesma sintaxe, exceto pelo fato de o nome da função ser omitido.

Em C#, uma expressão lambda é uma instância de um representante. Ela pode ser anônima ou nomeada. Uma expressão lambda anônima é mais simples que um método anônimo, pois os métodos precisam definir seus tipos de parâmetro e o tipo de retorno, mas as expressões lambda usam o processo de inferência de C# para evitar essas necessidades. A sintaxe de uma expressão lambda sem nome em C# é a seguinte:

parâmetro(s) => expressão

Se houver mais de um parâmetro, eles devem ser colocados entre parênteses. Se não houver parâmetros, parênteses vazios devem aparecer no lugar deles. Se o sistema não puder inferir os tipos dos parâmetros, eles podem ser precedidos por nomes de tipo. O tipo do valor de retorno nunca é especificado; ele é sempre inferido pelo contexto da expressão lambda. A expressão é simples ou uma sentença composta colocada entre chaves. Tal sentença composta deve incluir uma sentença **`return`**.

Um uso comum para expressões lambda anônimas é como parâmetros reais para métodos especificados de modo a receber representantes como parâmetros. C# tem uma coleção de métodos para vetores que efetuam operações de busca. Por exemplo, o método `FindAll` localiza todos os elementos de um vetor que satisfazem determinada instância de um representante, o qual recebe um parâmetro do tipo dos elementos do vetor e retorna um valor booleano. Por exemplo, poderíamos ter o seguinte:

```
int[] numbers = {-3, 0, 4, 5, 1, 7, -3, -6, -9, 0, 3};
int[] positives = Array.FindAll(numbers, n => n > 0);
// Agora, positives é {4, 5, 1, 7, 3}
```

Em C#, expressões lambda também podem ser nomeadas. A linguagem tem representantes genéricos que simplificam a definição de tais expressões lambda, embora não englobem todas as possibilidades. Um representante genérico comumente utilizado é `Func`, que pode receber até 16 parâmetros genéricos para a expressão lambda, mais um para o tipo de retorno. Por exemplo, considere a seguinte expressão lambda nomeada e uma chamada a ela:

```
Func<int, int, int> eval1 = (a, b) => 3 * a + (b / 2);
int result = eval1(6, 22);
```

As expressões lambda C# podem acessar variáveis definidas fora de suas definições. Quando isso acontece, os tempos de vida das variáveis externas acessadas, denominadas

variáveis capturadas, são estendidos para que elas ainda existam quando a expressão lambda for usada. Uma expressão lambda que captura variáveis externas é um fechamento.

As expressões lambda foram adicionadas a Java 8. A sintaxe geral dessas expressões é como a de C#, exceto que é usado ->, em vez de =>. As sintaxes dos parâmetros, da inferência de tipos de parâmetro e do tipo de retorno e da expressão ou do bloco com um **`return`** são iguais às de C#. Antes de Java 8, não havia nenhuma maneira conveniente de passar um bloco de código para um método ou de fazer um método retornar um bloco de código.[15]

As expressões lambda de Python definem funções anônimas simples de apenas uma sentença. A sintaxe de uma expressão lambda em Python é exemplificada assim:

```
lambda a, b : 2 * a - b
```

Note que os parâmetros formais são separados do corpo da função por dois pontos.

Python contém as funções de ordem superior `filter` e `map`. Ambas frequentemente usam expressões lambda como primeiro parâmetro. O segundo parâmetro delas é um tipo sequência e ambas retornam o mesmo tipo sequência como segundo parâmetro. Em Python, cadeias, listas e tuplas são consideradas sequências. Considere o seguinte exemplo de uso da função `map` em Python:

```
map(lambda x: x ** 3, [2, 4, 6, 8])
```

Essa chamada retorna `[8 64 216 512]`.

Python também suporta aplicações de funções parciais. Considere o seguinte exemplo:

```
from operator import add
add5 = partial (add, 5)
```

Aqui, a declaração **`from`** importa a versão funcional do operador de adição (chamado `add`) do operador módulo.

Após a definição de `add5`, ele pode ser usado com um parâmetro, como no seguinte:

```
add5(15)
```

Essa chamada retorna `20`.

Como descrito no Capítulo 6, Python inclui listas e compreensões de lista.

Os blocos de Ruby são efetivamente subprogramas enviados para métodos, o que torna o método um subprograma de ordem superior. Um bloco de Ruby pode ser convertido em um objeto subprograma com **`lambda`**. Por exemplo, considere o seguinte:

```
times = lambda {|a, b| a * b}
```

[15] Um modo inconveniente era definir uma classe com um método que contivesse o código, instanciá-la e passar uma referência para ela.

A seguir está um exemplo de uso de `times`:

```
x = times.(3, 4)
```

Isso configura `x` com `12`. O objeto `times` pode sofrer currying com o seguinte:

```
times5 = times.curry.(5)
```

Essa função pode ser usada como segue:

```
x5 = times5.(3)
```

Isso configura `x5` com `15`.

15.11 UMA COMPARAÇÃO ENTRE LINGUAGENS FUNCIONAIS E IMPERATIVAS

Esta seção discute algumas das diferenças entre as linguagens imperativas e funcionais.

Linguagens funcionais podem ter uma estrutura sintática muito simples. A estrutura de lista de Lisp, usada tanto para código quanto para dados, ilustra claramente isso. A sintaxe das linguagens imperativas é muito mais complexa. Isso dificulta sua aprendizagem e uso.

A semântica das linguagens funcionais também é mais simples que a das imperativas. Por exemplo, na descrição em semântica denotacional de uma construção de laço imperativa dada na Seção 3.5.2, o laço é convertido de uma construção iterativa em uma construção recursiva. Essa conversão é desnecessária em uma linguagem funcional pura, em que não existe iteração. Além disso, supomos que não existem efeitos colaterais em expressões em todas as descrições de semântica denotacional das construções imperativas no Capítulo 3. Essa restrição não é realista para linguagens imperativas, porque todas as linguagens baseadas em C incluem efeitos colaterais em expressões; e ela não é necessária para as descrições denotacionais de linguagens funcionais puras.

Algumas pessoas que trabalham com programação funcional afirmam que seu uso resulta em um aumento de uma ordem de magnitude na produtividade, pois os programas funcionais têm apenas 10% do tamanho de seus correspondentes imperativos. Embora tais números tenham sido, de fato, registrados em certas áreas de problemas, para outras os programas funcionais têm cerca de 25% do tamanho das soluções imperativas para os mesmos problemas (Wadler, 1998). Esses fatores permitem que os proponentes das linguagens funcionais reivindiquem vantagens de produtividade sobre linguagens imperativas que variam de 4 a 10 vezes. No entanto, o tamanho dos programas simplesmente não é uma boa medida de produtividade. Certamente, nem todas as linhas de código-fonte têm uma complexidade igual, nem levam o mesmo tempo para serem produzidas. Na verdade, devido à necessidade de lidar com variáveis, os programas imperativos têm muitas linhas trivialmente simples para inicializar e realizar pequenas mudanças nas variáveis.

A eficiência de execução é outra base para comparação. Quando programas funcionais são interpretados, eles são mais lentos que seus correspondentes imperativos compilados. Entretanto, existem agora compiladores para a maioria das linguagens funcionais; então, as disparidades de velocidade de execução entre funcionais e imperativas compiladas não são mais tão significativas. Alguém pode ficar tentado a afirmar que, como os programas funcionais são significativamente menores que os imperativos equivalentes, devem executar muito mais rápido que os imperativos. Entretanto, isso geralmente não é o caso, devido a uma coleção de características das linguagens funcionais, como a avaliação preguiçosa, que têm impacto negativo na eficiência de execução. Considerando a eficiência relativa de programas funcionais e imperativos, é razoável estimar que um programa funcional típico seria executado em cerca de duas vezes mais tempo do que seu correspondente imperativo (Wadler, 1998). Essa pode parecer uma diferença significativa, capaz de levar à exclusão de linguagens funcionais para uma aplicação. Entretanto, essa diferença de fator de dois é importante apenas em situações nas quais a velocidade de execução é de extrema importância. Existem muitas situações nas quais um fator de dois na velocidade de execução não é considerado relevante. Por exemplo, considere que muitos programas escritos em linguagens imperativas, como um software para a Web escrito em JavaScript e PHP, são interpretados e, dessa forma, são muito mais lentos que suas versões equivalentes compiladas. Para essas aplicações, a velocidade de execução não é prioridade.

Outra origem da diferença na eficiência de execução entre programas funcionais e imperativos é o fato de as linguagens imperativas terem sido projetadas para executar eficientemente em computadores com arquitetura de von Neumann, enquanto o projeto das linguagens funcionais é baseado em funções matemáticas. Isso dá uma grande vantagem às linguagens imperativas.

Já as linguagens funcionais têm uma possível vantagem na legibilidade. Em muitos programas imperativos, os detalhes de lidar com variáveis obscurecem a lógica do programa. Considere uma função que computa a soma dos cubos dos primeiros `n` positivos inteiros. Em C, essa função provavelmente seria similar a:

```
int sum_cubes(int  n){
  int sum = 0;
  for(int index = 1; index <= n; index++)
    sum += index * index * index;
  return sum;
}
```

Em Haskell, a função poderia ser:

```
sumCubes n = sum (map (^3) [1..n])
```

Essa versão simplesmente especifica três passos:

1. Construir a lista de números (`[1..n]`).
2. Criar uma nova lista mapeando uma função que compute o cubo de um número em cada número da lista.
3. Somar a nova lista.

Devido aos detalhes de variáveis e de controle de iteração, essa versão é mais legível do que a em C[16*].

A execução concorrente nas linguagens imperativas é difícil de projetar e usar, como vimos no Capítulo 13. Em uma linguagem imperativa, o programador deve criar uma divisão estática do programa em suas partes concorrentes, então escritas como tarefas cuja execução frequentemente deve ser sincronizada. Esse processo pode ser complicado. Os programas em linguagens funcionais são naturalmente divididos em funções. Em uma linguagem funcional pura, essas funções são independentes, ou seja, não criam efeitos colaterais e suas operações não dependem de quaisquer variáveis não locais ou globais. Portanto, é muito mais fácil determinar qual delas pode ser executada concorrentemente. Nas chamadas, as expressões com parâmetros reais podem ser avaliadas concorrentemente. Simplesmente especificando-se que isso pode ser feito, uma função pode ser avaliada de forma implícita em uma linha de execução separada, como em Multilisp. E, é claro, o acesso a dados imutáveis compartilhados não exige sincronização.

Um fator simples que afeta bastante a complexidade da programação imperativa ou procedural é a necessária atenção do programador ao estado do programa em cada etapa de seu desenvolvimento. Em um programa grande, o estado é um grande número de valores (para o grande número de variáveis do programa). Na programação funcional pura não há estado; assim, não há necessidade de atenção para se lembrar disso.

Não é simples determinar com precisão por que as linguagens funcionais não alcançaram grande popularidade. A ineficiência das primeiras implementações era claramente um fator na época, e é provável que ao menos alguns programadores imperativos contemporâneos ainda acreditem que os programas escritos em linguagens funcionais são lentos. Além disso, a maioria dos programadores aprende programação por meio de linguagens imperativas – assim, muitas vezes os programas funcionais parecem estranhos e difíceis de entender. Para aqueles que se sentem à vontade com programação imperativa, a funcional pode ser difícil e desestimulante. Por outro lado, quem começa com uma linguagem funcional não enfrenta dificuldades com os programas funcionais.

RESUMO

Funções matemáticas são mapeamentos nomeados ou não nomeados que usam apenas expressões condicionais e recursão para controlar suas avaliações. Funções complexas podem ser definidas com funções de ordem superior ou com formas funcionais, nas quais funções são usadas como parâmetros, valores retornados ou ambos.

Linguagens de programação funcional são modeladas por meio de funções matemáticas. Em sua forma pura, não usam variáveis ou sentenças de atribuição para produzir resultados; em vez disso, utilizam aplicações de função, expressões condicionais e recursão para o controle da execução, e formas funcionais para construir funções complexas. Lisp

[16]Evidentemente, a versão C poderia ser escrita em um estilo mais funcional, mas a maioria dos programadores C provavelmente não a escreveria desse modo.

* N. de T.: Entretanto, nada impede que seja criada uma versão em C sem variáveis e sem instruções de iteração, por recursão. Algo como: `int sum_cube(int n){return (n == 1) ? 0 : n * n * n + sum_cube(n-1);}`.

começou como uma linguagem puramente funcional, mas logo adquiriu vários recursos de linguagens imperativas, adicionados para aumentar sua eficiência e facilitar seu uso.

A primeira versão de Lisp se desenvolveu a partir da necessidade de uma linguagem de processamento de listas para aplicações de IA. Lisp ainda é a linguagem mais usada para essa área de aplicação.

A primeira implementação de Lisp foi descoberta acidentalmente: a versão original de `EVAL` foi desenvolvida somente para demonstrar que uma função universal em Lisp poderia ser escrita.

Como os dados e programas Lisp têm a mesma forma, é possível fazer um programa construir outro programa. A disponibilidade de `EVAL` permite que programas construídos dinamicamente sejam executados de imediato.

Scheme é um dialeto relativamente simples de Lisp que usa unicamente escopo estático. Como Lisp, as principais primitivas de Scheme incluem funções para construir e separar listas, funções para expressões condicionais e predicados simples para números, símbolos e listas.

Common Lisp é uma linguagem baseada em Lisp projetada para incluir a maioria dos recursos dos dialetos de Lisp do início dos anos 1980. Ela permite tanto variáveis de escopo estático quanto de escopo dinâmico e inclui muitos recursos imperativos. Além disso, usa macros para definir algumas de suas funções; os usuários podem definir suas próprias macros. A linguagem inclui ainda macros de leitor, as quais também podem ser definidas pelo usuário. As macros de leitor definem macros de um símbolo.

ML é uma linguagem de programação funcional de escopo estático fortemente tipada que usa uma sintaxe mais relacionada à de uma linguagem imperativa que à de Lisp. Ela inclui um sistema de inferência de tipos, tratamento de exceções, uma variedade de estruturas de dados e tipos de dados abstratos.

ML não faz quaisquer coerções de tipo e não permite sobrecarga de função. Várias definições de funções podem ser feitas com casamento de padrões da forma de parâmetros reais. Currying é o processo de substituição de uma função que recebe vários parâmetros por outra que recebe apenas um e retorna uma função que recebe os parâmetros da outra. ML, como várias outras linguagens funcionais, suporta currying.

Haskell é similar a ML, exceto que todas as expressões nela são avaliadas com um método preguiçoso, o que permite aos programas lidarem com listas infinitas. Haskell também suporta compreensões de lista, que fornecem uma sintaxe conveniente e familiar para descrever conjuntos. Ao contrário de ML e Scheme, Haskell é uma linguagem funcional pura.

F# é uma linguagem .NET que suporta programação funcional e imperativa, incluindo a orientada a objetos. Seu núcleo de programação funcional é baseado em OCaml, uma descendente de ML e Haskell. F# é suportada por um IDE elaborado e bastante utilizado. Ela também opera em conjunto com outras linguagens .NET e tem acesso à biblioteca de classes .NET.

NOTAS BIBLIOGRÁFICAS

A primeira versão publicada de Lisp pode ser encontrada em McCarthy (1960). Uma versão amplamente utilizada de meados dos anos 1960 até meados dos 1970 é descrita em McCarthy et al. (1965) e Weissman (1967). Common Lisp é descrita em Steele (1990). A

linguagem Scheme é descrita em Dybvig (2009). ML é definida em Milner et al. (1997). Ullman (1998) é um livro-texto introdutório excelente sobre ML. A programação em Haskell é apresentada em Thompson (1999). F# é descrita em Syme et al. (2010).

Os programas em Scheme deste capítulo foram desenvolvidos com a linguagem R5RS, legada por DrRacket.

Uma discussão rigorosa sobre programação funcional pode ser encontrada em Henderson (1980). O processo de implementar linguagens funcionais por meio de redução de grafos é discutido em detalhes em Peyton Jones (1987).

QUESTÕES DE REVISÃO

1. Defina *forma funcional*, *lista simples*, *variável vinculada* e *transparência referencial*.
2. O que uma expressão lambda especifica?
3. Que tipos de dados eram partes de Lisp original?
4. Em que estrutura de dados comum as listas de Lisp são normalmente armazenadas?
5. Explique por que `QUOTE` é necessária para um parâmetro que é uma lista de dados.
6. O que é uma lista simples?
7. O que a abreviatura REPL significa?
8. Quais são os três parâmetros para `IF`?
9. Quais são as diferenças entre =, `EQ?`, `EQV?` e `EQUAL?`?
10. Quais são as diferenças entre o método de avaliação usado para a forma especial `DEFINE` de Scheme e aquele usado para suas funções primitivas?
11. Quais são as duas formas de `DEFINE`?
12. Descreva a sintaxe e a semântica de `COND`.
13. Por que `CAR` e `CDR` são chamadas dessa forma?
14. Se `CONS` é chamada com dois átomos, como `A` e `B`, o que é retornado?
15. Descreva a sintaxe e a semântica de `LET` em Scheme.
16. Quais são as diferenças entre `CONS`, `LIST` e `APPEND`?
17. Descreva a sintaxe e a semântica de `map` em Scheme.
18. O que é recursão em cauda? Por que é importante definir como recursivas em cauda as funções que usam recursão para especificar repetição?
19. Por que recursos imperativos foram adicionados à maioria dos dialetos de Lisp?
20. De que maneiras Common Lisp e Scheme são opostas?
21. Que regra de escopo é usada em Scheme? E em Common Lisp, ML, Haskell e F#?
22. O que acontece durante a fase leitor de um processador de linguagem Common Lisp?

23. Quais são as duas maneiras pelas quais ML é fundamentalmente diferente de Scheme?
24. O que é armazenado em um ambiente de avaliação ML?
25. Qual é a diferença entre uma sentença ML **`val`** e uma sentença de atribuição em C?
26. O que é inferência de tipos, conforme usada em ML?
27. Qual é o uso da palavra reservada **`fn`** em ML?
28. As funções ML que lidam com escalares numéricos podem ser genéricas?
29. O que é uma função que sofreu currying?
30. O que significa avaliação parcial?
31. Descreva as ações da função ML `filter`.
32. Qual operador ML usa para `CAR` de Scheme?
33. Qual operador ML usa para composição funcional?
34. Quais são as três características de Haskell que a tornam significativamente diferente de ML?
35. O que significa avaliação preguiçosa?
36. O que é uma linguagem de programação estrita?
37. Quais paradigmas de programação são suportados por F#?
38. Com quais outras linguagens de programação F# pode interoperar?
39. O que faz a construção **`let`** de F#?
40. Como o escopo de uma construção F# **`let`** é terminado?
41. Qual é a diferença básica entre uma sentença e uma lista em F#?
42. Qual é a diferença entre a construção `let` de ML e a de F#, em termos de extensão?
43. Qual é a sintaxe de uma expressão lambda em F#?
44. F# faz a coerção de valores numéricos em expressões? Defenda a escolha de projeto.
45. Que suporte Python fornece para programação funcional?
46. Qual função em Ruby é usada para criar uma função com currying?
47. O uso de programação funcional está aumentando ou diminuindo?
48. Cite uma das características das linguagens de programação funcional que tornam sua semântica mais simples que a de linguagens imperativas.
49. Qual é o problema de usar linhas de código para comparar a produtividade de linguagens funcionais e imperativas?
50. Por que a concorrência pode ser mais fácil com linguagens funcionais do que com linguagens imperativas?

PROBLEMAS

1. Leia o artigo de John Backus sobre FP (Backus, 1978) e compare os recursos de Scheme discutidos neste capítulo com os recursos correspondentes de FP.
2. Encontre definições das funções `EVAL` e `APPLY` de Scheme e explique suas ações.
3. Um dos ambientes de programação mais modernos e completos é o sistema INTERLISP para Lisp, como descrito em "The INTERLISP Programming Environment" por Teitelmen e Masinter (IEEE Computer, Vol. 14, No. 4, April 1981). Leia esse artigo atentamente e compare a dificuldade de escrever programas Lisp em seu sistema com a de escrever usando INTERLISP (supondo que você não use INTERLISP normalmente).
4. Consulte um livro sobre programação em Lisp e determine quais argumentos suportam a inclusão do recurso `PROG` em Lisp.
5. Encontre pelo menos um exemplo de linguagem de programação funcional tipada que seja para construir um sistema comercial em cada uma das seguintes áreas: processamento de banco de dados, modelagem financeira, análise estatística e bioinformática.
6. Uma linguagem funcional pode usar outra estrutura de dados além de listas. Por exemplo, poderia usar cadeias de símbolos de um caractere. Que primitivas deveria ter tal linguagem no lugar de `CAR`, `CDR` e `CONS` de Scheme?
7. Crie uma lista de recursos de F# que não estão em ML.
8. Se Scheme fosse uma linguagem funcional pura, poderia incluir `DISPLAY`? Justifique sua resposta.
9. O que a seguinte função Scheme faz?

```
(define (y s lis)
  (cond
    ((null? lis) '() )
    ((equal? s (car lis)) lis)
    (else (y s (cdr lis)))
))
```

10. O que a seguinte função Scheme faz?

```
(define (x lis)
  (cond
    ((null? lis) 0)
    ((not (list? (car lis)))
      (cond
        ((eq? (car lis) #f) (x (cdr lis)))
        (else (+ 1 (x (cdr lis))))))
    (else (+ (x (car lis)) (x (cdr lis))))))
```

EXERCÍCIOS DE PROGRAMAÇÃO

1. Escreva uma função Scheme que compute o volume de uma esfera, dado o seu raio.
2. Escreva uma função Scheme que compute as raízes reais de uma equação quadrática. Se as raízes forem complexas, a função deverá mostrar uma mensagem indicando isso. Essa função deve usar uma função `IF`. Os três parâmetros para a função são os três coeficientes da equação quadrática.
3. Repita o Exercício de programação 2 usando uma função `COND` em vez de uma função `IF`.
4. Escreva uma função Scheme que receba dois parâmetros numéricos, `A` e `B`, e retorne `A` elevado a `B`-ésima potência.
5. Escreva uma função Scheme que retorne o número de zeros em uma dada lista simples de números.
6. Escreva uma função Scheme que receba uma lista simples de números como parâmetro e retorne uma lista com o maior e o menor número da lista de entrada.
7. Escreva uma função em Scheme que receba como parâmetros uma lista e um átomo e retorne uma lista idêntica à sua lista de parâmetros, mas com todas as instâncias de alto nível do átomo dado excluídas.
8. Escreva uma função em Scheme que receba uma lista como parâmetro e retorne uma lista idêntica ao parâmetro, mas com o último elemento excluído.
9. Repita o Exercício de programação 7, mas, nesse caso, admita que o átomo pode ser um átomo ou uma lista.
10. Escreva uma função em Scheme que receba dois átomos e uma lista como parâmetros e retorne uma lista idêntica à lista de parâmetros, exceto por todas as ocorrências do primeiro átomo dado na lista serem substituídas pelo segundo átomo dado, independentemente de quão profundamente o primeiro átomo esteja aninhado.
11. Escreva uma função Scheme que retorne o inverso de seu parâmetro, que é uma lista simples.
12. Escreva uma função de predicado em Scheme que teste a igualdade estrutural de duas listas dadas. As duas listas são estruturalmente iguais se têm a mesma estrutura de listas, apesar de seus átomos poderem ser diferentes.
13. Escreva uma função Scheme que retorne a união de dois parâmetros que são listas simples representando conjuntos.
14. Escreva uma função Scheme com dois parâmetros, um átomo e uma lista, que retorne uma lista idêntica à lista de parâmetros, mas com todas as ocorrências, independentemente de quão profundas, do átomo dado excluídas. A lista retornada não pode conter nada no lugar dos átomos excluídos.
15. Escreva uma função em Scheme que receba uma lista como parâmetro e retorne uma lista idêntica à lista de parâmetros, mas com o segundo elemento de nível superior excluído. Se a lista não tem dois elementos, a função deve retornar `()`.

16 Escreva uma função Scheme que receba uma lista simples de números como parâmetro e retorne uma lista idêntica à lista de parâmetros, mas com os números em ordem ascendente.

17. Escreva uma função Scheme que receba uma lista simples como parâmetro e retorne uma lista de todas as permutações da lista dada.

18. Escreva uma função Scheme que receba uma lista simples como parâmetro e retorne uma lista com todas as permutações da lista dada.

19. Escreva o algoritmo de ordenação rápida (quicksort) em Scheme.

20. Reescreva a seguinte função Scheme como uma função recursiva em cauda:

```
(DEFINE (doit n)
  (IF (= n 0)
   0
   (+ n (doit (- n 1)))
))
```

21. Escreva qualquer um dos 19 primeiros exercícios de programação em F#.

22. Escreva qualquer um dos 19 primeiros exercícios de programação em ML.

Esta página foi deixada em branco intencionalmente.

16

Linguagens de programação lógica

16.1 Introdução

16.2 Uma breve introdução ao cálculo de predicados

16.3 Cálculo de predicados e prova de teoremas

16.4 Panorama da programação lógica

16.5 As origens de Prolog

16.6 Os elementos básicos de Prolog

16.7 Deficiências de Prolog

16.8 Aplicações de programação lógica

O objetivo deste capítulo é apresentar os conceitos de programação lógica e de linguagens de programação lógica, incluindo uma breve descrição de um subconjunto de Prolog. Começamos com uma introdução ao cálculo de predicados, a base para linguagens de programação lógica. Em seguida, discutimos como ele pode ser usado para sistemas automáticos de prova de teoremas. Então, apresentamos um panorama da programação lógica. A seguir, uma extensa seção apresenta os fundamentos da linguagem de programação Prolog, incluindo aritmética, processamento de listas e uma ferramenta de rastreamento que pode ser usada para ajudar a depurar programas e a ilustrar como o sistema Prolog funciona. As duas últimas seções descrevem alguns dos problemas de Prolog, como linguagem lógica, e algumas áreas de aplicação nas quais ele é usado.

16.1 INTRODUÇÃO

O Capítulo 15 discutiu o paradigma da programação funcional, significativamente diferente das metodologias de desenvolvimento de software usadas com as linguagens imperativas. Neste capítulo, descrevemos outra metodologia de programação. Aqui, a abordagem é baseada na expressão de programas em uma forma de lógica simbólica e usa um processo de inferência lógico para produzir resultados. Programas lógicos são declarativos, em vez de procedurais; ou seja, apenas as especificações dos resultados desejados são expressas, em vez dos procedimentos detalhados para produzi-los. Nas linguagens de programação lógica, os programas são coleções de fatos e regras. Tais programas são usados da seguinte forma: são feitas perguntas a eles, às quais tentam responder consultando os fatos e as regras. "Consultando" talvez seja não seja a palavra que mais se encaixa aqui, pois o processo é bem mais complexo do que ela dá a entender. Pode parecer que essa estratégia de solução de problemas trata somente de uma categoria muito limitada de problemas, mas ela é mais flexível do que se pode imaginar.

A programação que usa uma forma de lógica simbólica como linguagem de programação é geralmente chamada de **programação lógica**, e as linguagens baseadas em lógica simbólica são chamadas de **linguagens de programação lógica** ou **linguagens declarativas**. Optamos por descrever a linguagem de programação lógica Prolog, porque ela é a única amplamente usada.

A sintaxe das linguagens de programação lógica é notavelmente diferente daquela das linguagens funcionais e imperativas. A semântica dos programas lógicos também tem pouca semelhança com a dos programas em linguagens imperativas. Essas observações devem despertar alguma curiosidade no leitor acerca da natureza da programação lógica e das linguagens declarativas.

16.2 UMA BREVE INTRODUÇÃO AO CÁLCULO DE PREDICADOS

Antes de discutir programação lógica, devemos investigar brevemente sua base, a lógica formal. Esse não é o nosso primeiro contato com a lógica formal neste livro; ela foi extensivamente usada na semântica axiomática descrita no Capítulo 3.

Uma **proposição** é passível de ser vista como uma sentença lógica que pode ou não ser verdadeira. Ela consiste em objetos e relacionamentos entre objetos. A lógica formal foi desenvolvida para fornecer um método para descrever proposições, com o objetivo de permitir que essas proposições formalmente descritas pudessem ser verificadas em relação à sua validade.

A **lógica simbólica** pode ser usada para as três necessidades básicas da lógica formal: expressar proposições, expressar os relacionamentos entre proposições e descrever como novas proposições podem ser inferidas a partir de outras proposições que se presume verdadeiras.

Existe uma estreita relação entre lógica formal e matemática. Na verdade, muito da matemática pode ser pensado em termos de lógica. Os axiomas fundamentais da teoria dos números e dos conjuntos formam o conjunto inicial de proposições, os quais se presume verdadeiros. Teoremas são as proposições adicionais que podem ser inferidas a partir do conjunto inicial.

A forma particular de lógica simbólica usada para programação lógica é chamada de **cálculo de predicados de primeira ordem** (apesar de ser um pouco impreciso, nos referenciaremos a essa forma como *cálculo de predicados*). Nas subseções a seguir, apresentamos um breve panorama do cálculo de predicados. Nosso objetivo é estabelecer a base para uma discussão sobre programação lógica e linguagem de programação lógica Prolog.

16.2.1 Proposições

Os objetos em proposições de programação lógica são representados por termos simples, constantes ou variáveis. Uma constante é um símbolo que representa um objeto. Uma variável é um símbolo capaz de representar objetos diferentes em momentos diferentes, apesar de, em certo sentido, essas variáveis serem muito mais próximas da matemática do que aquelas de uma linguagem de programação imperativa.

As proposições mais simples, chamadas de **proposições atômicas**, consistem em termos compostos. Um **termo composto** é um elemento de uma relação matemática, escrito em uma forma que tem a aparência de uma notação de função matemática. Lembre-se de que, no Capítulo 15, definimos uma função matemática como um mapeamento que pode ser representado como uma expressão ou como uma tabela ou lista de tuplas. Termos compostos são elementos da definição tabular de uma função.

Um termo composto tem duas partes: um **functor**, símbolo da função que nomeia a relação, e uma lista ordenada de parâmetros que, juntos, representam um elemento da relação. Um termo composto com um único parâmetro é uma 1-tupla; um com dois parâmetros é uma 2-tupla e assim por diante. Por exemplo, poderíamos ter as duas proposições

```
man(jake)
like(bob, steak)
```

que dizem que {jake} é uma 1-tupla na relação chamada man e que {bob, steak} é uma 2-tuple na relação chamada like. Se adicionássemos a proposição

```
man(fred)
```

às duas proposições anteriores, a relação manteria dois elementos distintos, {jake} e {fred}. Todos os termos simples nessas proposições – man, jake, like, bob e steak – são constantes. Note que essas proposições não têm uma semântica intrínseca. Elas significam qualquer coisa que quisermos. Por exemplo, o segundo exemplo pode significar que bob gosta de bifes, ou que bifes gostam de bob, ou que bob é de alguma forma similar a um bife.

Proposições podem ser definidas de dois modos: um no qual a proposição é definida como verdadeira e outro no qual a verdade da proposição é algo que deve ser determinado. Em outras palavras, as proposições podem ser fatos ou consultas. As proposições exemplificadas acima podem ser ambos.

Proposições compostas têm duas ou mais proposições atômicas, conectadas por conectores lógicos, ou operadores, da mesma maneira que as expressões lógicas compostas são construídas em linguagens imperativas. Os nomes, os símbolos e o significado dos conectores lógicos do cálculo de predicados são:

Nome	*Símbolo*	*Exemplo*	*Significado*
negação	$\neg$	$\neg$ a	não *a*
conjunção	$\cap$	$a \cap b$	*a* e *b*
disjunção	$\cup$	$a \cup b$	*a* ou *b*
equivalência	$\equiv$	$a \equiv b$	*a* é equivalente a *b*
implicação	$\supset$	$a \supset b$	*a* implica *b*
	$\subset$	$a \subset b$	*b* implica *a*

Os seguintes são exemplos de proposições compostas:

$a \cap b \supset c$
$a \cap \neg b \supset d$

O operador $\neg$ tem a precedência mais alta. Os operadores $\cap$ e $\cup$ têm precedência mais alta que $\supset$ e $\subset$. Então, o segundo exemplo acima é equivalente a

$(a \cap (\neg b)) \supset d$

Variáveis podem aparecer em proposições, mas apenas quando introduzidas por símbolos especiais chamados de *quantificadores*. O cálculo de predicados inclui dois quantificadores, conforme descrito abaixo, onde *X* é uma variável e *P* é uma proposição:

Nome	*Exemplo*	*Significado*
universal	$\forall X.P$	Para todo *X*, *P* é verdadeiro.
Existencial	$\exists X.P$	Existe um valor de *X* tal que *P* é verdadeiro.

O ponto entre *X* e *P* simplesmente separa a variável da proposição. Por exemplo, considere o seguinte:

$\forall X.$ (mulher(X) $\supset$ humano(X))
$\exists X.$ (mãe(mary, X) $\cap$ homem(X))

A primeira dessas proposições significa que para qualquer valor de *X*, se *X* é uma mulher, *X* é um humano. O segundo significa que existe um valor de *X* tal que mary é a mãe de *X* e que *X* é um homem; em outras palavras, mary tem um filho. O escopo dos quantificadores universal e existencial é atômico às proposições às quais eles estão anexados. Esse escopo pode ser estendido com o uso de parênteses, como nas duas proposições compostas recém-descritas. Então, os quantificadores universal e existencial têm precedência mais alta que qualquer um dos operadores.

16.2.2 Forma clausal

Estamos discutindo o cálculo de predicados porque ele é a base para linguagens de programação lógica. Assim como outras linguagens, as lógicas são melhores em sua forma simples; isso significa que a redundância deve ser minimizada.

Um problema do cálculo de predicados, como descrevemos até agora, é que existem muitas maneiras de definir proposições com o mesmo significado; ou seja, existe uma grande quantidade de redundância. Esse não é um problema para especialistas em lógica matemática, mas se o cálculo de predicados será usado em um sistema automatizado (computadorizado), é um problema sério. Para simplificar as coisas, é desejável uma forma padrão para proposições. A forma clausal, uma forma relativamente simples de proposições, é uma delas. Todas as proposições podem ser expressas em forma clausal. Uma proposição em forma clausal tem a seguinte sintaxe geral:

$$B_1 \cup B_2 \cup \ldots \cup B_n \subset A_1 \cap A_2 \cap \ldots \cap A_m$$

na qual os *A*s e *B*s são termos. O significado dessa proposição de forma clausal é: se todos os *A*s são verdadeiros, então pelo menos um *B* é verdadeiro. As principais características das proposições na forma clausal são: quantificadores existenciais não são obrigatórios; quantificadores universais são implícitos no uso de variáveis nas proposições atômicas; e nenhum operador, além da conjunção e da disjunção, é obrigatório. Além disso, a conjunção e a disjunção precisam aparecer apenas na ordem mostrada na forma clausal geral: as disjunções no lado esquerdo e as conjunções no lado direito. Todas as proposições de cálculo de predicados podem ser algoritmicamente convertidas na forma clausal. Nilsson (1971) prova que isso pode ser feito, bem como mostra um algoritmo de conversão simples para fazê-lo.

O lado direito de uma proposição na forma clausal é chamado de **antecedente**. O lado esquerdo é chamado de **consequente**, porque é a consequência da verdade do antecedente. Como exemplos de proposições na forma clausal, considere:

likes(bob, trout) $\subset$ likes(bob, fish) $\cap$ fish(trout)

father(louis, al) ∪ father(louis, violet) ⊂
    father(al, bob) ∩ mother(violet, bob) ∩ grandfather(louis, bob)

A versão em português da primeira dessas sentenças diz que, se bob gosta de peixe e se uma truta é um peixe, então bob gosta de trutas. A segunda diz que se al é o pai de bob, violet é a mãe de bob e louis é o avô de bob, então ou louis é o pai de al ou louis é o pai de violet.

16.3 CÁLCULO DE PREDICADOS E PROVA DE TEOREMAS

O cálculo de predicados fornece um método de expressar coleções de proposições. Um uso de coleções de proposições é determinar se quaisquer fatos interessantes ou úteis podem ser inferidos a partir delas. Isso é exatamente análogo ao trabalho dos matemáticos, os quais buscam descobrir novos teoremas que podem ser inferidos a partir de axiomas e teoremas conhecidos.

Nos primórdios da ciência da computação (os anos 1950 e o início dos anos 1960), havia interesse em automatizar o processo de prova de teoremas. Um dos avanços mais significativos na prova automática de teoremas foi a descoberta do princípio de resolução, por Alan Robinson (1965), na Universidade de Syracuse.

A **resolução** é uma regra de inferência que permite às proposições inferidas serem computadas a partir de proposições dadas, fornecendo um método com aplicação potencial para a prova automática de teoremas. A resolução foi desenvolvida para ser aplicada a proposições na forma clausal. O conceito de resolução é o seguinte: suponha que existam duas proposições, com as formas

$P_1 \subset P_2$
$Q_1 \subset Q_2$

Seus significados são que P_2 implica P_1 e Q_2 implica Q_1. Além disso, suponha que P_1 seja idêntico a Q_2, de modo que poderíamos chamar P_1 e Q_2 de T. Então, poderíamos reescrever as duas proposições como

$T \subset P_2$
$Q_1 \subset T$

Agora, como P_2 implica T e T implica Q_1, é logicamente óbvio que P_2 implica Q_1, o qual poderíamos escrever como

$Q_1 \subset P_2$

O processo de inferir essa proposição a partir das duas proposições originais é chamado de resolução.

Como outro exemplo, considere as duas proposições:

older(joanne, jake) ⊂ mother(joanne, jake)
wiser(joanne, jake) ⊂ older(joanne, jake)

A partir dessas proposições, as seguintes podem ser construídas por meio de resolução:

wiser(joanne, jake) ⊂ mother(joanne, jake)

A mecânica dessa construção de resolução é simples: os termos dos lados esquerdos das duas proposições clausais são unidos por um OU para formar o lado esquerdo da nova proposição. Então, os lados direitos das duas proposições clausais são unidos por um E para formar o lado direito da nova proposição. A seguir, qualquer termo que aparece nos dois lados da nova proposição é removido de ambos. O processo é exatamente o mesmo quando as proposições têm vários termos em um ou em ambos os lados. O lado esquerdo da nova proposição inferida inicialmente contém todos os termos dos lados esquerdos das duas proposições dadas. O novo lado direito é construído de maneira similar. Então, o termo que aparece em ambos os lados da nova proposição é removido. Por exemplo, se tivermos

father(bob, jake) ∪ mother(bob, jake) ⊂ parent(bob, jake)
grandfather(bob, fred) ⊂ father(bob, jake) ∩ father(jake, fred)

a resolução diz que

mother(bob, jake) ∪ grandfather(bob, fred) ⊂
parent(bob, jake) ∩ father(jake, fred)

que tem todas as proposições atômicas de ambas as proposições originais, exceto uma. A proposição atômica que permitiu a operação father(bob, jake) no lado esquerdo da primeira e no lado direito da segunda é deixada de fora. Em português, diríamos

se: bob ser um dos pais de jake implica que bob é ou o pai ou a mãe de jake
e: bob ser o pai de jake e jake ser o pai de fred implica que bob é o avô de fred
então: *se* bob é o pai de jake e jake é o pai de fred, *então:* bob é pai de jake ou bob é avô de fred

A resolução é mais complexa do que esses exemplos simples ilustram. Em particular, a presença de variáveis nas proposições exige que a resolução encontre valores para essas variáveis que permitam ao processo de casamento ser bem-sucedido. Esse processo de determinar valores úteis para variáveis é chamado de **unificação**. A atribuição temporária de valores a variáveis para permitir a unificação é chamada de **instanciação**.

É comum o processo de resolução instanciar uma variável com um valor, falhar em completar o casamento necessário e, então, ser necessária uma volta – chamada de rastreamento para trás (*backtracking*) – para instanciar a variável com um valor diferente. Discutiremos a unificação e o rastreamento para trás mais extensivamente no contexto de Prolog.

Uma propriedade criticamente importante da resolução é a capacidade de detectar qualquer inconsistência em um conjunto de proposições. Isso é baseado na propriedade formal da resolução chamada **refutação completa**, o que significa que, dado um conjunto de proposições inconsistentes, a resolução pode provar que elas são inconsistentes. Isso permite que a resolução seja usada para provar teoremas, o que pode ser feito como

segue: é possível visualizar uma prova de teorema, em termos de cálculo de predicados, como um conjunto de proposições pertinentes, com a negação do próprio teorema declarada como a nova proposição. O teorema é negado a fim de que a resolução possa ser usada para provar o teorema encontrando uma inconsistência. Isso se chama provar pela contradição, uma estratégia frequentemente usada na prova de teoremas matemáticos. As proposições originais são comumente chamadas de **hipóteses** e a negação do teorema é chamada de **objetivo**.

Teoricamente, esse processo é válido e útil. O tempo necessário para a resolução, entretanto, pode ser um problema. Embora a resolução seja um processo finito quando o conjunto de proposições é finito, o tempo necessário para encontrar uma inconsistência em uma grande base de dados de proposições pode ser imenso.

A prova de teoremas é a base para a programação lógica. Muito do que é computado pode ser descrito na forma de uma lista de fatos e relacionamentos dados como hipóteses e de um objetivo a ser inferido a partir das hipóteses, por meio de resolução.

Na prática, frequentemente não existe uma resolução para uma hipótese e um objetivo que sejam proposições gerais, mesmo que estejam em forma clausal. Embora possa parecer possível provar um teorema usando proposições de forma clausal, isso pode não acontecer em um período de tempo razoável. Um modo de simplificar o processo de resolução é restringir a forma das proposições. Uma restrição útil é exigir que as proposições sejam cláusulas de Horn. **Cláusulas de Horn** só podem estar em uma de duas formas: ou elas têm uma única proposição atômica no lado esquerdo ou um lado esquerdo vazio.[1] O lado esquerdo de uma proposição de forma clausal algumas vezes é chamado de cabeça, e as cláusulas de Horn com lados esquerdos são chamadas de cláusulas de Horn com cabeça. Estas são usadas para definir relacionamentos, como

likes(bob, trout) ⊂ likes(bob, fish) ∩ fish(trout)

Cláusulas de Horn com lados esquerdos vazios, usadas para definir fatos, são chamadas de *cláusulas de Horn sem cabeça*. Por exemplo,

father(bob, jake)

A maioria das proposições pode ser definida como cláusulas de Horn (nem todas, no entanto). A restrição em cláusulas de Horn torna a resolução um processo viável para a prova de teoremas.

16.4 PANORAMA DA PROGRAMAÇÃO LÓGICA

Linguagens usadas para programação lógica são chamadas de *linguagens declarativas*, porque os programas escritos nelas consistem em declarações, em vez de atribuições e sentenças de fluxo de controle. Essas declarações são na verdade sentenças, ou proposições, em lógica simbólica.

Uma das características essenciais das linguagens de programação lógica é sua semântica, chamada de **semântica declarativa**. O conceito básico dessa semântica é que existe uma maneira simples de determinar o significado de cada sentença, e ele não

[1] As cláusulas de Horn são chamadas assim devido a Alfred Horn (1951), que estudou cláusulas nessa forma.

depende de como ela poderia ser usada para resolver um problema. A semântica declarativa é consideravelmente mais simples do que a das linguagens imperativas. Por exemplo, o significado de uma proposição em uma linguagem de programação lógica pode ser concisamente determinado a partir da própria sentença. Em uma linguagem imperativa, a semântica de uma simples sentença de atribuição requer o exame de declarações locais, o conhecimento das regras de escopo da linguagem e possivelmente até mesmo o exame de programas em outros arquivos, apenas para determinar os tipos das variáveis na sentença de atribuição. Então, supondo que a expressão de atribuição contenha variáveis, a execução do programa antes da sentença de atribuição deve ser rastreada para determinar os valores dessas variáveis. Assim, a ação resultante da sentença depende de seu contexto de tempo de execução. Comparando essa semântica com a de uma proposição em uma linguagem lógica, sem a necessidade de considerar o contexto textual ou de sequências de execução, é claro que a semântica declarativa é muito mais simples que a das linguagens imperativas. Logo, geralmente se afirma que a semântica declarativa é uma das vantagens das linguagens declarativas em relação às imperativas (Hogger, 1984, pp. 240-241).

A programação tanto em linguagens imperativas quanto em funcionais é principalmente procedural. Isso significa que o programador sabe *o que* deve ser realizado por um programa e instrui o computador sobre *como*, exatamente, a computação deve ser feita. Em outras palavras, o computador é tratado como um simples dispositivo que obedece ordens. Deve ser informado sobre cada um dos detalhes de tudo o que será computado. Alguns acreditam que essa é a essência da dificuldade de programar usando linguagens imperativas e funcionais.

A programação em uma linguagem de programação lógica é não procedural. Os programas em tais linguagens não descrevem exatamente *como* um resultado será computado, mas a forma do resultado. A diferença é o fato de assumirmos que o sistema de computação pode, de alguma forma, determinar *como* o resultado será computado. O que é necessário para fornecer essa capacidade a linguagens de programação lógica é um meio conciso de dar ao computador tanto as informações relevantes quanto um método de inferência para computar os resultados desejados. O cálculo de predicados fornece a forma básica de comunicação com o computador e a resolução fornece a técnica de inferência.

A ordenação é comumente usada para ilustrar a diferença entre sistemas procedurais e não procedurais. Em uma linguagem como Java, a ordenação é feita por meio da explicação, em um programa, de todos os detalhes de algum algoritmo de ordenação para um computador que possui um compilador Java. O computador, após traduzir o programa Java em código de máquina ou algum código intermediário interpretável, segue as instruções e produz a lista ordenada.

Em uma linguagem não procedural, só é necessário descrever as características da lista ordenada: é alguma permutação da lista, tal que, para cada par de elementos adjacentes, um relacionamento se mantém entre os dois elementos. Para descrever isso formalmente, suponha que a lista a ser ordenada seja um vetor nomeado com uma faixa de índices de 1. . . *n*. O conceito de ordenar os elementos de uma lista, chamada old_list, e colocá-los em um vetor separado, chamado new_list, pode ser expresso como:

sort(old_list, new_list) ⊂ permute(old_list, new_list) ∩ sorted(new_list)
sorted(list) ⊂ “j such that 11 £ j < n, list(j) £ list(j + 1)

onde permute é um predicado que retorna verdadeiro se seu segundo parâmetro, um vetor, é uma permutação do primeiro parâmetro, também um vetor.

A partir dessa descrição, o sistema de linguagem não procedural pode produzir a lista ordenada. Isso faz a programação não procedural parecer uma mera produção de especificações de requisitos de software concisas, o que é uma avaliação justa. Infelizmente, porém, não é tão simples. Os programas lógicos que usam apenas resolução encaram sérios problemas de eficiência de execução. Em nosso exemplo de ordenação, se a lista é longa, o número de permutações é enorme, e elas devem ser geradas e testadas, uma por uma, até que seja encontrada uma que esteja em ordem — um processo muito demorado. Evidentemente, deve-se considerar a possibilidade de que a melhor forma para uma linguagem lógica ainda não tenha sido determinada, e a de que bons métodos para criar programas em linguagens de programação lógica para grandes problemas ainda não tenham sido desenvolvidos.

16.5 AS ORIGENS DE PROLOG

Conforme mencionado no Capítulo 2, Alain Colmerauer e Phillippe Roussel, na Universidade de Aix-Marseille, com alguma assistência de Robert Kowalski, na Universidade de Edimburgo, desenvolveram o projeto fundamental de Prolog. Colmerauer e Roussel estavam interessados em processamento de linguagem natural; Kowalski, na prova automatizada de teoremas. A colaboração entre a Universidade de Aix-Marseille e a Universidade de Edimburgo continuou até meados dos anos 1970. Desde então, a pesquisa sobre o desenvolvimento e o uso da linguagem progrediu independentemente nesses dois locais, resultando em dois dialetos sintaticamente diferentes de Prolog, entre outras coisas.

O desenvolvimento de Prolog e outros esforços de pesquisa em programação lógica receberam atenção limitada fora de Edimburgo e Marselha, até o anúncio, em 1981, de que o governo japonês estava lançando um grande projeto de pesquisa chamado de Quinta Geração de Sistemas de Computação (FGCS – Fifth Generation Computing Systems) (Fuchi, 1981; Moto-oka, 1981). Um dos objetivos primários do projeto era desenvolver máquinas inteligentes, e Prolog foi escolhida como base para esse esforço. O anúncio da FGCS fez surtir um acentuado e súbito interesse em inteligência artificial e em programação lógica nos pesquisadores e governos dos EUA e de diversos países europeus.

Após uma década de esforço, o projeto FGCS foi silenciosamente abandonado. Apesar do imenso potencial que se supunha na programação lógica e em Prolog, poucas coisas significativas foram descobertas. Isso levou ao declínio no interesse e no uso de Prolog, apesar de ele ainda ter seus defensores.

16.6 OS ELEMENTOS BÁSICOS DE PROLOG

Existem agora inúmeros dialetos de Prolog. Eles podem ser agrupados em diversas categorias: os que cresceram a partir do grupo de Marselha, os que vieram a partir do grupo de Edimburgo e alguns dialetos que foram desenvolvidos para microcomputa-

dores, como micro-Prolog, descrito por Clark e McCabe (1984). As formas sintáticas desses são de certa forma diferentes. Em vez de tentar descrever a sintaxe de diversos dialetos de Prolog ou algum híbrido entre eles, escolhemos um dialeto específico, amplamente disponível: aquele desenvolvido em Edimburgo. Essa forma da linguagem às vezes é chamada de **sintaxe de Edimburgo**. Sua primeira implementação foi em um DEC System-10 (Warren et al., 1979). Implementações de Prolog estão disponíveis para praticamente todas as plataformas de computadores populares, por exemplo, a partir da Organização de Software Livre (http://www.gnu.org).

16.6.1 Termos

Como os programas em outras linguagens, os escritos em Prolog consistem em coleções de sentenças. Existem apenas alguns tipos de sentenças em Prolog, mas elas podem ser complexas. Todas as sentenças e dados de Prolog são construídos a partir de termos.

Um **termo** Prolog é uma constante, uma variável ou uma estrutura. Uma constante é um átomo ou um inteiro. Átomos são os valores simbólicos de Prolog e são similares aos seus correspondentes em LISP. Em particular, um átomo é uma cadeia de letras, dígitos e sublinhados que iniciam com uma letra minúscula ou uma cadeia de quaisquer caracteres ASCII delimitados por apóstrofes.

Uma variável é qualquer cadeia de letras, dígitos e sublinhados que iniciam com uma letra maiúscula ou um sublinhado (_). Variáveis não são vinculadas a tipos por declarações. A vinculação de um valor a uma variável, e dessa forma a um tipo, é chamada de **instanciação**. A instanciação ocorre apenas no processo de resolução. Uma variável que ainda não recebeu um valor é chamada de **não instanciada**. As instanciações duram apenas o tempo necessário para satisfazer um objetivo completo, o que envolve a prova ou a falsidade de uma proposição. Variáveis Prolog são apenas parentes distantes, tanto em termos de semântica quanto de uso, das variáveis das linguagens imperativas.

O último tipo de termo é chamado de **estrutura**. Estruturas representam as proposições atômicas do cálculo de predicados e sua forma geral é a mesma:

```
functor(lista de parâmetros)
```

O functor é qualquer átomo e é usado para identificar a estrutura. A lista de parâmetros pode ser qualquer lista de átomos, de variáveis ou de outras estruturas. Conforme discutido extensivamente na subseção a seguir, as estruturas são a maneira de especificar fatos em Prolog. Elas podem ser consideradas objetos, permitindo que os fatos sejam definidos em termos de diversos átomos relacionados. Nesse sentido, são relações, já que definem relacionamentos entre termos. Uma estrutura também é um predicado quando seu contexto a especifica como uma consulta (pergunta).

16.6.2 Sentenças de fatos

Nossa discussão sobre sentenças Prolog começa com aquelas usadas para construir as hipóteses ou a base de dados de informações predefinidas – as sentenças a partir das quais novas informações podem ser inferidas.

Prolog tem duas formas básicas de sentenças; aquelas que correspondem às cláusulas de Horn com cabeça e às sem cabeça do cálculo de predicados. A forma mais simples de cláusulas de Horn sem cabeça em Prolog é uma estrutura única, interpretada como uma asserção incondicional, ou fato. Logicamente, fatos são simplesmente proposições que se presume serem verdadeiras.

Os exemplos a seguir ilustram os tipos de fatos que podem existir em um programa Prolog. Note que cada sentença Prolog é terminada por um ponto.

```
female(shelley).
male(bill).
female(mary).
male(jake).
father(bill, jake).
father(bill, shelley).
mother(mary, jake).
mother(mary, shelley).
```

Essas estruturas simples afirmam certos fatos acerca de `jake`, `shelley`, `bill` e `mary`. Por exemplo, a primeira diz que `shelley` é uma mulher (`female`). As últimas quatro conectam seus dois parâmetros com um relacionamento nomeado no átomo do functor; por exemplo, o significado da quinta proposição pode ser interpretado como `bill` é o pai (`father`) de `jake`. Note que essas proposições Prolog, como aquelas do cálculo de predicados, não têm uma semântica intrínseca. Elas significam qualquer coisa que os programadores queiram que signifiquem. Por exemplo, a proposição

```
father(bill, jake).
```

poderia significar que `bill` e `jake` têm o mesmo pai (`father`) ou que `jake` é o pai (`father`) de `bill`. Contudo, o significado mais comum e simples poderia ser que `bill` é o pai de `jake`.

16.6.3 Sentenças de regras

A outra forma básica de sentenças Prolog para construir a base de dados corresponde a uma cláusula de Horn com cabeça. Essa forma pode ser relacionada a um conhecido teorema na matemática a partir do qual uma conclusão pode ser tirada se o conjunto das condições dadas for satisfeito. O lado direito é o antecedente, ou parte *se*, e o lado esquerdo é o consequente, ou parte *então*. Se o antecedente de uma sentença Prolog é verdadeiro, então o consequente da sentença também deve ser. Como são cláusulas de Horn, o consequente de uma sentença Prolog é um termo simples, enquanto o antecedente pode ser um termo simples ou uma conjunção.

Conjunções contêm vários termos separados por operações E lógicas. Em Prolog, a operação E é implícita. As estruturas que especificam proposições atômicas em uma conjunção são separadas por vírgulas, então alguém pode considerar as vírgulas como operadores E. Como exemplo de uma conjunção, considere:

```
female(shelley), child(shelley).
```

A forma geral da sentença de cláusula de Horn com cabeça em Prolog é

```
consequente :- expressão_antecedente.
```

Ela é lida da seguinte forma: "o consequente pode ser concluído se a expressão antecedente for verdadeira, ou puder se tornar verdadeira por alguma instanciação de suas variáveis". Por exemplo,

```
ancestor(mary, shelley) :- mother(mary, shelley).
```

afirma que se `mary` é a mãe (`mother`) de `shelley`, então ela é uma ancestral (`ancestor`) de `shelley`. As cláusulas de Horn com cabeça são chamadas de **regras**, porque definem regras de implicação entre proposições.

Assim como as proposições de forma clausal no cálculo de predicados, sentenças Prolog podem usar variáveis para generalizar seu significado. Lembre-se de que as variáveis em forma clausal fornecem um tipo de quantificador universal implícito. O seguinte exemplo demonstra o uso de variáveis em sentenças Prolog:

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
grandparent(X, Z) :- parent(X, Y) , parent(Y, Z).
```

Essas sentenças dão regras de implicação entre algumas variáveis ou objetos universais. Nesse caso, os objetos universais são `X`, `Y` e `Z`. A primeira regra diz que se existem instanciações de `X` e `Y` tais que `mother(X, Y)` é verdadeira, então, para essas mesmas instanciações de `X` e `Y`, `parent(X, Y)` é verdadeira.

O operator =, que é infixo, é bem-sucedido se seus dois operandos de termo são iguais. Por exemplo, `X = Y`. O operador `not`, que é unário, inverte seu operando, ou seja, é bem-sucedido se seu operando falha. Por exemplo, `not(X = Y)` é bem-sucedido se `X` não é igual a `Y`.

16.6.4 Sentenças de objetivos

Até agora, descrevemos as sentenças Prolog para proposições lógicas, usadas para descrever tanto fatos conhecidos quanto regras que descrevem relacionamentos lógicos entre fatos. Essas sentenças servem de base para o modelo de prova de teoremas. O teorema tem a forma de uma proposição cuja veracidade ou falsidade queremos que o sistema prove. Em Prolog, essas proposições são chamadas de **objetivos** ou **consultas**. A forma sintática das sentenças de objetivo em Prolog é idêntica à das cláusulas de Horn sem cabeça. Por exemplo, poderíamos ter

```
man(fred).
```

para o qual o sistema responderia `yes` ou `no`. A resposta `yes` significa que o sistema provou o objetivo como verdadeiro, de acordo com a base de dados de fatos e relacionamentos dados. A resposta `no` significa que se determinou o objetivo como falso ou que o sistema não foi capaz de prová-lo.

Proposições conjuntivas e proposições com variáveis também são objetivos válidos. Quando variáveis estão presentes, o sistema não apenas avalia a validade do objetivo, mas também identifica as instanciações das variáveis que o tornam verdadeiro. Por exemplo,

```
father(X, mike).
```

pode ser perguntado. O sistema irá tentar, por unificação, encontrar uma instanciação de `X` que resulte em um valor verdadeiro para o objetivo.

Como sentenças de objetivo e algumas sentenças que não são de objetivo têm a mesma forma (cláusulas de Horn sem cabeça), uma implementação Prolog deve ter algum modo de distinguir entre as duas. Implementações de Prolog interativas fazem isso de dois modos, indicados por diferentes interpretadores de comandos interativos: um para informar fatos e sentenças de regras e um para informar objetivos. O usuário pode modificar o modo a qualquer momento.

16.6.5 O processo de inferência de Prolog

Esta seção examina a resolução de Prolog. O uso eficiente de Prolog requer que o programador saiba precisamente o que esse sistema faz com o seu programa.

As consultas são chamadas de **objetivos**. Quando um objetivo é uma proposição composta, cada um dos fatos (estruturas) é chamado de **subobjetivo**. Para provar que um objetivo é verdadeiro, o processo de inferência deve encontrar uma cadeia de regras de inferência e/ou fatos na base de dados que conecte o objetivo a um ou mais fatos nessa base. Por exemplo, se Q é o objetivo, então ele deve ser encontrado como um fato na base de dados, ou o processo de inferência deve encontrar um fato P_1 e uma sequência de proposições P_2, P_3, . . ., P_n tal que

P_2 :- P_1
P_3 :- P_2
. . .
Q :- P_n

É claro que o processo pode ser, e frequentemente é, complicado por regras com lados direitos compostos e regras com variáveis. O processo de encontrar os Ps, quando eles existem, é basicamente uma comparação, ou casamento, de termos uns com os outros.

Como o processo de fornecer um subobjetivo é feito por meio de um casamento de proposições, ele é algumas vezes chamado de **casamento**. Em alguns casos, provar um subobjetivo é chamado de **satisfazer** o subobjetivo.

Considere esta consulta:

```
man(bob).
```

Essa sentença de objetivo é do tipo mais simples. É relativamente fácil a resolução determinar se é verdadeira ou falsa: o padrão desse objetivo é comparado com os fatos e regras na base de dados. Se a base incluir o fato

```
man(bob).
```

a prova é trivial. Se, entretanto, a base de dados contiver os seguintes fato e regra de inferência,

```
father(bob).
man(X) :- father(X).
```

Prolog precisará encontrar essas duas sentenças e usá-las para inferir a verdade do objetivo. Isso necessitaria de unificação para instanciar `X` temporariamente para `bob`.

Agora, considere o objetivo

```
man(X).
```

Nesse caso, Prolog deveria casar o objetivo com as proposições na base de dados. A primeira proposição encontrada com o formato do objetivo, com qualquer objeto como seu parâmetro, fará `X` ser instanciado com o valor desse objeto. `X` então é mostrado como resultado. Se não existe uma proposição com a forma do objetivo, o sistema indica, dizendo que não (`no`), o objetivo não pode ser satisfeito.

Existem duas estratégias opostas para tentar casar um objetivo com um fato na base de dados. O sistema pode começar com os fatos e regras da base de dados e tentar encontrar uma sequência de casamentos que levem ao objetivo. Essa estratégia é chamada de **resolução ascendente** (*bottom-up*) ou **encadeamento para frente** (*forward chaining*). A alternativa é começar com o objetivo e tentar encontrar uma sequência de proposições que casem com ele e levem a algum conjunto de fatos originais na base de dados. Essa estratégia é chamada de **resolução descendente** (*top-down*) ou **encadeamento para trás** (*backward chaining*). Em geral, o encadeamento para trás funciona bem quando existe um conjunto razoavelmente pequeno de respostas candidatas. A estratégia do encadeamento para frente é melhor quando o número de possíveis respostas corretas é grande; nessa situação, o encadeamento para trás exigiria um número de casamentos muito grande para chegar a uma resposta. As implementações de Prolog usam encadeamento para trás para a resolução, presumivelmente porque os projetistas acreditavam que ele seria mais adequado para uma classe maior de problemas do que o encadeamento para frente.

O seguinte exemplo ilustra a diferença entre encadeamento para frente e para trás. Considere a consulta:

```
man(bob).
```

Suponha que a base de dados contenha

```
father(bob).
man(X) :- father(X).
```

O encadeamento para frente procuraria e encontraria a primeira proposição. O objetivo é, então, inferido por meio do casamento da primeira proposição com o lado direito da segunda regra `(father(X))` pela instanciação de `X` como `bob` e por meio do casamento do lado esquerdo da segunda proposição para o objetivo. O encadeamento para trás primeiro casaria o objetivo com o lado esquerdo da segunda proposição `(man(X))`

por meio da instanciação de `X` para `bob`. Como último passo, ele casaria o lado direito da segunda proposição (agora `father(bob)`) com a primeira proposição.

A próxima questão de projeto surge sempre que o objetivo tem mais de uma estrutura, como nosso exemplo. A questão então é se a busca da solução é feita primeiro em profundidade ou em largura. Uma busca **primeiro em profundidade** (*depth first*) encontra uma sequência completa de proposições – uma prova – para o primeiro subobjetivo antes de trabalhar com os outros. Uma busca **primeiro em largura** (*breadth first*) funciona em todos os subobjetivos de um objetivo em paralelo. Os projetistas de Prolog escolheram a estratégia de busca primeiro em profundidade porque ela pode ser feita com menos recursos computacionais. A estratégia primeiro em largura é uma busca paralela que pode exigir uma grande quantidade de memória.

O último recurso do mecanismo de resolução de Prolog que deve ser discutido é o rastreamento para trás (*backtraking*). Quando um objetivo com vários subobjetivos está sendo processado e o sistema não consegue mostrar a verdade de um de seus subobjetivos, ele abandona o subobjetivo que não pode provar. Então, o sistema reconsidera o subobjetivo anterior, se existir um, e tenta encontrar uma solução alternativa para ele. Essa volta no objetivo para a reconsideração de um subobjetivo previamente provado é chamada de **rastreamento para trás** (*backtracking*). Uma nova solução é encontrada iniciando-se a busca onde a busca anterior para tal subobjetivo parou. Múltiplas soluções para um subobjetivo resultam de diferentes instanciações de suas variáveis. O rastreamento para trás pode exigir uma boa dose de tempo e espaço, porque talvez precise encontrar todas as provas possíveis para cada um dos subobjetivos. Essas provas de cada subobjetivo podem não estar organizadas com o intuito de minimizar o tempo necessário para encontrar aquele que resultará na prova final completa, o que piora o problema.

Para solidificar seu entendimento sobre rastreamento para trás, considere o seguinte exemplo. Suponha que exista um conjunto de fatos e regras na base de dados e que foi apresentado ao Prolog o seguinte objetivo composto:

```
male(X), parent(X, shelley).
```

Esse objetivo pergunta se existe uma instanciação de `X` tal que `X` é um homem (male) e um dos pais (`parent`) de `shelley`. Como primeiro passo, Prolog encontra o primeiro fato na base de dados com homem (`male`) como seu functor. Então, ele instancia `X` para o parâmetro do fato encontrado, `mike`, por exemplo. Em seguida, ele tenta provar que `parent(mike, shelley)` é verdadeiro. Se for falso, ele volta para o primeiro subobjetivo, `male(X)`, e tenta satisfazê-lo novamente com alguma instanciação alternativa de `X`. O processo de resolução talvez precise encontrar todos os homens na base de dados antes de encontrar aquele que é um dos pais de `shelley`. Ele certamente precisa encontrar todos os homens para provar que o objetivo não pode ser satisfeito. Note que o objetivo do nosso exemplo poderia ser processado mais eficientemente se a ordem dos dois subobjetivos fosse a inversa. Então, apenas após a resolução encontrar um dos pais de `shelley` ela tentará casar essa pessoa com o subobjetivo homem (`male`). Seria mais eficiente se `shelley` tivesse menos pais do que homens existentes na base de dados, o que parece ser uma ideia razoável. A Seção 16.7.1 discute um método para limitar o rastreamento para trás feito por um sistema Prolog.

As buscas em bases de dados em Prolog sempre procedem na direção da primeira para a última.

As duas subseções a seguir descrevem exemplos em Prolog que ilustram melhor o processo de resolução.

16.6.6 Aritmética simples

Prolog suporta variáveis inteiras e aritmética de inteiros. Originalmente, os operadores aritméticos eram functores, então a soma de `7` com a variável `X` era formada com

```
+(7, X)
```

Prolog agora permite uma sintaxe mais abreviada para aritmética com o operador **`is`**. Esse operador recebe uma expressão aritmética como seu operando direito e uma variável como seu operando esquerdo. Todas as variáveis na expressão já devem estar instanciadas, mas a variável do lado esquerdo ainda não pode ter sido instanciada. Por exemplo, em

```
A is B / 17 + C.
```

se `B` e `C` são instanciadas, mas `A` não, essa cláusula fará `A` ser instanciada com o valor da expressão. Quando isso acontece, a cláusula é satisfeita. Se `B` ou `C` não for instanciada ou `A` estiver instanciada, a cláusula não será satisfeita e nenhuma instanciação de `A` poderá ocorrer. A semântica de uma proposição **`is`** é consideravelmente diferente de uma sentença de atribuição em uma linguagem imperativa. Essa diferença pode gerar um cenário interessante. Como o operador **`is`** torna a cláusula na qual aparece algo parecido com uma sentença de atribuição, um programador iniciante em Prolog pode ficar tentado a escrever uma sentença como

```
Sum is Sum + Number.
```

que nunca é útil, nem mesmo permitida, em Prolog. Se `Sum` não está instanciada, a referência a ela no lado direito é indefinida e a cláusula falha. Se `Sum` já está instanciada, a cláusula falha porque o operando esquerdo não pode ter uma instanciação atual quando **`is`** for avaliada. Em ambos os casos, a instanciação de `Sum` para o novo valor não ocorrerá. (Se o valor de `Sum + Number` for exigido, ele poderá ser vinculado a algum novo nome.)

Prolog não tem sentenças de atribuição no mesmo sentido das linguagens imperativas. Elas simplesmente não são necessárias para a maior parte da programação para a qual Prolog foi projetado. A utilidade das sentenças de atribuição em linguagens imperativas muitas vezes depende da capacidade do programador de controlar o fluxo de controle de execução do código no qual a sentença de atribuição está inserida. Como esse tipo de controle nem sempre é possível em Prolog, tais sentenças são muito menos úteis.

Como um exemplo simples do uso de computação numérica em Prolog, considere o seguinte problema: suponha que saibamos as velocidades médias de vários automóveis

em determinado autódromo e o tempo em que estão na pista. Essa informação básica pode ser codificada como fatos, e a relação entre velocidade, tempo e distância pode ser escrito como uma regra:

```
speed(ford, 100).
speed(chevy, 105).
speed(dodge, 95).
speed(volvo, 80).
time(ford, 20).
time(chevy, 21).
time(dodge, 24).
time(volvo, 24).
distance(X, Y) :- speed(X, Speed),
                  time(X, Time),
                  Y is Speed * Time.
```

Agora, consultas podem solicitar a distância viajada por determinado carro. Por exemplo, a consulta

```
distance(chevy, Chevy_Distance).
```

instancia `Chevy_Distance` com o valor 2205. As duas primeiras cláusulas no lado direito da sentença de computação da distância instanciam as variáveis `Speed` e `Time` com os valores correspondentes do functor do automóvel dado. Após satisfazer o objetivo, Prolog também mostra o nome `Chevy_Distance` e o seu valor.

Nesse ponto, é importante ter uma visão operacional de como um sistema Prolog produz resultados. O Prolog tem uma estrutura predefinida, chamada `trace`, que mostra as instanciações de valores a variáveis em cada um dos passos durante a tentativa de satisfazer um objetivo. Essa estrutura é usada para entender e depurar programas Prolog. Para entender `trace`, é melhor introduzir um modelo diferente de execução de programas Prolog, chamado de **modelo de rastreamento**.

O modelo de rastreamento descreve a execução de Prolog com base em quatro eventos: (1) chamar, que ocorre no início de uma tentativa de satisfazer um objetivo; (2) sair, quando um objetivo foi satisfeito; (3) refazer, quando um retorno faz com que seja feita uma tentativa de satisfazer novamente um objetivo; e (4) falhar, quando um objetivo falha. A chamada e a saída podem ser relacionadas diretamente ao modelo de execução de um subprograma em uma linguagem imperativa se processos como `distance` são considerados subprogramas. Os outros dois eventos são exclusivos para os sistemas de programação lógica. No seguinte exemplo de rastreamento, um rastreamento da computação do valor para `Chevy_Distance`, o objetivo não requer nenhum evento refazer ou falhar:

```
trace.
distance(chevy, Chevy_Distance).

(1) 1 Call: distance(chevy, _0)?
(2) 2 Call: speed(chevy, _5)?
```

```
(2) 2 Exit: speed(chevy, 105)
(3) 2 Call: time(chevy, _6)?
(3) 2 Exit: time(chevy, 21)
(4) 2 Call: _0 is 105*21?
(4) 2 Exit: 2205 is 105*21
(1) 1 Exit: distance(chevy, 2205)

Chevy_Distance = 2205
```

Símbolos no rastreamento que iniciam com o caractere sublinhado (_) são variáveis internas usadas para valores instanciados. A primeira coluna do rastreamento indica o subobjetivo para o qual um casamento é tentado. No rastreamento de exemplo, a primeira linha com a indicação (3) é uma tentativa de instanciar a variável temporária _6 com um valor de tempo (time) para chevy, em que o tempo é o segundo termo no lado direito da sentença que descreve a computação da distância (distance). A segunda coluna indica a profundidade da chamada do processo de casamento. A terceira coluna indica a ação atual.

Para ilustrar o rastreamento para trás, considere a seguinte base de dados de exemplo e o objetivo composto rastreado:

```
likes(jake, chocolate).
likes(jake, apricots).
likes(darcie, licorice).
likes(darcie, apricots).

trace.
likes(jake, X), likes(darcie, X).

(1) 1 Call: likes(jake, _0)?
(1) 1 Exit: likes(jake, chocolate)
(2) 1 Call: likes(darcie, chocolate)?
(2) 1 Fail: likes(darcie, chocolate)
(1) 1 Redo: likes(jake, _0)?
(1) 1 Exit: likes(jake, apricots)
(3) 1 Call: likes(darcie, apricots)?
(3) 1 Exit: likes(darcie, apricots)

X = apricots
```

É possível pensar acerca de computações Prolog graficamente, como segue: considere cada objetivo como uma caixa com quatro portas – chamar, falhar, sair, refazer. O controle entra em um objetivo na direção para frente, por sua porta chamar. O controle também pode entrar em um objetivo a partir da direção inversa, por sua porta refazer. Além disso, pode deixar um objetivo de duas maneiras: se o objetivo for bem-sucedido, o controle deixa-o pela porta sair; se o objetivo falhou, o controle deixa-o pela porta falhar. Um modelo desse exemplo é mostrado na Figura 16.1. Nele, o controle flui duas vezes por cada subobjetivo. O segundo subobjetivo falha na primeira vez, o que força um retorno pela porta refazer para o primeiro subobjetivo.

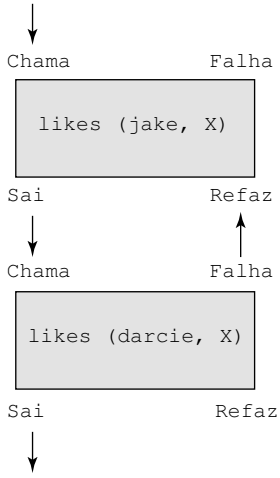


FIGURA 16.1
O modelo de fluxo de controle para o objetivo `likes (jake, X), likes (darcie, X).`

16.6.7 Estruturas de listas

Até agora, a única estrutura de dados Prolog que discutimos foi a proposição atômica, que se parece mais com uma chamada à função do que com uma estrutura de dados. Proposições atômicas, também chamadas de estruturas, são na verdade uma forma de registros. A outra estrutura de dados básica suportada é a lista. Listas são sequências de qualquer número de elementos, em que os elementos podem ser átomos, proposições atômicas ou quaisquer outros termos, inclusive outras listas.

Prolog usa a sintaxe de ML e Haskell para especificar listas. Os elementos de lista são separados por vírgulas, e a lista inteira é delimitada por colchetes, como em

```
[apple, prune, grape, kumquat]
```

A notação `[]` é usada para denotar a lista vazia. Em vez de ter funções explícitas para construir e manipular listas, Prolog simplesmente usa uma notação especial. `[X | Y]` denota uma lista com cabeça `X` e cauda `Y`, em que a cabeça e a cauda correspondem a `CAR` e `CDR` em LISP. Isso é similar à notação usada em ML e Haskell.

Uma lista pode ser criada com uma estrutura simples, como em

```
new_list([apple, prune, grape, kumquat]).
```

que diz que a lista constante `[apple, prune, grape, kumquat]` é um novo elemento da relação chamada `new_list` (nome que inventamos). Essa sentença não vincula a lista a uma variável chamada `new_list`; em vez disso, faz o tipo de coisa que a proposição

```
male(jake)
```

faz. Isto é, ela diz que `[apple, prune, grape, kumquat]` é um novo elemento de `new_list`. Logo, poderíamos ter uma segunda proposição com um argumento de lista, como

```
new_list([apricot, peach, pear])
```

No modo de consulta, um dos elementos de `new_list` pode ser separado em cabeça e cauda com

```
new_list([New_List_Head | New_List_Tail]).
```

Se `new_list` foi configurada para ter os dois elementos mostrados, essa sentença instancia `New_List_Head` com a cabeça do primeiro elemento da lista (nesse caso, `apple`) e `New_List_Tail` com a cauda da lista (ou seja, `[prune, grape, kumquat]`). Se isso fizesse parte de um objetivo composto e um rastreamento para trás forçasse uma nova avaliação dele, `New_List_Head` e `New_List_Tail` seriam novamente instanciadas para `apricot` e `[peach, pear]`, respectivamente, porque `[apricot,peach,pear]` é o próximo elemento de `new_list`.

O operador `|` usado para manipular listas também pode ser utilizado para criar listas a partir de componentes cabeça e cauda que foram instanciados e fornecidos, como em

```
[Element_1 | List_2]
```

Se `Element_1` tivesse sido instanciado com `pickle` e `List_2` tivesse sido instanciada com `[peanut, prune, popcorn]`, a mesma notação criaria, para essa referência, a lista `[pickle, peanut, prune, popcorn]`.

Conforme mencionado, a notação de lista que inclui o símbolo `|` é universal: ela pode especificar uma construção ou uma separação de lista. Note que as seguintes sentenças são equivalentes:

```
[apricot, peach, pear | []]
[apricot, peach | [pear]]
[apricot | [peach, pear]]
```

Com listas, certas operações básicas são geralmente necessárias, como aquelas encontradas em LISP, ML e Haskell. Como exemplo de tais operações em Prolog, examinamos uma definição de `append`, relacionada a tal função em LISP. Nesse exemplo podem ser vistas as diferenças e similaridades entre linguagens funcionais e declarativas. Não precisamos especificar como Prolog deve construir uma nova lista a partir de listas dadas; em vez disso, precisamos definir apenas as características de uma nova lista em termos de listas dadas.

Na aparência, a definição de Prolog de `append` é bastante similar à versão de ML que aparece no Capítulo 15, e um tipo de recursão na resolução é usado de maneira similar para produzir a nova lista. No caso de Prolog, a recursão é causada e controlada pelo processo de resolução. Assim como em ML e Haskell, um processo de casamento de padrões é usado para escolher, baseado no parâmetro real, entre duas definições diferentes do processo de inserção.

Os primeiros dois parâmetros para a operação `append` no código a seguir são as duas listas a serem concatenadas, e o terceiro parâmetro é a lista resultante:

```
append([], List, List).
append([Head | List_1], List_2, [Head | List_3]) :-
            append(List_1, List_2, List_3).
```

A primeira proposição especifica que quando a lista vazia é inserida no final de qualquer outra lista, esta é o resultado. Essa sentença corresponde ao passo de término de recursão da função `append` de ML. Note que a proposição de término é colocada antes da proposição de recursão. Isso é feito porque sabemos que Prolog casará as duas proposições em ordem, começando com a primeira (devido ao seu uso de ordem primeiro em profundidade).

A segunda proposição especifica diversas características da nova lista. Ela corresponde ao passo de recursão na função de ML. O predicado do lado esquerdo diz que o primeiro elemento da nova lista é igual ao primeiro elemento da primeira lista, porque ambos são chamados `Head`. Sempre que `Head` for instanciado para um valor, todas as suas ocorrências no objetivo serão, na prática, simultaneamente instanciadas para esse valor. O lado direito da segunda sentença especifica que a cauda da primeira lista dada (`List_1`) possui a segunda lista (`List_2`) anexada a ela para formar a cauda (`List_3`) da lista resultante.

Um modo de ler a segunda sentença de `append` é: anexar a lista `[Head | List_1]` a qualquer lista `List_2` produz a lista `[Head | List_3]`, mas somente se a lista `List_3` for formada pela anexação de `List_1` a `List_2`. Em LISP, isso poderia ser

```
(CONS (CAR FIRST) (APPEND (CDR FIRST) SECOND))
```

Nas versões em Prolog e LISP, a lista resultante não é construída até a recursão produzir a condição de término; nesse caso, a primeira lista deve se tornar vazia. Então, a lista resultante é criada por meio da função `append` propriamente dita; os elementos retirados da primeira lista são adicionados, na ordem inversa, na segunda lista. A inversão é feita pela recursão.

Uma diferença fundamental entre a `append` de Prolog e as de LISP e ML é que a primeira é um predicado – ela não retorna uma lista, mas `yes` ou `no`. A nova lista é o valor de seu terceiro parâmetro.

Para ilustrar como o processo `append` progride, considere o seguinte exemplo rastreado:

```
trace.
append([bob, jo], [jake, darcie], Family).

(1) 1 Call: append([bob, jo], [jake, darcie], _10)?
(2) 2 Call: append([jo], [jake, darcie], _18)?
(3) 3 Call: append([], [jake, darcie], _25)?
(3) 3 Exit: append([], [jake, darcie], [jake, darcie])
```

```
(2) 2 Exit: append([jo], [jake, darcie], [jo, jake,
                    darcie])
(1) 1 Exit: append([bob, jo], [jake, darcie],
                   [bob, jo, jake, darcie])
Family = [bob, jo, jake, darcie]
yes
```

As primeiras duas chamadas, que representam subobjetivos, possuem `List_1` não vazia, então elas criam chamadas recursivas a partir do lado direito da segunda sentença. O lado esquerdo da segunda sentença efetivamente especifica os argumentos para as chamadas recursivas, ou objetivos, separando a primeira lista, um elemento por passo. Quando a primeira lista fica vazia, em uma chamada, ou subobjetivo, a instância atual do lado direito da segunda sentença ocorre pelo casamento com a primeira. O efeito disso é retornar como terceiro parâmetro o valor da lista vazia, adicionado à lista original passada como segundo parâmetro. Em saídas sucessivas, as quais representam casamentos bem-sucedidos, os elementos que foram removidos da primeira lista são anexados à lista resultante, `Family`. Quando a saída do primeiro objetivo é realizada, o processo está completo e a lista resultante é mostrada.

Outra diferença entre a `append` de Prolog e as de LISP e ML é que a primeira é mais flexível. Por exemplo, em Prolog podemos usar `append` para determinar quais duas listas podem ser anexadas para obtermos `[a, b, c]` com

```
append(X, Y, [a, b, c]).
```

Isso resulta no seguinte:

```
X = []
Y = [a, b, c]
```

Se digitarmos um ponto e vírgula nessa saída, obteremos o resultado alternativo:

```
X = [a]
Y = [b, c]
```

Continuando, obteremos o seguinte:

```
X = [a, b]
Y = [c];
X = [a, b, c]
Y = []
```

O predicado `append` também pode ser usado para criar outras operações de lista, como a seguinte, cujo efeito convidamos o leitor a determinar. Note que `list_op_2` deve ser usada fornecendo-se uma lista como seu primeiro parâmetro e uma variável como o segundo, e o resultado dela é o valor para o qual o segundo parâmetro é instanciado.

```
list_op_2([], []).
list_op_2([Head | Tail], List) :-
list_op_2(Tail, Result), append(Result, [Head], List).
```

Conforme você provavelmente conseguiu determinar, `list_op_2` faz o sistema Prolog instanciar seu segundo parâmetro com uma lista que tem os elementos da lista do primeiro parâmetro, mas na ordem inversa. Por exemplo, `([apple, orange, grape], Q)` instancia `Q` com a lista `[grape, orange, apple]`.

Mais uma vez, apesar de as linguagens LISP e Prolog serem fundamentalmente diferentes, operações similares podem usar estratégias similares. No caso da operação de inversão, tanto `list_op_2` de Prolog quanto a função `reverse` de LISP incluem a condição de término da recursão, com o processo básico de inserir o inverso do `CDR` ou cauda da lista ao `CAR` ou à cabeça da lista para criar a lista resultante.

A seguir, temos um rastreamento desse processo, agora chamado `reverse`:

```
trace.
reverse([a, b, c], Q).

(1) 1 Call: reverse([a, b, c], _6)?
(2) 2 Call: reverse([b, c], _65636)?
(3) 3 Call: reverse([c], _65646)?
(4) 4 Call: reverse([], _65656)?
(4) 4 Exit: reverse([], [])
(5) 4 Call: append([], [c], _65646)?
(5) 4 Exit: append([], [c], [c])
(3) 3 Exit: reverse([c], [c])
(6) 3 Call: append([c], [b], _65636)?
(7) 4 Call: append([], [b], _25)?
(7) 4 Exit: append([], [b], [b])
(6) 3 Exit: append([c], [b], [c, b])
(2) 2 Exit: reverse([b, c], [c, b])
(8) 2 Call: append([c, b], [a], _6)?
(9) 3 Call: append([b], [a], _32)?
(10) 4 Call: append([], [a], _39)?
(10) 4 Exit: append([], [a], [a])
(9) 3 Exit: append([b], [a], [b, a])
(8) 2 Exit: append([c, b], [a], [c, b, a])
(1) 1 Exit: reverse([a, b, c], [c, b, a])

Q = [c, b, a]
```

Suponha que precisássemos determinar se um símbolo está em uma lista. Uma descrição direta disso em Prolog seria

```
member(Element, [Element | _]).
member(Element, [_ | List]) :- member(Element, List).
```

O sublinhado indica uma variável "anônima"; ela significa que não nos importamos com que instanciação ela poderia obter com a unificação. A primeira sentença no exemplo anterior será bem-sucedida se `Element` for a cabeça da lista, seja inicialmente ou após diversas recursões por meio do segundo elemento. A segunda sentença será bem-sucedida se `Element` estiver na cauda da lista. Considere os exemplos rastreados:

```
trace.
member(a, [b, c, d]).
(1) 1 Call: member(a, [b, c, d])?
(2) 2 Call: member(a, [c, d])?
(3) 3 Call: member(a, [d])?
(4) 4 Call: member(a, [])?
(4) 4 Fail: member(a, [])
(3) 3 Fail: member(a, [d])
(2) 2 Fail: member(a, [c, d])
(1) 1 Fail: member(a, [b, c, d])
no

member(a, [b, a, c]).
(1) 1 Call: member(a, [b, a, c])?
(2) 2 Call: member(a, [a, c])?
(2) 2 Exit: member(a, [a, c])
(1) 1 Exit: member(a, [b, a, c])
Yes
```

16.7 DEFICIÊNCIAS DE PROLOG

Apesar de Prolog ser uma ferramenta útil, não é uma linguagem de programação lógica pura nem perfeita. Esta seção descreve alguns dos seus problemas.

16.7.1 Controle da ordem de resolução

Prolog, por razões de eficiência, permite que o usuário controle a ordem do casamento de padrões durante a resolução. Em um ambiente de programação lógica puro, a ordem dos casamentos tentados que ocorrem durante a resolução é não determinística, e todos os casamentos podem ser tentados concorrentemente. Entretanto, como Prolog sempre casa na mesma ordem, começando no início da base de dados e no final esquerdo de um objetivo, o usuário pode afetar profundamente a eficiência ordenando que as sentenças da base de dados otimizem uma aplicação específica. Por exemplo, se o usuário sabe que certas regras são muito mais propensas a serem bem-sucedidas que outras durante determinada "execução", o programa pode ser mais eficiente se essas regras forem colocadas no início da base de dados.

Além de permitir que o usuário controle a ordem da base de dados e dos subobjetivos, Prolog, em outra concessão à eficiência, permite algum controle explícito do rastreamento para trás. Isso é feito com o operador de corte, especificado por um ponto de exclamação (`!`). Na verdade, o operador de corte é um objetivo, não um operador. Como um objetivo, ele sempre é bem-sucedido de imediato, mas não pode ser novamente satisfeito por rastreamento para trás. Logo, um efeito colateral do corte é que o subobjetivo à sua esquerda em um objetivo composto também não pode ser novamente satisfeito por rastreamento para trás. Por exemplo, no objetivo

```
a, b, !, c, d.
```

se tanto `a` quanto `b` forem bem-sucedidos, mas `c` falhar, o objetivo completo falhará. Esse objetivo seria usado caso se soubesse que sempre que `c` falha é inútil tentar satisfazer novamente `b` ou `a`.

O propósito do corte, então, é permitir que o usuário torne os programas mais eficientes, dizendo ao sistema quando ele não deve tentar satisfazer novamente subobjetivos que presumivelmente não podem resultar em uma prova completa.

Como exemplo do uso do operador de corte, considere as regras `member` da Seção 16.6.7:

```
member(Element, [Element | _]).
member(Element, [_ | List]) :- member(Element, List).
```

Se a lista passada como parâmetro para `member` representa um conjunto, ela só pode ser satisfeita uma vez (conjuntos não contêm elementos duplicados). Logo, se `member` é usada como subobjetivo em uma sentença de objetivo com vários subobjetivos, pode haver um problema. O problema é que se `member` for bem-sucedida, mas o próximo subobjetivo falhar, o rastreamento para trás tentará novamente satisfazer `member` ao continuar um casamento anterior. Mas, devido ao fato de a lista passada como parâmetro para `member` ter apenas uma cópia do elemento para começar, `member` não pode ser bem-sucedida novamente, o que finalmente faz todo o objetivo falhar, apesar de quaisquer tentativas adicionais de satisfazer `member` novamente. Por exemplo, considere:

```
dem_candidate(X) :- member(X, democrats), tests(X).
```

Esse objetivo determina se uma pessoa é democrata e se é uma boa candidata para concorrer a determinada posição. O subobjetivo `tests` verifica uma variedade de características do democrata informado para determinar sua adequação ao cargo. Caso o conjunto de democratas não tenha duplicatas, não precisamos voltar ao subobjetivo `member` se o subobjetivo `tests` falhar, porque `member` procurará todos os outros democratas e falhará, já que não existem duplicatas. A segunda tentativa ao subobjetivo `member` será uma perda de tempo computacional. A solução para essa ineficiência é adicionar um lado direito à primeira sentença da definição de `member`, com o operador de corte como único elemento, como em

```
member(Element, [Element | _]) :- !.
```

O rastreamento para trás não tentará satisfazer novamente `member`, mas fará o subobjetivo inteiro falhar.

O corte é particularmente útil em uma estratégia de programação em Prolog chamada **gerar e testar**. Em subprogramas que usam essa estratégia, o objetivo consiste em subobjetivos que geram soluções em potencial, as quais são então verificadas por subobjetivos posteriores do tipo "teste". Soluções rejeitadas exigem rastreamento para trás para subobjetivos de "geração", os quais geram novas soluções em potencial. Como exemplo de programa gerar e testar, considere o seguinte, que aparece em Clocksin e Mellish (2003):

```
divide(N1, N2, Result) :- is_integer(Result),
                          Product1 is Result * N2,
                          Product2 is (Result + 1) * N2,
                          Product1 =< N1, Product2 >
                            N1, !.
```

Esse programa efetua divisão inteira usando adição e multiplicação. Como a maioria dos sistemas Prolog fornece divisão como um operador, esse programa na verdade não é útil, exceto para ilustrar um programa simples de geração e teste.

O predicado `is_integer` é bem-sucedido desde que seu parâmetro possa ser instanciado para algum valor não negativo. Se seu argumento não estiver instanciado, `is_integer` instancia seu valor como 0. Se o argumento estiver instanciado para um inteiro, `is_integer` o instanciará para o próximo valor superior inteiro.

Então, em `divide`, `is_integer` é o subobjetivo gerador. Ele gera elementos da sequência 0, 1, 2, ..., um a cada vez que for satisfeito. Todos os outros subobjetivos são de testes – eles verificam na tentativa de determinar se os valores produzidos por `is_integer` são, na verdade, a divisão dos dois primeiros parâmetros, `N1` e `N2`. A finalidade do corte como último subojetivo é simples: ele impede que `divide` tente encontrar uma solução alternativa, uma vez que tenha encontrado *a* solução. Apesar de `is_integer` poder gerar um grande número de candidatos, apenas um é a solução; então, o corte aqui evita tentativas desnecessárias de produzir soluções secundárias.

O uso do operador de corte tem sido comparado ao de desvios incondicionais (gotos) em linguagens imperativas (van Emden, 1980). Apesar de ser algumas vezes necessário, é possível abusar dele. De fato, ele é eventualmente usado para fazer programas lógicos terem um fluxo de controle inspirado em estilos de programação imperativa.

A capacidade de modificar indevidamente o fluxo de controle em um programa Prolog é uma deficiência, porque é diretamente prejudicial a uma das vantagens mais importantes da programação lógica – a de que os programas não especificam como as soluções devem ser encontradas. Em vez disso, eles simplesmente determinam como seu aspecto deve ser. Esse projeto facilita a escrita e a leitura dos programas. Eles não são entremeados com os detalhes de como as soluções devem ser determinadas e, em particular, qual a ordem precisa das computações que devem ser feitas para produzir a solução. Então, embora a programação lógica não exija modificações no fluxo de controle, os programas Prolog geralmente as usam, em sua maioria devido à eficiência.

16.7.2 A premissa do mundo fechado

A natureza da resolução em Prolog algumas vezes produz resultados enganosos. As únicas verdades, no que diz respeito ao Prolog, são aquelas que podem ser provadas por meio de sua base de dados. Ele não tem nenhum conhecimento, exceto sua base de dados. Quando o sistema recebe uma consulta e a base de dados não tem informações para prová-la de forma absoluta, presume-se que a consulta é falsa. O Prolog pode provar que um objetivo dado é verdadeiro, mas não pode prová-lo como falso. Ele simplesmente presume que, como não pode provar que um objetivo é verdadeiro, esse objetivo deve ser

falso. Em sua essência, o Prolog é um sistema verdadeiro/falha, em vez de um sistema verdadeiro/falso.

Na verdade, a premissa do mundo fechado não deve ser de todo estranha a você – nosso sistema judicial funciona da mesma maneira. Os suspeitos são inocentes até que se prove o contrário. Eles não precisam ser provados inocentes. Se um julgamento não pode provar que uma pessoa é culpada, ela é considerada inocente.

O problema da premissa do mundo fechado está relacionada ao problema da negação, discutido na subseção a seguir.

16.7.3 O problema da negação

Outro problema do Prolog é sua dificuldade com a negação. Considere a seguinte base de dados de dois fatos e um relacionamento:

```
parent(bill, jake).
parent(bill, shelley).
sibling(X, Y) :- (parent(M, X), parent(M, Y).
```

Agora, suponha que digitássemos a consulta

```
sibling(X, Y).
```

O Prolog responderia com

```
X = jake
Y = jake
```

Então, o Prolog "pensa" que `jake` é irmão (`sibling`) dele próprio. Isso acontece porque o sistema primeiro instancia `M` com `bill` e `X` com `jake` para tornar o primeiro subobjetivo, `parent(M, X)`, verdadeiro. Então, ele começa no princípio da base de dados novamente, para casar o segundo subobjetivo, `parent(M, Y)`, e chega às instanciações de `M` com `bill` e `Y` com `jake`. Como os dois subobjetivos são satisfeitos independentemente, com ambos os casamentos começando no princípio da base de dados, as respostas mostradas aparecem. Para evitar esse resultado, `X` deve ser especificado como irmão de `Y` apenas se eles tiverem os mesmos pais (`parents`) *e* não forem os mesmos. Infelizmente, determinar que eles não são iguais não é simples em Prolog, como iremos discutir. O método mais exato exigiria adicionar um fato para cada par de átomos, estabelecendo que eles não são a mesma coisa. Isso, é claro, torna a base de dados muito grande, geralmente com mais informação negativa que positiva. Por exemplo, as pessoas têm 364 mais dias de "não aniversário" que de aniversário.

Uma solução alternativa simples é dizer no objetivo que `X` não deve ser o mesmo que `Y`, como em

```
sibling(X, Y) :- parent(M, X), parent(M, Y), not(X = Y).
```

Em outras situações, a solução não é tão simples.

O operador `not` de Prolog será satisfeito nesse caso se a resolução não puder satisfazer o subobjetivo `X = Y`. Logo; se `not` for bem-sucedido, isso não necessariamente

significa que X não é igual a Y; em vez disso, significa que a resolução não pode provar, a partir da base de dados, que X é o mesmo que Y. Logo, o operador not em Prolog não é equivalente ao operador lógico NÃO, pois NÃO significa que o operando pode ser provado como verdadeiro. Essa não equivalência pode levar a um problema se tivermos um objetivo da forma

```
not(not(some_goal)).
```

que seria equivalente a

```
some_goal.
```

se o operador not de Prolog fosse um operador lógico NÃO. Em alguns casos, entretanto, eles não são a mesma coisa. Por exemplo, considere novamente as regras de member:

```
member(Element, [Element | _]) :- !.
member(Element, [_ | List]) :- member(Element, List).
```

Para descobrir um dos elementos de uma lista, poderíamos usar o objetivo

```
member(X, [mary, fred, barb]).
```

que faria X ser instanciado com mary, que por sua vez seria impresso. Mas se usássemos

```
not(not(member(X, [mary, fred, barb]))).
```

ocorreria a seguinte sequência de eventos: primeiro, o objetivo interno seria bem-sucedido, instanciando X como mary. Então, o Prolog tentaria satisfazer o próximo objetivo:

```
not(member(X, [mary, fred, barb])).
```

Essa sentença falharia porque member seria bem-sucedido. Quando esse objetivo falhasse, X teria sua instância removida, porque o Prolog sempre remove as instâncias de todas as variáveis em todos os objetivos que falham. A seguir, o Prolog tentaria satisfazer o objetivo externo not, que seria bem-sucedido, porque seu argumento falhou. Por fim, o resultado, que é X, seria impresso. Mas X não estaria atualmente instanciado, então o sistema indicaria isso. Geralmente, variáveis não instanciadas são impressas na forma de uma cadeia de dígitos precedida por um sublinhado. Então, o fato de o not de Prolog não ser equivalente ao NÃO lógico pode ser, no mínimo, enganoso.

A razão fundamental pela qual o NÃO lógico não pode ser uma parte integrante de Prolog é a forma da cláusula de Horn:

$$A \text{ :- } B_1 \cap B_2 \cap \ldots \cap B_n$$

Se todas as proposições B forem verdadeiras, pode-se concluir que A é verdadeira. Mas, independentemente da veracidade ou da falsidade de qualquer um dos Bs, não se pode provar que A é falso. A partir de lógica positiva, alguém pode concluir apenas lógica negativa. Então, o uso da forma da cláusula de Horn evita quaisquer conclusões negativas.

16.7.4 Limitações intrínsecas

Um objetivo fundamental da programação lógica, conforme descrito na Seção 16.4, é fornecer programação não procedural, ou seja, um sistema no qual os programadores especificam o que um programa deve fazer, mas não precisam especificar como isso deve ser feito. O exemplo dado lá é reescrito aqui:

sort(old_list, new_list) ⊂ permute(old_list, new_list) ∩ sorted(new_list)
sorted(list) ⊂ ∀j such that 1 ≤ j < n, list(j) ≤ list (j+1)

É simples escrever isso em Prolog. Por exemplo, o subobjetivo ordenado pode ser expresso como

```
sorted ([]).
sorted ([x]).
sorted ([x, y | list]) :- x <= y, sorted ([y | list]).
```

O problema desse processo de ordenação é que ele não tem ideia de como ordenar, além de simplesmente enumerar todas as permutações de uma lista até que aconteça de ser criada uma que tenha a lista ordenada – um processo muito lento.

Até agora ninguém descobriu um processo pelo qual a descrição de uma lista ordenada possa ser transformada em algum algoritmo eficiente para a ordenação. A resolução é capaz de muitas coisas interessantes, mas certamente não essa. Logo, um programa em Prolog que ordena uma lista deve especificar os detalhes de como essa ordenação pode ser feita, como é o caso nas linguagens imperativas e funcionais.

Todos esses problemas significam que a programação lógica deve ser abandonada? De forma alguma! Como é atualmente, ela é capaz de lidar com muitas aplicações úteis. Além disso, é baseada em um conceito intrigante e, dessa forma, é interessante por si só. Por fim, existe a possibilidade de desenvolvimentos de novas técnicas de inferência que permitirão a um sistema de linguagem de programação lógica tratar de classes de problemas progressivamente maiores.

16.8 APLICAÇÕES DE PROGRAMAÇÃO LÓGICA

Nesta seção, descrevemos brevemente algumas das maiores classes de aplicações em potencial da programação lógica em geral e de Prolog em particular.

16.8.1 Sistemas de gerenciamento de bases de dados relacionais

Sistemas de gerenciamento de bases de dados relacionais (SGBDRs) armazenam dados em forma de tabelas. Consultas em tais tabelas são geralmente descritas em uma linguagem de consulta chamada linguagem de consulta estruturada (SQL – Structured Query Language). SQL é não procedural no mesmo sentido que a programação lógica é não procedural. O usuário não descreve como obter a resposta; em vez disso, descreve apenas as características da resposta. A conexão entre programação lógica e SGBDRs deve ser óbvia. Tabelas simples de informação podem ser descritas por estruturas Prolog

e relacionamentos entre tabelas podem ser fácil e convenientemente descritos por regras Prolog. O processo de recuperação é inerente à operação de resolução. As sentenças de objetivo de Prolog fornecem as consultas para o SGBDR. A programação lógica é então uma união natural às necessidades de implementar um SGBDR.

Uma das vantagens de usar programação lógica para implementar um SGBDR é que apenas uma linguagem é necessária. Em um SGBDR típico, uma linguagem de base de dados inclui sentenças para definição de dados, manipulação de dados e consultas, todas as quais estão embutidas em uma linguagem de programação de propósito geral, como COBOL. A linguagem de propósito geral é usada para processar os dados e funções de entrada e de saída. Todas essas funções podem ser feitas em uma linguagem de programação lógica.

Outra vantagem do uso de programação lógica para implementar um SGBDR é que a capacidade de dedução é predefinida. Os SGBDRs convencionais não podem deduzir nada a partir de uma base de dados além daquilo que é armazenado explicitamente neles. Eles contêm apenas fatos, em vez de fatos *e* regras de inferência. A principal desvantagem de usar programação lógica para um SGBDR, comparado com um SGBDR convencional, é que a implementação em programação lógica é mais lenta. As inferências lógicas levam mais tempo que métodos de busca em tabelas normais usando técnicas de programação imperativa.

16.8.2 Sistemas especialistas

Sistemas especialistas são sistemas computacionais projetados para emular especialidades humanas em algum domínio em particular. Eles consistem em uma base de dados de fatos, um processo dse inferência, algumas heurísticas acerca do domínio e alguma interface amigável com o usuário que faz o sistema se parecer mais com um consultor humano especialista. Além de possuírem sua base de dados de conhecimento inicial, que é fornecida por um especialista humano, os sistemas especialistas aprendem a partir do processo de sua utilização, então suas bases devem ser capazes de crescer dinamicamente. Além disso, um sistema especialista deve incluir a capacidade de interrogar o usuário para obter informação adicional quando ele determinar que isso é necessário.

Um dos problemas centrais para o projetista de um sistema especialista é tratar das inconsistências e incompletudes inevitáveis da base de dados. A programação lógica parece ser bem adequada para tratar desses problemas. Por exemplo, regras de inferência padrão podem ajudar no problema da incompletude.

O Prolog pode e tem sido usado para construir sistemas especialistas, podendo facilmente satisfazer suas necessidades básicas. Isso ocorre por meio do uso da resolução como base para o processamento de consultas, da utilização da sua habilidade de adicionar fatos e regras para fornecer a capacidade de aprendizagem, e ainda do emprego de seu recurso de rastreamento para informar o usuário acerca do "pensamento" por trás de um resultado. Falta ao Prolog a capacidade automática do sistema de solicitar ao usuário informações adicionais, quando são necessárias.

Um dos usos mais conhecidos de programação lógica em sistemas especialistas é chamado APES, descrito em Sergot (1983) e Hammond (1983). O sistema APES inclui um recurso muito flexível para obter informações do usuário durante a construção de um sistema especialista. Inclui também um segundo interpretador que produz explicações para suas respostas a consultas.

O APES é usado de maneira bem-sucedida para produzir diversos sistemas especialistas, inclusive um para as regras do programa de benefícios sociais e um para o British Nationality Act, fonte definitiva de regras de cidadania britânica.

16.8.3 Processamento de linguagem natural

Certos tipos de processamento de linguagem natural podem ser feitos com programação lógica. Em particular, interfaces de linguagem natural para sistemas de software de computadores, como bases de dados inteligentes e outros sistemas baseados em conhecimento, podem ser feitas convenientemente com ela. Para descrever a sintaxe de linguagens, formas de programação lógica foram descobertas como equivalentes às gramáticas livres de contexto. Procedimentos de prova em linguagens de programação lógica foram descobertos como equivalentes a certas estratégias de análise sintática. Na verdade, a resolução de encadeamento para trás pode ser usada diretamente para analisar sintaticamente sentenças cujas estruturas são descritas por gramáticas livres de contexto. Foi descoberto também que alguns tipos de semântica de linguagens naturais podem se tornar claros por meio da modelagem das linguagens com programação lógica. Em particular, a pesquisa em redes semânticas baseadas em lógica mostrou que conjuntos de sentenças em linguagem natural podem ser expressos em forma clausal (Deliyanni e Kowalski, 1979). Kowalski (1979) também discute redes semânticas baseadas em lógica.

RESUMO

A lógica simbólica fornece a base para a programação lógica e para as linguagens de programação lógica. A estratégia da programação lógica é usar como base de dados uma coleção de fatos e regras que definem relacionamentos entre fatos, e usar um processo automático de inferência para verificar a validade de novas proposições, supondo que os fatos e as regras da base de dados sejam verdadeiros. Essa estratégia é a desenvolvida para a prova automática de teoremas.

Prolog é a linguagem de programação lógica mais utilizada. As origens desse tipo de linguagem advêm do desenvolvimento da regra de resolução para inferência lógica de Robinson. Prolog foi desenvolvida principalmente por Colmeraeur e Roussel, em Marselha, com alguma ajuda de Kowalski, em Edimburgo.

Os programas lógicos são não procedurais. Isso significa que as características da solução são dadas, mas o processo de obter a solução não.

As sentenças Prolog são fatos, regras ou objetivos. A maioria é constituída de estruturas, que são proposições atômicas, e operadores lógicos, apesar de expressões aritméticas também serem permitidas.

A resolução é a principal atividade de um interpretador Prolog. Esse processo, que usa rastreamento para trás extensivamente, envolve principalmente o casamento de padrões entre proposições. Quando variáveis estão envolvidas, elas podem ser instanciadas para valores para fornecer casamentos. Esse processo de instanciação é chamado de *unificação*.

Existem diversos problemas no estado atual da programação lógica. Por questões de eficiência, e mesmo para evitar laços de repetição infinitos, algumas vezes os progra-

madores devem declarar informações de fluxo de controle em seus programas. Existem também os problemas da premissa do mundo fechado e da negação.

A programação lógica é usada em algumas áreas diferentes, principalmente em sistemas de banco de dados relacionais, sistemas especialistas e processamento de linguagem natural.

NOTAS BIBLIOGRÁFICAS

A linguagem Prolog é descrita em diversos livros. A forma de Edimburgo da linguagem é abordada em Clocksin e Mellish (2003). A implementação para microcomputadores é descrita em Clark e McCabe (1984).

Hogger (1991) é um livro excelente sobre o tópico geral de programação lógica; é a fonte da seção sobre aplicações de programação lógica deste capítulo.

QUESTÕES DE REVISÃO

1. Quais são os três principais usos de lógica simbólica na lógica formal?
2. Quais são as duas partes de um termo composto?
3. Quais são os dois modos pelos quais uma proposição pode ser definida?
4. Qual é a forma geral de uma proposição em uma forma clausal?
5. O que são antecedentes? E consequências?
6. Escreva definições gerais (não rigorosas) de *resolução* e *unificação*.
7. Quais são as formas das cláusulas de Horn?
8. Qual é o conceito básico da semântica declarativa?
9. O que significa para uma linguagem ser não procedural?
10. Quais são as três formas de um termo Prolog?
11. O que é uma variável não instanciada?
12. Quais são as formas sintáticas e o uso de sentenças de fatos e de regras em Prolog?
13. O que é uma conjunção?
14. Explique as duas estratégias para casar objetivos a fatos em uma base de dados.
15. Explique a diferença entre uma busca primeiro em profundidade e primeiro em amplitude, discutindo como múltiplos objetivos são satisfeitos.
16. Explique como o rastreamento para trás (*backtracking*) funciona em Prolog.
17. Explique o que está errado na sentença Prolog `K is K + 1`.
18. Quais são as duas maneiras pelas quais um programador Prolog pode controlar a ordem do casamento de padrões durante a resolução?

19. Explique a estratégia de programação em Prolog chamada gerar e testar.
20. Explique a premissa de mundo fechado usada por Prolog. Por que ela é uma limitação?
21. Explique o problema da negação em Prolog. Por que ele é uma limitação?
22. Explique a conexão entre a prova automática de teoremas e o processo de inferência de Prolog.
23. Explique a diferença entre linguagens procedurais e não procedurais.
24. Explique por que os sistemas Prolog devem fazer rastreamento para trás.
25. Qual é a relação entre resolução e unificação em Prolog?

PROBLEMAS

1. Compare o conceito de tipagem de dados em C# com o de Prolog.
2. Descreva como uma máquina multiprocessada poderia ser usada para implementar resolução. Prolog, em sua definição atual, poderia usar esse método?
3. Escreva uma descrição em Prolog de sua árvore genealógica (baseada apenas em fatos), retrocedendo a seus avós e incluindo todos os descendentes. Certifique-se de incluir todos os relacionamentos.
4. Escreva um conjunto de regras para relacionamentos familiares, incluindo todos os relacionamentos desde os avós até duas gerações posteriores. Depois acrescente a ele os fatos do Problema 3 e elimine quantos fatos você conseguir.
5. Escreva as seguintes sentenças condicionais, descritas em português, como cláusulas de Horn com cabeça em Prolog:
   a. Se Frederico é o pai de Michael, então Frederico é um ancestral de Michael.
   b. Se Michael é o pai de João e Michael é o pai de Maria, então Maria é a irmã de João.
   c. Se Michael é o irmão de Frederico e Frederico é o pai de Maria, então Michael é o tio de Maria.
6. Explique duas maneiras pelas quais os recursos de processamento de listas de Scheme e de Prolog são similares.
7. De que maneira os recursos de processamento de listas de Scheme e de Prolog são diferentes?
8. Escreva uma comparação de Prolog com ML, incluindo duas similaridades e duas diferenças.
9. Usando um livro sobre Prolog, aprenda e escreva uma descrição de um problema de ocorrência-verificação. Por que Prolog permite que esse problema exista em sua implementação?

10. Encontre uma boa fonte de informações acerca da forma normal de Skolem e escreva uma explanação breve, mas clara, sobre ela.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Usando as estruturas `parent(X, Y)`, `male(X)` e `female(X)`, escreva uma estrutura que defina `mother(X, Y)`.
2. Usando as estruturas `parent(X, Y)`, `male(X)` e `female(X)`, escreva uma estrutura que defina `sister(X, Y)`.
3. Escreva um programa Prolog que encontre o máximo de uma lista de números.
4. Escreva um programa Prolog que seja bem-sucedido se a interseção de duas listas passadas como parâmetros for vazia.
5. Escreva um programa Prolog que retorne uma lista contendo a união dos elementos de duas listas.
6. Escreva um programa Prolog que retorne o último elemento de uma lista.
7. Escreva um programa Prolog que implemente a ordenação rápida (quicksort).

Esta página foi deixada em branco intencionalmente.

Bibliografia

ACM. (1979) "Part A: Preliminary Ada Reference Manual" and "Part B: Rationale for the Design of the Ada Programming Language." SIGPLAN Notices, Vol. 14, No. 6.

ACM. (1993a) "History of Programming Language Conference Proceedings." ACM SIGPLAN Notices, Vol. 28, No. 3, March.

ACM. (1993b) "High Performance FORTRAN Language Specification Part 1." FORTRAN Forum, Vol. 12, No. 4.

Aho, A. V., B. W. Kernighan, and P. J. Weinberger. (1988) The AWK Programming Language. Addison-Wesley, Reading, MA.

Aho, A. V., M. S. Lam, R. Sethi, and J. D. Ullman. (2006) Compilers: Principles, Techniques, and Tools, 2e. Addison-Wesley, Reading, MA.

Andrews, G. R., and F. B. Schneider. (1983) "Concepts and Notations for Concurrent Programming." ACM Computing Surveys, Vol. 15, No. 1, pp. 3–43.

ANSI. (1966) American National Standard Programming Language FORTRAN. American National Standards Institute, New York.

ANSI. (1976) American National Standard Programming Language PL/I. ANSI X3.53–1976. American National Standards Institute, New York.

ANSI. (1978a) American National Standard Programming Language FORTRAN. ANSI X3.9–1978. American National Standards Institute, New York.

ANSI. (1978b) American National Standard Programming Language Minimal BASIC. ANSI X3.60–1978. American National Standards Institute, New York.

ANSI. (1985) American National Standard Programming Language COBOL. ANSI X3.23–1985. American National Standards Institute, New York.

ANSI. (1989) American National Standard Programming Language C. ANSI X3.159–1989. American National Standards Institute, New York.

ANSI. (1992) American National Standard Programming Language FORTRAN 90. ANSI X3. 198–1992. American National Standards Institute, New York.

Arden, B. W., B. A. Galler, and R. M. Graham. (1961) "MAD at Michigan." Datamation, Vol. 7, No. 12, pp. 27–28.

ARM. (1995) Ada Reference Manual. ISO/IEC/ANSI 8652:19. Intermetrics, Cambridge, MA.

Arnold, K., J. Gosling, and D. Holmes. (2006) The Java (TM) Programming Language, 4e. Addison-Wesley, Reading, MA.

Backus, J. (1954) "The IBM 701 Speedcoding System." J. ACM, Vol. 1, pp. 4–6.

Backus, J. (1959) "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference." Proceedings International Conference on Information Processing. UNESCO, Paris, pp. 125–132.

Backus, J. (1978) "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." Commun. ACM, Vol. 21, No. 8, pp. 613–641.

Backus, J., F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. (1963) "Revised Report on the Algorithmic Language ALGOL 60." Commun. ACM, Vol. 6, No. 1, pp. 1–17.

Balena, F. (2003) Programming Microsoft Visual Basic .NET Version 2003, Microsoft Press, Redmond, WA.

Ben-Ari, M. (1982) Principles of Concurrent Programming. Prentice-Hall, Englewood Cliffs, NJ.

Birtwistle, G. M., O.-J. Dahl, B. Myhrhaug, and K. Nygaard. (1973) Simula BEGIN. Van Nostrand Reinhold, New York.

Bodwin, J. M., L. Bradley, K. Kanda, D. Litle, and U. F. Pleban. (1982) "Experience with an Experimental Compiler Generator Based on Denotational Semantics." ACM SIGPLAN Notices, Vol. 17, No. 6, pp. 216–229.

Bohm, C., and G. Jacopini. (1966) "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules." Commun. ACM, Vol. 9, No. 5, pp. 366–371.

Bolsky, M., and D. Korn. (1995) The New KornShell Command and Programming Language. Prentice-Hall, Englewood Cliffs, NJ.

Booch, G. (1987) Software Engineering with Ada, 2e. Benjamin/Cummings, Redwood City, CA.

Bradley, J. C. (1989) QuickBASIC and QBASIC Using Modular Structures. W. C. Brown, Dubuque, IA.

Brinch Hansen, P. (1973) Operating System Principles. Prentice-Hall, Englewood Cliffs, NJ.

Brinch Hansen, P. (1975) "The Programming Language Concurrent-Pascal." IEEE Transactions on Software Engineering, Vol. 1, No. 2, pp. 199–207.

Brinch Hansen, P. (1977) The Architecture of Concurrent Programs. Prentice-Hall, Englewood Cliffs, NJ.

Brinch Hansen, P. (1978) "Distributed Processes: A Concurrent Programming Concept." Commun. ACM, Vol. 21, No. 11, pp. 934–941.

Brown, J. A., S. Pakin, and R. P. Polivka. (1988) APL2 at a Glance. Prentice-Hall, Englewood Cliffs, NJ.

Campione, M., K. Walrath, and A. Huml. (2001) The Java Tutorial, 3e. Addison-Wesley, Reading, MA.

Cardelli, L., J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. (1989) Modula-3 Report (revised). Digital System Research Center, Palo Alto, CA.

Chambers, C., and D. Ungar. (1991) "Making Pure Object-Oriented Languages Practical." SIGPLAN Notices, Vol. 26, No. 1, pp. 1–15.

Chomsky, N. (1956) "Three Models for the Description of Language." IRE Transactions on Information Theory, Vol. 2, No. 3, pp. 113–124.

Chomsky, N. (1959) "On Certain Formal Properties of Grammars." Information and Control, Vol. 2, No. 2, pp. 137–167.

Christiansen, T., B. D. Foy, and L. Wall with J. Orwant. (2013) Programming Perl, 4e. O'Reilly & Associates, Sebastopol, CA.

Church, A. (1941) Annals of Mathematics Studies. Calculi of Lambda Conversion, Vol. 6. Princeton University Press, Princeton, NJ. Reprinted by Klaus Reprint Corporation, New York, 1965.

Clark, K. L., and F. G. McCabe. (1984) Micro-PROLOG: Programming in Logic. Prentice-Hall, Englewood Cliffs, NJ.

Clarke, L. A., J. C. Wileden, and A. L. Wolf. (1980) "Nesting in Ada Is for the Birds." ACM SIGPLAN Notices, Vol. 15, No. 11, pp. 139–145.

Cleaveland, J. C. (1986) An Introduction to Data Types. Addison-Wesley, Reading, MA.

Cleaveland, J. C., and R. C. Uzgalis. (1976) Grammars for Programming Languages: What Every Programmer Should Know About Grammar. American Elsevier, New York.

Clocksin, W. F., and C. S. Mellish. (2013) Programming in Prolog: Using the ISO Standard. Springer-Verlag, New York.

Cohen, J. (1981) "Garbage Collection of Linked Data Structures." ACM Computing Surveys, Vol. 13, No. 3, pp. 341–368.

Converse, T., and J. Park. (2000) PHP 4 Bible. IDG Books, New York.

Conway, M. E. (1963). "Design of a Separable Transition-Diagram Compiler." Commun. ACM, Vol. 6, No. 7, pp. 396–408.

Conway, R., and R. Constable. (1976) "PL/CS—A Disciplined Subset of PL/I." Technical Report TR76/293. Department of Computer Science, Cornell University, Ithaca, NY.

Cornell University. (1977) PL/C User's Guide, Release 7.6. Department of Computer Science, Cornell University, Ithaca, NY.

Correa, N. (1992) “Empty Categories, Chain Binding, and Parsing.” In Principle-Based Parsing, R. C. Berwick, S. P. Abney, and C. Tenny (eds.). Kluwer Academic Publishers, Boston, pp. 83–121.

Cousineau, G., M. Mauny, and K. Callaway. (1998) The Functional Approach to Programming. Cambridge University Press, Cambridge, UK.

Dahl, O.-J., E. W. Dijkstra, and C. A. R. Hoare. (1972) Structured Programming. Academic Press, New York.

Dahl, O.-J., and K. Nygaard. (1967) SIMULA 67 Common Base Proposal. Norwegian Computing Center Document, Oslo.

Deitel, H. M., D. J. Deitel, and T. R. Nieto. (2002) Visual BASIC .Net: How to Program, 2e. Prentice-Hall, Upper Saddle River, NJ.

Deliyanni, A., and R. A. Kowalski. (1979) “Logic and Semantic Networks.” Commun. ACM, Vol. 22, No. 3, pp. 184–192.

Department of Defense. (1960) COBOL, Initial Specifications for a Common Business Oriented Language. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1961) COBOL—1961, Revised Specifications for a Common Business Oriented Language. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1962) COBOL—1961 EXTENDED, Extended Specifications for a Common Business Oriented Language. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1975a) Requirements for High Order Programming Languages, STRAWMAN. July. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1975b) Requirements for High Order Programming Languages, WOODENMAN. August. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1976) Requirements for High Order Programming Languages, TINMAN. June. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1977) Requirements for High Order Programming Languages, IRONMAN. January. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1978) Requirements for High Order Programming Languages, STEELMAN. June. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1980a) Requirements for High Order Programming Languages, STONEMAN. February. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1980b) Requirements for the Programming Environment for the Common High Order Language, STONEMAN. U.S. Department of Defense, Washington, D.C.

DeRemer, F. (1971) “Simple LR(k) Grammars.” Commun. ACM, Vol. 14, No. 7, pp. 453–460.

DeRemer, F., and T. Pennello. (1982) “Efficient Computation of LALR(1) Look-Ahead Sets.” ACM TOPLAS, Vol. 4, No. 4, pp. 615–649.

Deutsch, L. P., and D. G. Bobrow. (1976) “An Efficient Incremental Automatic Garbage Collector.” Commun. ACM, Vol. 11, No. 3, pp. 522–526.

Dijkstra, E. W. (1968a) “Goto Statement Considered Harmful.” Commun. ACM, Vol. 11, No. 3, pp. 147–149.

Dijkstra, E. W. (1968b) “Cooperating Sequential Processes.” In Programming Languages, F. Genuys (ed.). Academic Press, New York, pp. 43–112.

Dijkstra, E. W. (1972) “The Humble Programmer.” Commun. ACM, Vol. 15, No. 10, pp. 859–866.

Dijkstra, E. W. (1975) “Guarded Commands, Nondeterminacy, and Formal Derivation of Programs.” Commun. ACM, Vol. 18, No. 8, pp. 453–457.

Dijkstra, E. W. (1976). A Discipline of Programming. Prentice-Hall, Englewood Cliffs, NJ.

Dybvig, R. K. (2009) The Scheme Programming Language, 4e. MIT Press, Boston.

Ellis, M. A., and B. Stroustrup. (1990) The Annotated C++ Reference Manual. Addison-Wesley, Reading, MA.

Farber, D. J., R. E. Griswold, and I. P. Polonsky. (1964) "SNOBOL, a String Manipulation Language." J. ACM, Vol. 11, No. 1, pp. 21–30.

Farrow, R. (1982) "LINGUIST 86: Yet Another Translator Writing System Based on Attribute Grammars." ACM SIGPLAN Notices, Vol. 17, No. 6, pp. 160–171.

Fischer, C. N., G. F. Johnson, J. Mauney, A. Pal, and D. L. Stock. (1984) "The Poe Language-Based Editor Project." ACM SIGPLAN Notices, Vol. 19, No. 5, pp. 21–29.

Fischer, C. N., and R. J. LeBlanc. (1977) UW-Pascal Reference Manual. Madison Academic Computing Center, Madison, WI.

Fischer, C. N., and R. J. LeBlanc. (1980) "Implementation of Runtime Diagnostics in Pascal." IEEE Transactions on Software Engineering, Vol. SE-6, No. 4, pp. 313–319.

Fischer, C. N., and R. J. LeBlanc. (1991) Crafting a Compiler in C. Benjamin/Cummings, Menlo Park, CA.

Flanagan, D. (2011) JavaScript: The Definitive Guide, 6e. O'Reilly Media, Sebastopol, CA.

Floyd, R. W. (1967) "Assigning Meanings to Programs." Proceedings Symposium Applied Mathematics. Mathematical Aspects of Computer Science, J. T. Schwartz (ed.). American Mathematical Society, Providence, RI.

Frege, G. (1892) "Über Sinn und Bedeutung." Zeitschrift für Philosophie und Philosophisches Kritik, Vol. 100, pp. 25–50.

Friedl, J. E. F. (2006) Mastering Regular Expressions, 3e. O'Reilly Media, Sebastopol, CA.

Friedman, D. P., and D. S. Wise. (1979) "Reference Counting's Ability to Collect Cycles Is Not Insurmountable." Information Processing Letters, Vol. 8, No. 1, pp. 41–45.

Fuchi, K. (1981) "Aiming for Knowledge Information Processing Systems." Proceedings of the International Conference on Fifth Generation Computing Systems. Japan Information Processing Development Center, Tokyo. Republished (1982) by North-Holland Publishing, Amsterdam.

Gehani, N. (1983) Ada: An Advanced Introduction. Prentice-Hall, Englewood Cliffs, NJ.

Gilman, L., and A. J. Rose. (1983) APL: An Interactive Approach, 3e. John Wiley, New York.

Goldberg, A., and D. Robson. (1983) Smalltalk-80: The Language and Its Implementation. Addison-Wesley, Reading, MA.

Goldberg, A., and D. Robson. (1989) Smalltalk-80: The Language. Addison-Wesley, Reading, MA.

Goodenough, J. B. (1975) "Exception Handling: Issues and Proposed Notation." Commun. ACM, Vol. 18, No. 12, pp. 683–696.

Goos, G., and J. Hartmanis (eds.) (1983) The Programming Language Ada Reference Manual. American National Standards Institute. ANSI/MIL-STD-1815A–1983. Lecture Notes in Computer Science 155. Springer-Verlag, New York.

Gordon, M. (1979) The Denotational Description of Programming Languages, An Introduction. Springer-Verlag, New York.

Graham, P. (1996) ANSI Common LISP. Prentice-Hall, Englewood Cliffs, NJ.

Gries, D. (1981) The Science of Programming. Springer-Verlag, New York.

Griswold, R. E., and M. T. Griswold. (1983) The ICON Programming Language. Prentice-Hall, Englewood Cliffs, NJ.

Griswold, R. E., F. Poage, and I. P. Polonsky. (1971) The SNOBOL 4 Programming Language, 2e. Prentice-Hall, Englewood Cliffs, NJ.

Halstead, R. H., Jr. (1985) "Multilisp: A Language for Concurrent Symbolic Computation." ACM Transactions on Programming Language and Systems, Vol. 7, No. 4, October 1985, pp. 501–538.

Halvorson, M. (2013) Microsoft Visual Basic 2013 Step by Step. Microsoft Press, Redmond, WA.

Hammond, P. (1983) APES: A User Manual. Department of Computing Report 82/9. Imperial College of Science and Technology, London.

Harbison, S. P. III, and G. L. Steele, Jr. (2002) A. C. Reference Manual, 5e. Prentice-Hall, Upper Saddle River, NJ.

Henderson, P. (1980) Functional Programming: Application and Implementation. Prentice-Hall, Englewood Cliffs, NJ.

Hoare, C. A. R. (1969) "An Axiomatic Basis of Computer Programming." Commun. ACM, Vol. 12, No. 10, pp. 576–580.

Hoare, C. A. R. (1972) "Proof of Correctness of Data Representations." Acta Informatica, Vol. 1, pp. 271–281.

Hoare, C. A. R. (1973) "Hints on Programming Language Design." Proceedings ACM SIGACT/SIGPLAN Conference on Principles of Programming Languages. Also published as Technical Report STAN-CS-73-403, Stanford University Computer Science Department.

Hoare, C. A. R. (1974) "Monitors: An Operating System Structuring Concept." Commun. ACM, Vol. 17, No. 10, pp. 549–557.

Hoare, C. A. R. (1978) "Communicating Sequential Processes." Commun. ACM, Vol. 21, No. 8, pp. 666–677.

Hoare, C. A. R. (1981) "The Emperor's Old Clothes." Commun. ACM, Vol. 24, No. 2, pp. 75–83.

Hoare, C. A. R., and N. Wirth. (1973) "An Axiomatic Definition of the Programming Language Pascal." Acta Informatica, Vol. 2, pp. 335–355.

Hogger, C. J. (1984) Introduction to Logic Programming. Academic Press, London.

Hogger, C. J. (1991) Essentials of Logic Programming. Oxford Science Publications, Oxford, England.

Holt, R. C., G. S. Graham, E. D. Lazowska, and M. A. Scott. (1978) Structured Concurrent Programming with Operating Systems Applications. Addison-Wesley, Reading, MA.

Horn, A. (1951) "On Sentences Which Are True of Direct Unions of Algebras." J. Symbolic Logic, Vol. 16, pp. 14–21.

Hudak, P., and J. Fasel. (1992) "A Gentle Introduction to Haskell." ACM SIGPLAN Notices, Vol. 27, No. 5, May 1992, pp. T1–T53.

Hughes, J. (1989) "Why Functional Programming Matters." The Computer Journal, Vol. 32, No. 2, pp. 98–107.

Huskey, H. K., R. Love, and N. Wirth. (1963) "A Syntactic Description of BC NELIAC." Commun. ACM, Vol. 6, No. 7, pp. 367–375.

IBM. (1954) "Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN." IBM Corporation, New York.

IBM. (1956) "Programmer's Reference Manual, The FORTRAN Automatic Coding System for the IBM 704 EDPM." IBM Corporation, New York.

IBM. (1964) The New Programming Language. IBM UK Laboratories, Hursley, England.

Ichbiah, J. D., J. C. Heliard, O. Roubine, J. G. P. Barnes, B. Krieg-Brueckner, and B. A. Wichmann. (1979) "Part B: Rationale for the Design of the Ada Programming Language." ACM SIGPLAN Notices, Vol. 14, No. 6.

IEEE. (1985) "Binary Floating-Point Arithmetic." IEEE Standard 754, IEEE, New York.

Ierusalimschy, R. (2006) Programming in Lua, 2e. Lua.org, Rio de Janeiro, Brazil.

INCITS/ISO/IEC. (1997) 1539-1-1997, Information Technology—Programming Languages—FORTRAN, Part 1: Base Language. American National Standards Institute, New York.

Ingerman, P. Z. (1967). "Panini-Backus Form Suggested." Commun. ACM, Vol. 10, No. 3, p. 137.

ISO. (1982) ISO7185:1982, Specification for Programming Language Pascal. International Organization for Standardization, Geneva, Switzerland.

ISO. (1998) ISO14882-1, ISO/IEC Standard – Information Technology—Programming Language—C++. International Organization for Standardization, Geneva, Switzerland.

ISO. (1999) ISO/IEC 9899:1999, Programming Language C. American National Standards Institute, New York.

ISO/IEC. (1996) 14977:1996, Information Technology—Syntactic Metalanguage—Extended BNF. International Organization for Standardization, Geneva, Switzerland.

ISO/IEC. (2002) 1989:2002, Information Technology—Programming Languages—COBOL. American National Standards Institute, New York.

ISO/IEC. (2010) 1539-1, Information Technology—Programming Languages—Fortran. American National Standards Institute, New York.

ISO/IEC. (2014) 8652/2012(E), Ada 2012 Reference Manual. Springer-Verlag, New York.

Iverson, K. E. (1962) A Programming Language. John Wiley, New York.

Jensen, K., and N. Wirth. (1974) Pascal Users Manual and Report. Springer-Verlag, Berlin.

Johnson, S. C. (1975) "Yacc—Yet Another Compiler Compiler." Computing Science Report 32. AT&T Bell Laboratories, Murray Hill, NJ.

Jones, N. D. (ed.) (1980) Semantic-Directed Compiler Generation. Lecture Notes in Computer Science, Vol. 94. Springer-Verlag, Heidelberg, FRG.

Kay, A. (1969) The Reactive Engine. PhD Thesis. University of Utah, September.

Kernighan, B. W., and D. M. Ritchie. (1978) The C Programming Language. Prentice-Hall, Englewood Cliffs, NJ.

Knuth, D. E. (1965) "On the Translation of Languages from Left to Right." Information & Control, Vol. 8, No. 6, pp. 607–639.

Knuth, D. E. (1967) "The Remaining Trouble Spots in ALGOL 60." Commun. ACM, Vol. 10, No. 10, pp. 611–618.

Knuth, D. E. (1968a) "Semantics of Context-Free Languages." Mathematical Systems Theory, Vol. 2, No. 2, pp. 127–146.

Knuth, D. E. (1968b) The Art of Computer Programming, Vol. I, 2e. Addison-Wesley, Reading, MA.

Knuth, D. E. (1974) "Structured Programming with GOTO Statements." ACM Computing Surveys, Vol. 6, No. 4, pp. 261–301.

Knuth, D. E. (1981) The Art of Computer Programming, Vol. II, 2e. Addison-Wesley, Reading, MA.

Knuth, D. E., and L. T. Pardo. (1977) "Early Development of Programming Languages." In Encyclopedia of Computer Science and Technology, G. Holzman and A. Kent (eds.). Vol. 7. Dekker, New York, pp. 419–493.

Kochan, S. G. (2009) Programming in Objective-C 2.0. Addison-Wesley, Upper Saddle River, NJ.

Kowalski, R. A. (1979) Logic for Problem Solving. Artificial Intelligence Series, Vol. 7. Elsevier-North Holland, New York.

Laning, J. H., Jr., and N. Zierler. (1954) "A Program for Translation of Mathematical Equations for Whirlwind I." Engineering memorandum E-364. Instrumentation Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Ledgard, H. (1984) The American Pascal Standard. Springer-Verlag, New York.

Ledgard, H. F., and M. Marcotty. (1975) "A Genealogy of Control Structures." Commun. ACM, Vol. 18, No. 11, pp. 629–639.

Lischner, R. (2000) Delphi in a Nutshell. O'Reilly Media, Sebastopol, CA.

Liskov, B., R. L. Atkinson, T. Bloom, J. E. B. Moss, C. Scheffert, R. Scheifler, and A. Snyder. (1981) CLU Reference Manual. Springer, New York.

Liskov, B., and A. Snyder. (1979) "Exception Handling in CLU." IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, pp. 546–558.

Lomet, D. (1975) "Scheme for Invalidating References to Freed Storage." IBM Journal of Research and Development, Vol. 19, pp. 26–35.

Lutz, M. (2013) Learning Python, 5e. O'Reilly Media, Sebastopol, CA.

MacLaren, M. D. (1977) "Exception Handling in PL/I." ACM SIGPLAN Notices, Vol. 12, No. 3, pp. 101–104.

Marcotty, M., H. F. Ledgard, and G. V. Bochmann. (1976) "A Sampler of Formal Definitions." ACM Computing Surveys, Vol. 8, No. 2, pp. 191–276.

Mather, D. G., and S. V. Waite (eds.) (1971) BASIC, 6e. University Press of New England, Hanover, NH.

McCarthy, J. (1960) "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I." Commun. ACM, Vol. 3, No. 4, pp. 184–195.

McCarthy, J., P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. Levin. (1965) LISP 1.5 Programmer's Manual, 2e. MIT Press, Cambridge, MA.

McCracken, D. (1961) Guide to FORTRAN Programming. John Wiley & Sons, Inc., New York.

McCracken, D. (1970) "Whither APL." Datamation, September 15, pp. 53–57.

Metcalf, M., J. Reid, and M. Cohen. (2004) Fortran 95/2003 Explained, 3e. Oxford University Press, Oxford, England.

Meyer, B. (1990) Introduction to the Theory of Programming Languages. Prentice-Hall, Englewood Cliffs, NJ.

Meyer, B. (1992) Eiffel: The Language. Prentice-Hall, Englewood Cliffs, NJ.

Milner, R., R. Harper, and M. Tofle. (1997) The Definition of Standard ML-Revised. MIT Press, Cambridge, MA.

Milos, D., U. Pleban, and G. Loegel. (1984) "Direct Implementation of Compiler Specifications." POPL '84 Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Programming Languages, pp. 196–202.

Mitchell, J. G., W. Maybury, and R. Sweet. (1979) Mesa Language Manual, Version 5.0, CSL-79-3. Xerox Research Center, Palo Alto, CA.

Moss, C. (1994) Prolog++: The Power of Object-Oriented and Logic Programming. Addison-Wesley, Reading, MA.

Moto-oka, T. (1981) "Challenge for Knowledge Information Processing Systems." Proceedings of the International Conference on Fifth Generation Computing Systems. Japan Information Processing Development Center, Tokyo. Republished (1982) by North-Holland Publishing, Amsterdam.

Naur, P. (ed.) (1960) "Report on the Algorithmic Language ALGOL 60." Commun. ACM, Vol. 3, No. 5, pp. 299–314.

Newell, A., and H. A. Simon. (1956) "The Logic Theory Machine—A Complex Information Processing System." IRE Transactions on Information Theory, Vol. IT-2, No. 3, pp. 61–79.

Newell, A., and F. M. Tonge. (1960) "An Introduction to Information Processing Language V." Commun. ACM, Vol. 3, No. 4, pp. 205–211.

Nilsson, N. J. (1971) Problem Solving Methods in Artificial Intelligence. McGraw-Hill, New York.

Ousterhout, J. K. (1994) Tcl and the Tk Toolkit. Addison-Wesley, Reading, MA.

Paepcke, E. (ed.) (1993) Object-Oriented Programming: The CLOS Perspective. MIT Press, Cambridge, MA.

Pagan, F. G. (1981) Formal Specifications of Programming Languages. Prentice-Hall, Englewood Cliffs, NJ.

Papert, S. (1980) MindStorms: Children, Computers and Powerful Ideas. Basic Books, New York.

Perlis, A., and K. Samelson. (1958) "Preliminary Report—International Algebraic Language." Commun. ACM, Vol. 1, No. 12, pp. 8–22.

Peyton Jones, S. L. (1987) The Implementation of Functional Programming Languages. Prentice-Hall, Englewood Cliffs, NJ.

Pratt, T. W. (1984) Programming Languages: Design and Implementation, 2e. Prentice-Hall, Englewood Cliffs, NJ.

Pratt, T. W., and M. V. Zelkowitz. (2001) Programming Languages: Design and Implementation, 4e. Prentice-Hall, Englewood Cliffs, NJ.

Remington-Rand. (1952) "UNIVAC Short Code." Unpublished collection of dittoed notes. Preface by A. B. Tonik, dated October 25, 1955 (1 p.); Preface by J. R. Logan, undated but apparently from 1952 (1 p.); Preliminary exposition, 1952? (22 pp., where in which pp. 20–22 appear to be a later replacement); Short code supplementary information, topic one (7 pp.); Addenda #1, 2, 3, 4 (9 pp.).

Reppy, J. H. (1999) Concurrent Programming in ML. Cambridge University Press, New York.

Richards, M. (1969) "BCPL: A Tool for Compiler Writing and Systems Programming." Proc. AFIPS SJCC, Vol. 34, pp. 557–566.

Robbins, A. (2005) Unix in a Nutshell, 4e. O'Reilly Media, Sebastopol, CA.

Robinson, J. A. (1965) "A Machine-Oriented Logic Based on the Resolution Principle." Journal of the ACM, Vol. 12, pp. 23–41.

Roussel, P. (1975) "PROLOG: Manual de Reference et D'utilisation." Research Report. Artificial Intelligence Group, University of Aix-Marseille, Luming, France.

Rubin, F. (1987) "'GOTO Statement Considered Harmful' considered harmful" (letter to editor). Commun. ACM, Vol. 30, No. 3, pp. 195–196.

Rutishauser, H. (1967) Description of ALGOL 60. Springer-Verlag, New York.

Sammet, J. E. (1969) Programming Languages: History and Fundamentals. Prentice-Hall, Englewood Cliffs, NJ.

Sammet, J. E. (1976) "Roster of Programming Languages for 1974–75." Commun. ACM, Vol. 19, No. 12, pp. 655–669.

Schneider, D. I. (1999) An Introduction to Programming Using Visual BASIC 6.0. Prentice-Hall, Englewood Cliffs, NJ.

Schorr, H., and W. Waite. (1967) "An Efficient Machine Independent Procedure for Garbage Collection in Various List Structures." Commun. ACM, Vol. 10, No. 8, pp. 501–506.

Scott, D. S., and C. Strachey. (1971) "Towards a Mathematical Semantics for Computer Language." In Proceedings, Symposium on Computers and Automation, J. Fox (ed.). Polytechnic Institute of Brooklyn Press, New York, pp. 19–46.

Scott, M. (2009) Programming Language Pragmatics, 3e. Morgan Kaufman, San Francisco, CA.

Sebesta, R. W. (1991) VAX Structured Assembly Language Programming, 2e. Benjamin/Cummings, Redwood City, CA.

Sergot, M. J. (1983) "A Query-the-User Facility for Logic Programming." In Integrated Interactive Computer Systems, P. Degano and E. Sandewall (eds.). North-Holland Publishing, Amsterdam.

Shaw, C. J. (1963) "A Specification of JOVIAL." Commun. ACM, Vol. 6, No. 12, pp. 721–736.

Smith, J. B. (2006) Practical OCaml. Apress, Springer-Verlag, New York.

Sommerville, I. (2010) Software Engineering, 9e. Addison-Wesley, Reading, MA.

Steele, G. L., Jr. (1990) Common LISP The Language, 2e. Digital Press, Burlington, MA.

Stoy, J. E. (1977) Denotational Semantics: The Scott–Strachey Approach to Programming Language Semantics. MIT Press, Cambridge, MA.

Stroustrup, B. (1983) "Adding Classes to C: An Exercise in Language Evolution." Software—Practice and Experience, Vol. 13, pp. 139–161.

Stroustrup, B. (1984) "Data Abstraction in C." AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, pp. 1701–1732.

Stroustrup, B. (1986) The C++ Programming Language. Addison-Wesley, Reading, MA.

Stroustrup, B. (1988) "What Is Object-Oriented Programming?" IEEE Software, May 1988, pp. 10–20.

Stroustrup, B. (1991) The C++ Programming Language, 2e. Addison-Wesley, Reading, MA.

Stroustrup, B. (1994) The Design and Evolution of C++. Addison-Wesley, Reading, MA.

Stroustrup, B. (1997) The C++ Programming Language, 3e. Addison-Wesley, Reading, MA.

Sussman, G. J., and G. L. Steele, Jr. (1975) "Scheme: An Interpreter for Extended Lambda Calculus." MIT AI Memo No. 349 (December 1975).

Suzuki, N. (1982) "Analysis of Pointer 'Rotation'." Commun. ACM, Vol. 25, No. 5, pp. 330–335.

Syme, D., A. Granicz, and A. Cisternino. (2010) Expert F# 2.0. Apress, Springer-Verlag, New York.

Tatroe, K., P. MacIntyre, and R. Lerdorf. (2013) Programming PHP, 3e. O'Reilly Media, Sebastopol, CA.

Tanenbaum, A. S. (2005) Structured Computer Organization, 5e. Prentice-Hall, Englewood Cliffs, NJ.

Teitelbaum, T., and T. Reps. (1981) "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment." Commun. ACM, Vol. 24, No. 9, pp. 563–573.

Teitelman, W. (1975) INTERLISP Reference Manual. Xerox Palo Alto Research Center, Palo Alto, CA.

Tenenbaum, A. M., Y. Langsam, and M. J. Augenstein. (1990) Data Structures Using C. Prentice-Hall, Englewood Cliffs, NJ.

Thomas, D., A. Hunt, and C. Fowler. (2013) Programming Ruby 1.9 & 2.0: The Pragmatic Programmers Guide (The Facets of Ruby). The Pragmatic Bookshelf, Raleigh, NC.

Thompson, S. (1999) Haskell: The Craft of Functional Programming, 2e. Addison-Wesley, Reading, MA.

Turner, D. (1986) "An Overview of Miranda." ACM SIGPLAN Notices, Vol. 21, No. 12, pp. 158–166.

Ullman, J. D. (1998) Elements of ML Programming. ML97 Edition. Prentice-Hall, Englewood Cliffs, NJ.

van Emden, M. H. (1980) "McDermott on Prolog: A Rejoinder." SIGART Newsletter, No. 72, August, pp. 19–20.

van Wijngaarden, A., B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster. (1969) "Report on the Algorithmic Language ALGOL 68." Numerische Mathematik, Vol. 14, No. 2, pp. 79–218.

Wadler, P. (1998) "Why No One Uses Functional Languages." ACM SIGPLAN Notices, Vol. 33, No. 2, February 1998, pp. 25–30.

Warren, D. H. D., L. M. Pereira, and F. C. N. Pereira. (1979) "User's Guide to DEC System-10 Prolog." Occasional Paper 15. Department of Artificial Intelligence, University of Edinburgh, Scotland.

Watt, D. A. (1979) "An Extended Attribute Grammar for Pascal." ACM SIGPLAN Notices, Vol. 14, No. 2, pp. 60–74.

Wegner, P. (1972) "The Vienna Definition Language." ACM Computing Surveys, Vol. 4, No. 1, pp. 5–63.

Weissman, C. (1967) LISP 1.5 Primer. Dickenson Press, Belmont, CA.

Wexelblat, R. L. (ed.) (1981) History of Programming Languages. Academic Press, New York.

Wheeler, D. J. (1950) "Programme Organization and Initial Orders for the EDSAC." Proc. R. Soc. London, Ser. A, Vol. 202, pp. 573–589.

Wilkes, M. V. (1952) "Pure and Applied Programming." In Proceedings of the ACM National Conference, Vol. 2. Toronto, pp. 121–124.

Wilkes, M. V., D. J. Wheeler, and S. Gill. (1951) The Preparation of Programs for an Electronic Digital Computer, with Special Reference to the EDSAC and the Use of a Library of Subroutines. Addison-Wesley, Reading, MA.

Wilkes, M. V., D. J. Wheeler, and S. Gill. (1957) The Preparation of Programs for an Electronic Digital Computer, 2e. Addison-Wesley, Reading, MA.

Wilson, P. R. (2005) "Uniprocessor Garbage Collection Techniques." Available at http://www.cs.utexas.edu/users/oops/papers.htm#bigsurv.

Wirth, N. (1971) "The Programming Language Pascal." Acta Informatica, Vol. 1, No. 1, pp. 35–63.

Wirth, N. (1973) Systematic Programming: An Introduction. Prentice-Hall, Englewood Cliffs, NJ.

Wirth, N. (1975) "On the Design of Programming Languages." Information Processing 74 (Proceedings of IFIP Congress 74). North Holland, Amsterdam, pp. 386–393.

Wirth, N. (1977) "Modula: A Language for Modular Multi-Programming." Software—Practice and Experience, Vol. 7, pp. 3–35.

Wirth, N. (1985) Programming in Modula-2, 3e. Springer-Verlag, New York.

Wirth, N. (1988) "The Programming Language Oberon." Software—Practice and Experience, Vol. 18, No. 7, pp. 671–690.

Wirth, N., and C. A. R. Hoare. (1966) "A Contribution to the Development of ALGOL." Commun. ACM, Vol. 9, No. 6, pp. 413–431.

Wulf, W. A., D. B. Russell, and A. N. Habermann. (1971) "BLISS: A Language for Systems Programming." Commun. ACM, Vol. 14, No. 12, pp. 780–790.

Zuse, K. (1972) "Der Plankalkül." Manuscript prepared in 1945, published in Berichte der Gesellschaft für Mathematik und Datenverarbeitung, No. 63 (Bonn, 1972); Part 3, 285 pp. English translation of all but pp. 176–196 in No. 106 (Bonn, 1976), pp. 42–244.

Conheça também

AGUILAR, L.
Fundamentos de Programação, 3.ed.
Programação em C++, 2.ed.

BROOKSHEAR, J.G.
Ciência da Computação, uma Visão Abrangente, 11.ed.

DEITEL, P.; DEITEL, H.
Android 6 para Programadores: Uma Abordagem Baseada em Aplicativos, 3.ed.

FLANAGAN, D.
JavaScript – O Guia Definitivo, 6.ed.

GOODRICH, M.T.; TAMASSIA, R.
Estruturas de Dados e Algoritmos em Java, 5.ed.

HORTMANN, C.
Conceitos de Computação com o Essencial de C ++, 3.ed.
Conceitos de Computação com o Essencial de Java, 3.ed.
Conceitos de Computação com Java, 5.ed.

KOCHAN, S.
Programação com Objective-C, 5.ed.

LIPPMAN, S.B.
C # – Um guia prático

MEYERS, S.
C + + Eficaz, 3.ed.

PINHEIRO, F.A.C.
Elementos de Programação em C

SCHILDT, H.
Java para iniciantes, 6.ed.
Programação com Java

STROUSTRUP, B.
Princípios e Práticas de Programação com C ++

TUCKER, A.B.; NOONAN, R.E.
Linguagens de Programação – Princípios e Paradigmas, 2.ed.

Sobre o Grupo A

O Grupo A está preparado para ajudar pessoas e instituições a encontrarem respostas para os desafios da educação. Estudantes, professores, médicos, engenheiros, psicólogos. Profissionais das carreiras que ainda não têm nome. Universidades, escolas, hospitais e empresas das mais diferentes áreas.
O Grupo A está ao lado de cada um. E também está nas suas mãos. Nos seus conteúdos virtuais.
E no lugar mais importante: nas suas mentes.

Acesse

0800 703 3444
sac@grupoa.com.br
Av. Jerônimo de Ornelas, 670
Santana
CEP: 90040-340 · Porto Alegre / RS