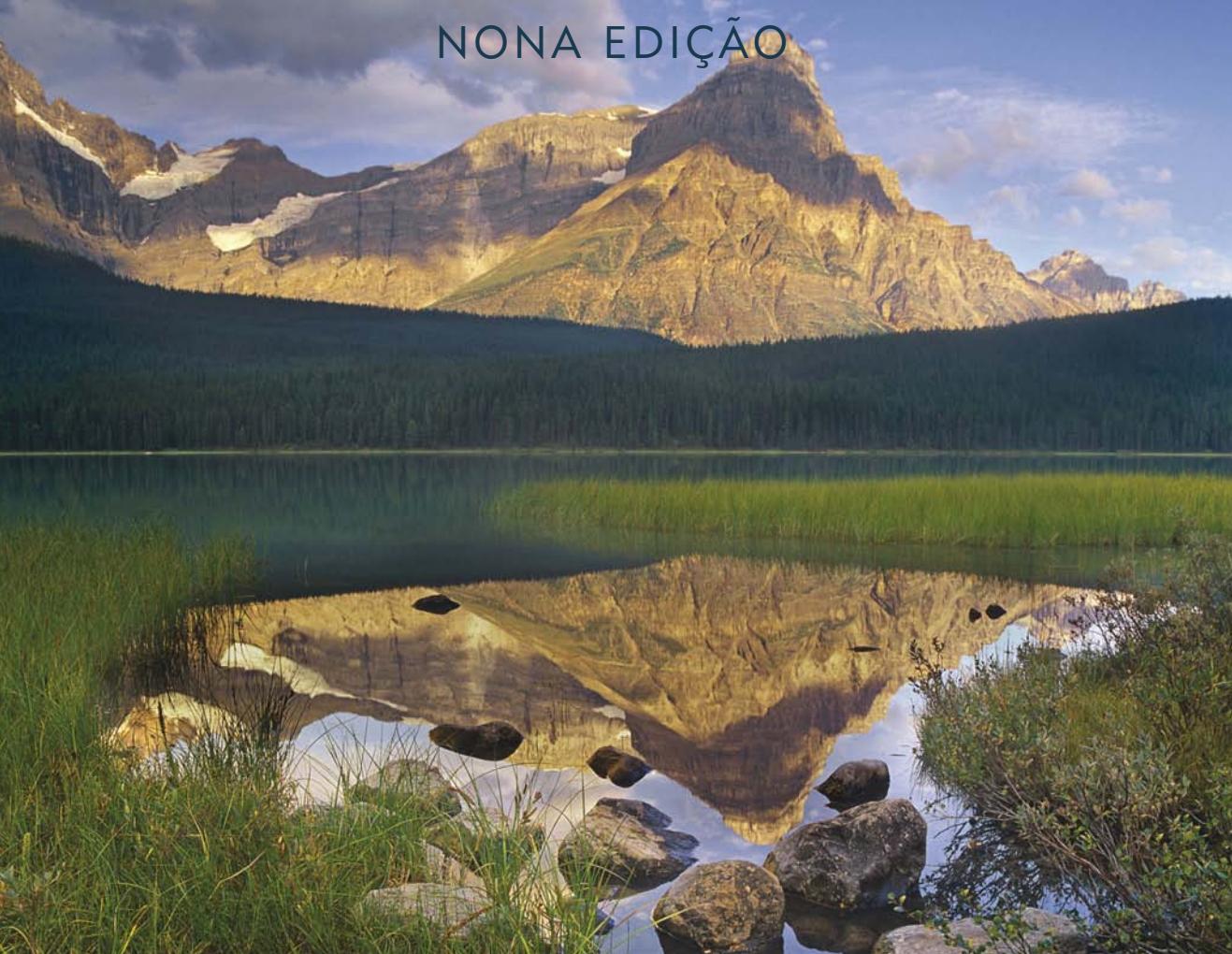


CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO

NONA EDIÇÃO



ROBERT W. SEBESTA



ROBERT W. SEBESTA

CONCEITOS DE
**LINGUAGENS DE
PROGRAMAÇÃO**

NONA EDIÇÃO

Tradução técnica:

Eduardo Kessler Piveta
Doutor em Ciência da Computação – UFRGS
Professor Adjunto da Universidade Federal de Santa Maria – UFSM

Versão impressa
desta obra: 2011



2011

Obra originalmente publicada sob o título
Concepts of Programming Languages, 9th Edition

ISBN 9780136073475

Authorized translation from the English language edition, entitled CONCEPTS OF PROGRAMMING LANGUAGES, 9th Edition, by ROBERT SEBESTA, published by Pearson Education, Inc., publishing as Addison-Wesley, Copyright © 2010. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Portuguese language edition published by Bookman Companhia Editora Ltda, a Division of Artmed Editora SA, Copyright © 2010

Tradução autorizada a partir do original em língua inglesa da obra intitulada CONCEPTS OF PROGRAMMING LANGUAGES, 9^a Edição, autoria de ROBERT SEBESTA, publicado por Pearson Education, Inc., sob o selo Addison-Wesley, Copyright © 2010. Todos os direitos reservados. Este livro não poderá ser reproduzido nem em parte nem na íntegra, nem ter partes ou sua íntegra armazenado em qualquer meio, seja mecânico ou eletrônico, inclusive fotoreprogração, sem permissão da Pearson Education, Inc.

A edição em língua portuguesa desta obra é publicada por Bookman Companhia Editora Ltda, uma divisão da Artmed Editora SA, Copyright © 2010

Capa: *Rogério Grillo*, arte sobre capa original

Leitura final: *Mirella Nascimento*

Editora Sênior: *Denise Weber Nowaczyk*

Editora responsável por esta obra: *Elisa Viali*

Projeto e editoração: *Techbooks*

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

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

SÃO PAULO
Av. Embaixador Macedo Soares, 10.735 - Pavilhão 5 - Cond. Espace Center
Vila Anastácio 05095-035 São Paulo SP
Fone (11) 3665-1100 Fax (11) 3667-1333

SAC 0800 703-3444

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

O AUTOR

Robert Sebesta é Professor Associado Emérito no Departamento de Ciência da Computação da Universidade do Colorado – Colorado Springs. Tem bacharelado em Matemática Aplicada pela Universidade de Colorado em Boulder e mestrado e doutorado (PhD) em Ciência da Computação pela Universidade Estadual da Pensilvânia. Ele ensina ciência da computação há mais de 38 anos. Seus interesses profissionais são o projeto e a avaliação de linguagens de programação, projeto de compiladores e métodos e ferramentas de testes de software.



S443c Sebesta, Robert W.

Conceitos de linguagens de programação [recurso eletrônico] /
Robert W. Sebesta; tradução técnica: Eduardo Kessler Piveta. –
9. ed. – Dados eletrônicos. – Porto Alegre : Bookman, 2011.

Editado também como livro impresso em 2011.
ISBN 978-85-7780-862-5

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

CDU 004.43

Prefácio

MUDANÇAS PARA A NONA EDIÇÃO

Os objetivos, a estrutura geral e a abordagem desta nona edição de *Conceitos de Linguagens de Programação* permanecem as mesmas das oito anteriores. Os objetivos principais são introduzir as construções mais importantes das linguagens de programação contemporâneas e fornecer ao leitor as ferramentas necessárias para a avaliação crítica de linguagens existentes e futuras. Um objetivo adicional é preparar o leitor para o estudo de projeto de compiladores, apresentando um método formal de descrição de sintaxe e introduzindo abordagens para as análises sintática e léxica.

Para manter a atualidade do conteúdo, algumas das discussões acerca de linguagens de programação mais antigas foram substituídas por material de linguagens mais recentes na nona edição. Por exemplo, no Capítulo 6, foram adicionadas discussões acerca dos tipos tabela em Lua, das tuplas e das compreensões de lista de Python, e das fatias em Ruby. Algum material mais antigo foi removido do Capítulo 7, diminuindo-o em duas páginas. A seção sobre o modelo de objetos de JavaScript foi removida do Capítulo 11. Em alguns casos, o material foi movido. Por exemplo, três seções do Capítulo 5 foram para o Capítulo 6, cujo tópico trata de tipos de dados. Em algumas seções, o material foi expandido. No Capítulo 4, por exemplo, o código do analisador léxico foi expandido para um programa completo, e uma saída real dele foi incluída. A seção sobre escopo no Capítulo 5 foi expandida com subseções sobre escopo global e ordem de declaração. A discussão dos tipos de dados abstratos no Capítulo 11 foi revisada e aumentada. No Capítulo 15, uma seção sobre recursão em cauda em Scheme foi adicionada. No Capítulo 3, a introdução à semântica foi revisada e as subseções foram reordenadas. Além disso, pequenas mudanças foram feitas em diversas seções do livro, principalmente para melhorar a clareza. Por fim, mais de 50 problemas e exercícios de programação, além de mais de cem questões de revisão, foram adicionados à obra.

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, dentre os quais estão os métodos de descrição de sintaxe e de semântica, tratados no Capítulo 3. Além disso, técnicas de implementação para várias construções de linguagem devem ser consideradas: as análises léxica e sintática são discutidas no Capítulo 4, e a implementação da ligação de subprogramas é assunto do Capítulo 10. A implementação de outras construções de linguagem é discutida em outras partes do livro.

Os parágrafos a seguir descrevem o conteúdo da nona edição.

Descrição dos Capítulos

O Capítulo 1 começa com uma discussão sobre por que estudar linguagens de programação. Ele então discute os critérios usados para avaliar linguagens e construções de linguagem. A influência primária no projeto de linguagens, alguns compromissos comuns de projeto e as abordagens básicas para a implementação também são examinadas.

O Capítulo 2 descreve a evolução da maioria das linguagens importantes discutidas neste livro. Apesar de nenhuma delas ser descrita completamente, as origens, os propósitos e as contribuições de cada uma são discutidos. Essa visão histórica é valiosa, porque fornece o conhecimento necessário para entender as bases teóricas e práticas do projeto contemporâneo de linguagens. Também motiva o estudo adicional do projeto e da avaliação de linguagens. E, já que nenhum dos capítulos restantes depende do Capítulo 2, ele pode ser lido independentemente.

O Capítulo 3 mostra o modelo formal primário para descrever a sintaxe de linguagens de programação – BNF. Esse é seguido por uma apresentação das 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 as faculdades que não mais requerem um curso de projeto de compiladores em seu currículo. Como o Capítulo 2, pode ser lido independentemente do restante do livro.

Os Capítulos 5 a 14 descrevem em detalhes as questões de projeto para as construções primárias das linguagens imperativas. Em cada um dos casos, as escolhas de projeto para diversas linguagens de exemplo são apresentadas e avaliadas. Especificamente, o Capítulo 5 cobre 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 fala sobre o tratamento de exceções, com uma breve discussão sobre o tratamento de eventos.

Os dois últimos capítulos (15 e 16) descrevem dois dos paradigmas de programação mais importantes: a programação funcional e a programação lógica. 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 em Scheme. Introduções breves a ML e a Haskell são dadas para ilustrar alguns tipos diferentes de linguagens funcionais. O Capítulo 16 introduz a programação lógica e a linguagem Prolog.

PARA O INSTRUTOR

No curso inicial de linguagens de programação na Universidade do Colorado, em Colorado Springs, o livro é usado da seguinte forma: cobrimos os Capítulos 1 e 3 em detalhes, e – apesar de os estudantes acharem interessante e de leitura benéfica – o Capítulo 2 recebe pouco tempo de aula devido à falta de conteúdo altamente técnico. Como nenhum material nos capítulos subsequentes depende do Capítulo 2, ele pode ser ignorado completamente. E, já que requeremos um curso de projeto de compiladores, o Capítulo 4 não é tratado.

Os Capítulos 5 a 9 devem ser relativamente fáceis para estudantes com extensa experiência em programação em C++, Java ou C#. Os Capítulos 10 a 14 são mais desafiadores e requerem aulas mais detalhadas.

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

Cursos de graduação provavelmente não serão capazes de cobrir todo o conteúdo dos últimos dois capítulos. Cursos de pós-graduação, entretanto, devem conseguir discutir completamente o material pulando partes dos primeiros capítulos acerca de linguagens imperativas.

MATERIAIS SUPLEMENTARES

Os seguintes suplementos estão disponíveis para os leitores deste livro em www.bookman.com.br.

- Um conjunto de slides (em português) com notas de aulas. Slides em *Power Point* estão disponíveis para cada um dos capítulos do livro.
- Slides em *Power Point* contendo todas as figuras do livro (em português).

Para acessar os recursos, procure pelo livro no site www.bookman.com.br e clique em **Conteúdo Online**.

Soluções para muitos dos conjuntos de problemas (em inglês) estão disponíveis para instrutores qualificados na nossa Área do Professor. Para acessar os materiais procure pelo livro em www.bookman.com.br e clique em **Material para professores**. Faça o seu cadastro de professor no site, caso ainda não tenha, para receber uma senha de acesso ao material.

Disponibilidade de processador de 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#	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 inclusa em praticamente todos os navegadores; PHP está incluso em praticamente todos os servidores Web.

Todas essas informações também estão incluídas no site do livro, em www.aw-bc.com/sebesta.

AGRADECIMENTOS

As sugestões de revisores magníficos contribuíram muito para a forma atual deste livro. Em ordem alfabética de sobrenomes, são eles:

I-ping Chu	<i>Universidade DePaul</i>
Amer Diwan	<i>Universidade do Colorado</i>
Stephen Edwards	<i>Virginia Tech</i>
Nigel Gwee	<i>Southern University – Baton Rouge</i>
K. N. King	<i>Universidade Estadual da Geórgia</i>
Donald Kraft	<i>Universidade Estadual da Louisiana</i>
Simon H. Lin	<i>Universidade Estadual da Califórnia – Northridge</i>
Mark Llewellyn	<i>Universidade da Flórida Central</i>
Bruce R. Maxim	<i>Universidade de Michigan – Dearborn</i>
Gloria Melara	<i>Universidade Estadual da Califórnia – Northridge</i>
Frank J. Mitropoulos	<i>Nova Universidade de Southeastern</i>
Eurípides Montagne	<i>Universidade da Flórida Central</i>
Bob Neufeld	<i>Universidade Estadual de Wichita</i>
Amar Raheja	<i>Universidade Politécnica do Estado da Califórnia – Pomona</i>
Hossein Saiedian	<i>Universidade de Kansas</i>
Neelam Soundarajan	<i>Universidade Estadual de Ohio</i>

Paul Tymann *Instituto de Tecnologia de Rochester*
Cristian Videira Lopes *Universidade da Califórnia – Irvine*
Salih Yurttas *Universidade Texas A&M*

Diversas outras pessoas forneceram sugestões para as edições prévias 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 elas: 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, Raghvinder Sangwan, Steven Rapkin, Hamilton Richard, Tom Sager, 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; Sarah Milmore, assistente editorial; e Meredith Gertz, supervisora de produção da Addison-Wesley, e Dennis Free da Aptara, merecem minha gratidão por seus esforços para produzir a nona edição rápida e cuidadosamente.

Sumário

Capítulo 1 Aspectos preliminares	19
1.1 Razões para estudar conceitos de linguagens de programação.....	20
1.2 Domínios de programação.....	23
1.3 Critérios de avaliação de linguagens.....	26
1.4 Influências no projeto de linguagens.....	38
1.5 Categorias de linguagens.....	42
1.6 <i>Trade-offs</i> no projeto de linguagens.....	43
1.7 Métodos de implementação.....	44
1.8 Ambientes de programação	52
Resumo • Questões de revisão • Conjunto de problemas	52
Capítulo 2 Evolução das principais linguagens de programação	56
2.1 Plankalkül de Zuse.....	58
2.2 Programação de hardware mínima: pseudocódigos	60
2.3 O IBM 704 e Fortran	63
2.4 Programação funcional: LISP	68
2.5 O primeiro passo em direção à sofisticação: ALGOL 60	74
2.6 Informatizando os registros comerciais: COBOL	80
2.7 O início do compartilhamento de tempo: BASIC.....	85
Entrevista: ALAN COOPER – Projeto de usuário e projeto de linguagens ...	88
2.8 Tudo para todos: PL/I	90
2.9 Duas das primeiras linguagens dinâmicas: APL e SNOBOL	93
2.10 O início da abstração de dados: SIMULA 67	95
2.11 Projeto ortogonal: ALGOL 68	96
2.12 Alguns dos primeiros descendentes dos ALGOLs.....	97
2.13 Programação baseada em lógica: Prolog.....	102
2.14 O maior esforço de projeto da história: Ada	103

2.15	Programação orientada a objetos: Smalltalk.....	108
2.16	Combinando recursos imperativos e orientados a objetos: C++	110
2.17	Uma linguagem orientada a objetos baseada no paradigma imperativo: Java	114
2.18	Linguagens de <i>scripting</i>	118
2.19	Uma linguagem baseada em C para o novo milênio: C#	125
2.20	Linguagens híbridas de marcação/programação	128
	Resumo • Notas bibliográficas • Questões de revisão • Conjunto de problemas • Exercícios de programação.....	130
Capítulo 3	Descrevendo sintaxe e semântica	135
3.1	Introdução.....	136
3.2	O problema geral de descrever sintaxe	137
3.3	Métodos formais para descrever sintaxe	139
3.4	Gramáticas de atributos	155
	Nota histórica.....	155
3.5	Descrevendo o significado de programas: semântica dinâmica.....	161
	Nota Histórica	176
	Resumo • Notas bibliográficas • Questões de revisão • Conjunto de problemas	183
Capítulo 4	Análise léxica e sintática	189
4.1	Introdução.....	190
4.2	Análise léxica	192
4.3	O problema da análise sintática	200
4.4	Análise sintática descendente recursiva	204
4.5	Análise sintática ascendente.....	213
	Resumo • Questões de revisão • Conjunto de problemas • Exercícios de programação	221
Capítulo 5	Nomes, vinculações e escopos	226
5.1	Introdução.....	227
5.2	Nomes.....	228
	Nota histórica.....	228
	Nota histórica.....	229

5.3	Variáveis.....	231
	Nota histórica.....	232
5.4	O conceito de vinculação.....	233
	Entrevista: RASMUS LERDORF – Linguagens de scripting e outros exemplos de soluções simples	238
5.5	Escopo	245
	Nota histórica.....	247
5.6	Escopo e tempo de vida	255
5.7	Ambientes de referenciamento	256
5.8	Constantes nomeadas	258
	Resumo • Questões de revisão • Conjunto de problemas • Exercícios de programação	260
Capítulo 6	Tipos de dados	267
6.1	Introdução.....	268
6.2	Tipos de dados primitivos.....	270
6.3	Cadeias de caracteres.....	274
	Nota histórica.....	275
6.4	Tipos ordinais definidos pelo usuário.....	280
6.5	Tipos de matrizes.....	284
	Nota histórica.....	285
	Nota histórica.....	288
6.6	Matrizes associativas	299
6.7	Registros	301
	Entrevista: ROBERTO IERUSALIMSCHY – Lua	302
6.8	Unões	308
6.9	Ponteiros e referências.....	312
	Nota histórica.....	317
6.10	Verificação de tipos.....	327
6.11	Tipagem forte.....	328
6.12	Equivalência de tipos.....	330
6.13	Teoria e tipos de dados	334
	Resumo • Notas bibliográficas • Questões de revisão • Conjunto de problemas • Exercícios de programação.....	336

Capítulo 7 Expressões e sentenças de atribuição	341
7.1 Introdução.....	342
7.2 Expressões aritméticas	343
7.3 Operadores sobrecarregados	352
7.4 Conversões de tipos.....	354
Nota histórica.....	356
7.5 Expressões relacionais e booleanas	357
Nota histórica.....	357
7.6 Avaliação em curto-circuito	360
7.7 Sentenças de atribuição	361
Nota histórica.....	365
7.8 Atribuição de modo misto.....	366
Resumo • Questões de revisão • Conjunto de problemas •	
Exercícios de programação	366
Capítulo 8 Estruturas de controle no nível de sentença	371
8.1 Introdução.....	372
8.2 Sentenças de seleção	374
Nota histórica.....	374
Nota histórica.....	376
8.3 Sentenças de iteração	387
Nota histórica.....	387
Entrevista: LARRY WALL – Parte 1: Linguística e o nascimento de Perl ..	392
Nota histórica.....	400
8.4 Desvio incondicional.....	403
8.5 Comandos protegidos.....	404
8.6 Conclusões.....	406
Resumo • Questões de revisão • Conjunto de problemas •	
Exercícios de programação	407
Capítulo 9 Subprogramas	413
9.1 Introdução.....	414
9.2 Fundamentos de subprogramas	414

9.3	Questões de projeto para subprogramas	425
9.4	Ambientes de referenciamento local.....	425
9.5	Métodos de passagem de parâmetros	428
Entrevista: LARRY WALL – Parte 2: Linguagens de <i>scripting</i> em geral e Perl em particular	430	
Nota histórica.....	438	
Nota histórica.....	438	
Nota histórica.....	442	
9.6	Parâmetros que são subprogramas.....	450
Nota histórica.....	452	
9.7	Subprogramas sobrecarregados	452
9.8	Subprogramas genéricos	453
9.9	Questões de projeto para funções	460
9.10	Operadores sobrecarregados definidos pelo usuário	462
9.11	Corrotinas.....	462
Nota histórica.....	462	
Resumo • Questões de revisão • Conjunto de problemas •		
Exercícios de programação	465	
Capítulo 10 Implementando subprogramas		471
10.1	A semântica geral de chamadas e retornos.....	472
10.2	Implementando subprogramas “simples”	473
10.3	Implementando subprogramas com variáveis locais dinâmicas da pilha	475
10.4	Subprogramas aninhados.....	484
Entrevista: NIKLAUS WIRTH – Mantendo a simplicidade	486	
10.5	Blocos.....	492
10.6	Implementando escopo dinâmico.....	494
Resumo • Questões de revisão • Conjunto de problemas •		
Exercícios de programação	498	
Capítulo 11 Tipos de dados abstratos e construções de encapsulamento		503
11.1	O conceito de abstração	504
11.2	Introdução à abstração de dados	505

11.3	Questões de projeto para tipos de dados abstratos	508
11.4	Exemplos de linguagem.....	509
	Entrevista: BJARNE STROUSTRUP – C++: nascimento, onipresença e críticas comuns.....	512
11.5	Tipos de dados abstratos parametrizados	527
11.6	Construções de encapsulamento	531
11.7	Nomeando encapsulamentos	535
	Resumo • Questões de revisão • Conjunto de problemas •	
	Exercícios de programação	540
Capítulo 12	Suporte para a programação orientada a objetos	545
12.1	Introdução.....	546
12.2	Programação orientada a objetos	546
12.3	Questões de projeto para programação orientada a objetos	550
12.4	Suporte para programação orientada a objetos em Smalltalk	555
12.5	Suporte para programação orientada a objetos em C++	558
	Entrevista: BJARNE STROUSTRUP – Sobre paradigmas e uma programação melhor.....	560
12.6	Suporte para programação orientada a objetos em Java.....	569
12.7	Suporte para programação orientada a objetos em C#.....	574
12.8	Suporte para programação orientada a objetos em Ada 95	575
12.9	Suporte para programação orientada a objetos em Ruby	580
12.10	Implementação de construções orientadas a objetos	584
	Resumo • Questões de revisão • Conjunto de problemas •	
	Exercícios de programação	587
Capítulo 13	Concorrência	592
13.1	Introdução.....	593
13.2	Introdução à concorrência no nível de subprograma.....	598
	Nota histórica.....	601
13.3	Semáforos.....	602
13.4	Monitores	607
13.5	Passagem de mensagens.....	609

13.6	Suporte de Ada para concorrência	611
13.7	Linhas de execução em Java	622
13.8	Linhas de execução em C#	630
13.9	Concorrência no nível de sentença	632
Resumo • Notas bibliográficas • Questões de revisão •		
Conjunto de problemas • Exercícios de programação.....		634
Capítulo 14 Tratamento de exceções e tratamento de eventos		639
14.1	Introdução ao tratamento de exceções	640
	Nota histórica.....	644
14.2	Tratamento de exceções em Ada	647
14.3	Tratamento de exceções em C++	654
14.4	Tratamento de exceções em Java	659
Entrevista: JAMES GOSLING – O nascimento de Java		660
14.5	Introdução ao tratamento de eventos	668
14.6	Tratamento de eventos com Java	670
Resumo • Notas bibliográficas • Questões de revisão •		
Conjunto de problemas • Exercícios de programação.....		675
Capítulo 15 Linguagens de programação funcional		680
15.1	Introdução.....	681
15.2	Funções matemáticas.....	682
15.3	Fundamentos das linguagens de programação funcional.....	685
15.4	A primeira linguagem de programação funcional: LISP	686
15.5	Uma introdução a Scheme.....	690
15.6	COMMON LISP	708
15.7	ML.....	709
15.8	Haskell.....	712
15.9	Aplicações de linguagens funcionais.....	717
15.10	Uma comparação entre linguagens funcionais e imperativas	718
Resumo • Notas Bibliográficas • Questões de Revisão •		
Conjunto de Problemas • Exercícios de Programação.....		721

Capítulo 16 Linguagens de programação lógica	726
16.1 Introdução.....	727
16.2 Uma breve introdução ao cálculo de predicados.....	727
16.3 Cálculo de predicados e prova de teoremas.....	731
16.4 Uma visão geral da programação lógica.....	734
16.5 As origens do Prolog.....	735
16.6 Os elementos básicos do Prolog.....	736
16.7 Deficiências do Prolog	751
16.8 Aplicações de programação lógica.....	757
Resumo • Notas bibliográficas • Questões de revisão •	
Conjunto de problemas • Exercícios de programação.....	759
Referências.....	763
Índice	775

Capítulo 1

Aspectos 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 valiosa para quem acredita que um conhecimento funcional de uma ou duas linguagens e programação é suficiente para cientistas da computação. Na sequência, descrevemos brevemente os principais domínios de programação. A seguir, como o livro avalia construções e recursos de linguagens, apresentamos uma lista de critérios que podem servir de base para tais julgamentos. Então, discutimos as duas maiores influências no projeto de linguagens: a arquitetura de máquinas e as metodologias de projeto de programas. Depois, introduzimos as diversas categorias de linguagens de programação. A seguir, descrevemos alguns dos principais compromissos que devem ser considerados durante o projeto de linguagens.

Como este livro trata também da implementação de linguagens de programação, este capítulo inclui uma visão geral 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 perguntam 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:

- *Capacidade aumentada para expressar ideias.* Acredita-se que a profundidade com a qual as pessoas podem pensar é influenciada pelo poder de expressividade da linguagem que elas usam para comunicar seus pensamentos. As pessoas que têm apenas um 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 podem descrever verbalmente ou expressar na escrita.

Programadores no processo de desenvolvimento 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 eles 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 faixa 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 forçado a usar uma 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 oferecem suporte diretamente. Por exemplo, um programador C que aprendeu a estrutura e os usos de matrizes associativas em Perl (Wall et al., 2000) pode projetar estruturas que as simulem em C. Em outras palavras, o estudo de conceitos de linguagens de programação constrói uma apreciação de recursos e construções valiosas das linguagens e incentiva os programadores a usá-las, mesmo quando a linguagem utilizada não oferece suporte direto para esses recursos e construções.

- *Embasamento para escolher linguagens adequadas.* Muitos programadores profissionais tiveram pouca educação formal em ciência da computação – em vez disso, aprenderam programação por conta própria ou em programas de treinamento em suas empresas. Tais programas normalmente ensinam apenas uma ou duas linguagens diretamente relevantes para os projetos atuais da organização. Possivelmente, muitos outros programadores receberam seu treinamento formal em um passado distante. As linguagens que aprenderam na época não são mais usadas, e muitos recursos agora disponíveis em outras não eram amplamente conhecidos. O resultado é que muitos programadores, quando podem escolher a linguagem para um novo projeto, continuam a usar aquela com a qual estão mais familiarizados, mesmo que ela seja pobre na sua adequação ao projeto. Se esses programadores conhecessem uma faixa mais ampla de linguagens e construções de linguagens, estariam mais capacitados a escolher a que inclui os recursos melhor adaptados às características do problema em questão.

Alguns dos recursos de uma linguagem podem ser simulados em outra. Entretanto, é sempre melhor 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 – em uma linguagem que não suporta tal recurso.

- *Habilidade aumentada 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 excitante, mas também exige aprendizado contínuo. O processo de aprender uma nova linguagem de programação pode ser longo e difícil, especialmente para alguém que esteja confortável com apenas uma ou duas e 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

Java muito mais facilmente (Arnold et al., 2006) 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 também tem o efeito colateral benéfico de ensinar a você mais sobre a primeira.

A Comunidade de Programação TIOBE disponibiliza um índice (http://www.tiobe.com/tiobe_index/index.htm) que funciona como indicador da popularidade relativa das linguagens de programação. Por exemplo, de acordo com o índice, Java, C e C++ foram as três linguagens mais populares em uso em outubro de 2008. Entretanto, dezenas de outras linguagens foram amplamente usadas nessa mesma época. Os dados do índice também mostram que a distribuição do uso das linguagens de programação está sempre mudando. Tanto o tamanho da lista das linguagens em uso quanto a natureza dinâmica das estatísticas implicam que os desenvolvedores de software devem aprender linguagens diferentes constantemente.

Por fim, é essencial que programadores em atividade conheçam o vocabulário e os conceitos fundamentais das linguagens de programação para poderem ler e entender descrições e avaliações de linguagens de programação, assim como a literatura promocional de linguagens e compiladores. Essas são as fontes de informações necessárias tanto para escolher quanto para 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 a por que as linguagens foram projetadas de uma determinada forma. Esse conhecimento muitas vezes leva à habilidade de usar uma linguagem de maneira mais inteligente e como ela 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.* Muitas linguagens de programação contemporâneas são grandes e complexas. É incomum um programador conhecer e usar todos os recursos da linguagem que ele utiliza. Ao estudar os conceitos de linguagens de programação, os programadores podem aprender sobre partes antes desconhecidas e não utilizadas das linguagens que eles 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 uma linguagem em particular 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 o ALGOL 60 (Backus et al., 1963) tivesse substituído o Fortran (Metcalf et al., 2004) no início dos anos 1960, porque ele era mais elegante e tinha sentenças de controle muito melhores do que o Fortran, dentre outras razões. Ele não o substituiu, em parte por causa dos programadores e dos gerentes de desenvolvimento de software da época; muitos não entendiam o projeto conceitual do 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 de estruturas de controle bem estruturadas, então falharam em ver as melhorias do ALGOL 60 ao relação ao Fortran.

É claro, muitos outros fatores contribuíram para a falta de aceitação do 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 mais bem informados, talvez linguagens melhores ganhassem de outras.

1.2 DOMÍNIOS DE PROGRAMAÇÃO

Computadores têm sido aplicados a uma infinidade de áreas, desde controlar usinas nucleares até disponibilizar jogos eletrônicos em telefones celulares. Por causa dessa diversidade de uso, linguagens de programação com objetivos muito diferentes têm sido desenvolvidas. Nesta seção, discutimos brevemente algumas das áreas de aplicação dos computadores e de suas linguagens associadas.

1.2.1 Aplicações científicas

Os primeiros computadores digitais, que apareceram nos anos 1940, foram inventados e usados para aplicações científicas. Normalmente, aplicações científicas têm estruturas de dados relativamente simples, mas requerem diversas computações de aritmética de ponto flutuante. As estruturas de dados mais comuns são os vetores e matrizes; as estruturas de controle mais comuns são 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 competidora era a linguagem *assembly* – logo, a eficiência era uma preocupação primordial. A primeira linguagem para aplicações científicas foi o Fortran. O ALGOL 60 e a maioria de seus descendentes também tinham a intenção de serem usados nessa área, apesar de terem sido projetados também para outras 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 do que o Fortran, o que explica por que o Fortran ainda é usado.

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. O COBOL 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 empresariais fora do desenvolvimento e evolução do COBOL. Logo, este livro inclui apenas discussões limitadas das estruturas em COBOL.

1.2.3 Inteligência Artificial

Inteligência Artificial (IA) é uma ampla área de aplicações computacionais caracterizada 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 habilidade de criar e de 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 uma de suas parentes próximas. No início dos anos 1970, entretanto, uma abordagem alternativa a algumas dessas aplicações apareceu – programação lógica usando a linguagem Prolog (Clocksin e Mellish, 2003). Mais recentemente, algumas aplicações de IA têm sido escritas em linguagens de sistemas como C, Scheme (Dybvig, 2003), um dialeto de LISP, e Prolog são introduzidas nos Capítulos 15 e 16, respectivamente.

1.2.4 Programação de sistemas

O sistema operacional e todas as ferramentas de suporte à programação de um sistema de computação são coletivamente conhecidos como seu **software de sistema**. Software de sistema são aplicativos usados quase continuamente e, dessa forma, devem ser eficientes. Além disso, tais aplicativos devem ter recursos de baixo nível que permitam aos aplicativos de software se comunicarem com dispositivos externos a serem escritos.

Nos anos 1960 e 1970, alguns fabricantes de computadores, como a IBM, a Digital e a Burroughs (agora UNISYS), desenvolveram linguagens especiais de alto nível orientadas à máquina para software de sistema que rodassem em suas máquinas. Para os computadores *mainframes* da IBM, a linguagem era PL/S, um dialeto de PL/I; para a Digital, era BLISS, uma linguagem apenas um nível acima da *assembly*; para a Burroughs, era o ALGOL Estendido.

O sistema operacional UNIX é escrito quase todo em C (ISO, 1999), o que o fez relativamente fácil de ser portado, ou movido, para diferentes máquinas. Algumas das características de C fazem que ele seja uma boa escolha para a programação de sistemas. C é uma linguagem de baixo nível, têm uma execução eficiente e não sobrecarrega o usuário com muitas restrições de segurança. Os programadores de sistemas são excelentes e não acreditam que precisam de tais restrições. Alguns programadores que não trabalham com a programação de sistemas, no entanto, acham C muito perigoso para ser usado em sistemas de software grandes e importantes.

1.2.5 Software para a Web

A *World Wide Web* é mantida por uma eclética coleção de linguagens, que vão desde linguagens de marcação, como XHTML, que não é de programação, até linguagens de programação de propósito geral, como Java. Dada a necessidade universal por 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 XHTML. Tal código é normalmente escrito com uma linguagem de *scripting*, como JavaScript ou PHP. 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 do 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. Iremos também avaliar esses recursos, focando em 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 praticamente impossível 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 seguintes¹. 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 alguns dos critérios são amplos – e, de certa forma, vagos – como a facilidade de escrita, enquanto outros são construções específicas de linguagens, como o tratamento de exceções. Apesar de a discussão parecer implicar que os critérios têm uma importância idêntica, não é o caso.

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

Característica	CRITÉRIOS		
	LEGIBILIDADE	FACILIDADE DE ESCRITA	CONFIABILIDADE
Simplicidade	•	•	•
Ortogonalidade	•	•	•
Tipos de dados	•	•	•
Projeto de sintaxe	•	•	•
Supporte para abstração		•	•
Expressividade		•	•
Verificação de tipos			•
Tratamento de exceções			•
Apelidos restritos			•

¹ O quarto critério principal é o custo, que não foi incluído na tabela porque é apenas superficialmente relacionado aos outros três critérios e às características que os influenciam.

1.3.1 Legibilidade

Um dos critérios mais importantes para julgar uma linguagem de programação é a facilidade com a qual os programas podem ser lidos e entendidos. Antes de 1970, o desenvolvimento de software era amplamente pensado em termos da escrita de código. As principais características positivas das linguagens de programação eram a eficiência e a legibilidade de máquina. 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 principal do ciclo, particularmente em termos de custo. Como a facilidade de manutenção é determinada, em grande parte, pela legibilidade dos programas, a legibilidade se tornou uma medida importante da qualidade dos programas e das linguagens de programação, o que representou um marco na sua evolução. Ocorreu uma transição de foco bem definida: da orientação à máquina à orientação às pessoas.

A legibilidade deve ser considerada no contexto do domínio do problema. Por exemplo, se um programa que descreve uma computação é escrito em uma linguagem que não foi projetada para tal uso, ele pode não ser natural e 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 fortemente 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 grande 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 aprender um subconjunto diferente daquele com o qual o leitor está familiarizado.

Outra 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, as quatro

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, isso tornaria o programa mais confuso, tanto para o autor quanto para 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 seja 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 muito 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; e como as sentenças são simples, são necessárias mais 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 legal 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 (o nome ortogonal vem do conceito matemático de vetores ortogonais, independentes uns dos outros). A ortogonalidade vem de uma simetria de relacionamentos entre primitivas. Uma falta de ortogonalidade leva a exceções às regras de linguagem. Por exemplo, deve ser possível, em uma linguagem de programação que possibilita o uso de

ponteiros, definir que um aponte para qualquer tipo específico definido na linguagem. Entretanto, se não for permitido aos ponteiros apontar para vetores, muitas estruturas de dados potencialmente úteis definidas pelos usuários não poderiam ser definidas.

Podemos ilustrar o uso da ortogonalidade como um conceito de projeto ao comparar um aspecto das linguagens *assembly* dos mainframes da IBM com a série VAX de minicomputadores. Consideramos uma situação simples: adicionar 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 registradores. A semântica desses 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 operand_1, operand_2
```

cuja semântica é

```
operand_2 ← contents(operand_1) + contents(operand_2)
```

Nesse caso, cada operando pode ser 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 para especificar operandos, as quais podem ser combinadas de todas as maneiras possíveis. O projeto da IBM não é ortogonal. Apenas duas combinações de operandos são legais (entre quatro possibilidades), 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 adicionar 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 é fortemente 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 testemunhar sobre a dificuldade de aprender suas muitas exceções de regras (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 (**structs**), os registros podem ser retornados por funções, mas os vetores não. Um membro de uma

estrutura pode ser de qualquer tipo de dados, exceto **void** ou uma estrutura do mesmo tipo. Um elemento de um vetor pode ser qualquer tipo de dados, 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 um 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 adicionados juntos. Entretanto, se `a` for um ponteiro, afeta o valor de `b`. Por exemplo, se `a` aponta para um valor de ponto flutuante que ocupa quatro bytes, o valor de `b` deve ser ampliado – nesse caso, multiplicado por 4 – antes que seja adicionado a `a`. Logo, o tipo de `a` afeta o tratamento do valor de `b`. O contexto de `b` afeta seu significado.

Muita ortogonalidade também pode causar problemas. Talvez a linguagem de programação mais ortogonal seja o ALGOL 68 (van Wijngaarden et al., 1969). Cada construção de linguagem no ALGOL 68 tem um tipo, e não existem restrições nesses tipos. Além disso, suas construções produzem 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 de uma combinação de um número relativamente pequeno de construções primitivas e um 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 uma na qual as computações são feitas basicamente pela aplicação de funções para parâmetros informados. 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 C++. Outros fatores, como a eficiência, entretanto, têm prevenido 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 dois exemplos de escolhas de projeto sintáticas que afetam a legibilidade:

- *Formato dos identificadores.* Restringir os identificadores a tamanhos muito curtos piora a legibilidade. Se os identificadores podem ter no máximo seis caracteres, como no Fortran 77, é praticamente impossível usar nomes conotativos para variáveis. Um exemplo mais extremo é o ANSI* BASIC (ANSI, 1978b), no qual um identificador poderia ser formado por apenas uma letra ou por uma letra seguida de um dígito. Outras questões de projeto relacionadas ao formato dos identificadores são discutidas no Capítulo 5.
- *Palavras especiais.* A aparência de um programa e sua legibilidade são fortemente 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 sofrem porque os grupos de sentenças são sempre terminados da mesma forma, o que torna difícil determinar qual grupo está sendo finalizado quando um `end` ou um `}` aparece. O Fortran 95 e Ada 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 simpli-

* N. de T.: American National Standards Institute, Instituto Nacional Norte-Americano de Padrões.

cidade resultante ao serem usadas menos palavras reservadas, como em C++, 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, no 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 é uma ajuda óbvia à 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 sobre os comandos de *shell* do UNIX (Raymond, 2004) é que a 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 */regular_expression/* busca uma subcadeia que casa com a expressão regular. Preceder isso com g faz ele 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 casada devem ser impressas. Logo *g/regular_expression/p*, que pode ser abreviado como grep, imprime todas as linhas em um arquivo que contenham subcadeias que casem com a expressão regular.)

1.3.2 Facilidade de escrita

A facilidade de escrita é a medida do quanto facilmente uma linguagem pode ser usada para criar programas para um domínio. A maioria das características de linguagem que afetam a legibilidade também afeta a facilidade de escrita. Isso é derivado do fato de que o processo de escrita de um programa requer que o programador releia sua 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 razoável comparar a facilidade de escrita de duas linguagens no contexto de uma aplicação em particular quando uma delas foi projetada para tal aplicação

e a outra não. Por exemplo, as facilidades de escrita do Visual BASIC (VB) e do C são drasticamente diferentes para criar um programa com uma interface gráfica com o usuário, 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 estarão familiarizados com todas. 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) do que os usados. Pode até mesmo ser possível, conforme destacado por Hoare (1973), usar recursos desconhecidos accidentalmente, com resultados inesperados. Logo, um número menor de construções primitivas e um conjunto de regras consistente para combiná-las (isso é, ortogonalidade) é muito melhor do que diversas construções primitivas. Um programador pode projetar uma solução para um problema complexo após aprender apenas um conjunto simples de construções primitivas.

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

1.3.2.2 Suporte à abstração

Brevemente, **abstração** significa a habilidade de definir e usar estruturas ou operações complicadas de forma a permitir que muitos dos detalhes sejam ignorados. A abstração é um conceito fundamental no projeto atual de linguagens de programação. O grau de abstração permitido por uma linguagem de programação e a naturalidade de sua expressão são importantes para sua facilidade de escrita. As linguagens de programação podem oferecer suporte a duas categorias de abstrações: processos e dados.

Um exemplo simples da abstração de processos é o uso de um subprograma para implementar um algoritmo de ordenação necessário diversas vezes em um programa. Sem o subprograma, o código de ordenação teria de ser replicado em todos os lugares onde fosse preciso, o que tornaria o programa muito mais longo e tedioso de ser escrito. Talvez o mais importante, se o subprograma não fosse usado, o código que usava o subprograma de ordenação estaria mesclado com os detalhes do algoritmo de ordenação, obscurecendo o fluxo e a intenção geral do código.

Como um exemplo de abstração de dados, considere uma árvore binária que armazena dados inteiros em seus nós. Tal árvore poderia ser implementada em uma linguagem que não oferece suporte a ponteiros e gerenciamento de memória dinâmica usando um monte (*heap*), como no Fortran 77, com o

uso de três vetores inteiros paralelos, onde dois dos inteiros são usados como índices para especificar nós filhos. Em C++ e Java, essas árvores podem ser implementadas utilizando uma abstração de um nó de árvore na forma de uma simples classe com dois ponteiros (ou referências) e um inteiro. A naturalidade da última representação torna muito mais fácil escrever um programa que usa árvores binárias nessas linguagens do que um em Fortran 77. É uma simples questão do domínio da solução do problema da linguagem ser mais próxima do domínio do problema.

O suporte geral para abstrações é um fator importante na facilidade de escrita de uma linguagem.

1.3.2.3 Expressividade

A expressividade em uma linguagem pode se referir a diversas características. Em uma linguagem como APL (Gilman e Rose, 1976), 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 desleitante. Por exemplo, em C, a notação `count++` é mais conveniente e menor do que `count = count + 1`. Além disso, o operador booleano `and then` em Ada é uma maneira conveniente de especificar avaliação em curto-círcuito 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

Um programa é dito confiável quando está de acordo com suas especificações em todas as condições. As seguintes subseções 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. A verificação de tipos é um fator importante na confiabilidade de uma linguagem. Como a verificação de tipos em tempo de execução é cara, a verificação em tempo de compilação é mais desejável. Além disso, quanto mais cedo os erros nos programas forem detectados, menos caro é 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 real em uma chamada à função não era verificado para determinar se seu tipo casava com aquele do parâmetro formal correspondente na função. Uma variável do tipo `int` poderia ser usada como um parâmetro real em uma chamada à uma função que esperava um tipo `float` como seu parâmetro formal, e nem o compilador nem o sistema de tempo de execução teriam detectado 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 ponto flutuante 23, se um inteiro 23 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 serem diagnosticados². A versão atual de C eliminou esse problema ao requerer 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 é uma ajuda óbvia para a confiabilidade. Tal facilidade é chamada de **tratamento de exceções**. Ada, C++ e Java incluem diversas capacidades para tratamento de exceções, mas tais facilidades são praticamente inexistentes em muitas linguagens amplamente usadas, como C e Fortran. O tratamento de exceções é discutido no Capítulo 14.

1.3.3.3 Utilização de apelidos

Em uma definição bastante informal, **apelidos** são permitidos quando é possível ter um ou mais nomes para acessar a mesma célula de memória. Atualmente, é amplamente 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 configurados para apontarem para a mesma variável, o que é possível na maioria das linguagens. O programador deve sempre lembrar que trocar 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.

² Em resposta a isso e a outros problemas similares, os sistemas UNIX incluem um programa utilitário chamado `lint`, que verifica os programas em C para checar tais problemas.

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 oferece maneiras naturais para expressar os algoritmos requeridos irá necessariamente usar abordagens não naturais, menos prováveis de serem corretas em todas as situações possíveis. Quanto mais fácil é escrever um programa, mais provavelmente ele estará 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 definitivo 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 necessariamente serem mais difíceis de aprender, normalmente elas o são.

Segundo, 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, o custo de compilar programas na linguagem. Um grande impedimento 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 melhores.

Quarto, o custo de executar programas escritos em uma linguagem é amplamente influenciado pelo projeto dela. Uma linguagem que requer muitas verificações de tipos em tempo de execução proibirá uma execução rápida de código, independentemente da qualidade do compilador. Apesar de eficiência de execução ser a principal preocupação no projeto das primeiras linguagens, atualmente 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 pouca ou nenhuma otimização é feita, a compilação pode ser feita muito mais rapidamente do que se um esforço significativo for feito 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 que estão iniciando a programação – os quais geralmente compilam seus programas

diversas vezes durante o desenvolvimento, mas que usam pouco tempo de execução de código (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 estavam 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 terão uma chance muito menor de se tornarem amplamente usados. Por exemplo, o alto custo da primeira geração de compiladores Ada ajudou a prevenir que Ada tivesse se tornado 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 de manter programas, que inclui tanto as correções quanto as modificações para adicionar novas funcionalidades. O custo da manutenção de software depende de um número de características de linguagem, principalmente da legibilidade. Como que 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. Tem sido estimado que, para grandes sistemas de software com tempos de vida relativamente longos, os custos de manutenção podem ser tão grandes como o dobro ou o quádruplo dos custos de desenvolvimento (Sommerville, 2005).

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, fortemente influenciada pelo grau de padronização da linguagem. Algumas, como o BASIC, não são padronizadas, fazendo os programas escritos nessas linguagens serem difíceis de mover de uma implementação para outra.

A padronização é um processo difícil e consome muito tempo. Um comitê começou a trabalhar em uma versão padrão de C++ em 1989, aprovada em 1998.

A **generalidade** (a aplicabilidade a uma ampla faixa 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 exatamente mensurável. Independentemente disso, são conceitos úteis e fornecem ideias valiosas para o projeto e para a avaliação de linguagens de programação.

Uma nota 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 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 habilidade de atrair um grande número de usuários. Essas características geralmente entram em conflito.

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 50 anos tem sido projetada considerando a principal arquitetura de computadores, chamada de **arquitetura de von Neumann**, cujo nome é derivado de um de seus criadores, John von Neumann (pronuncia-se “von Noyman”). 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 têm sido baseados nessa arquitetura. A estrutura geral de um computador von Neumann é mostrada na Figura 1.1.

Por causa da 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 requer apenas uma simples instrução de desvio. Essa eficiência desencoraja o uso de recursão para repetição, embora a recursão seja às vezes mais natural.

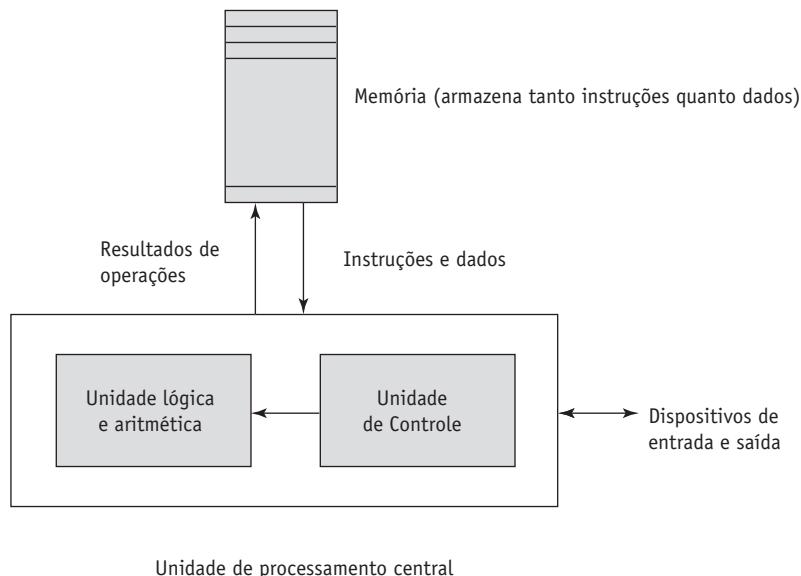


Figura 1.1 A arquitetura de computadores de von Neumann.

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 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:

```

initialize o contador de programa
repita para sempre
    obtenha a instrução apontada pelo contador de programa
    incremente o contador de programa para que esse 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 estiver completa. Em um sistema de

computação no qual mais de um programa de usuário pode estar na memória em um dado tempo, esse processo é muito mais complexo.

Deve ser possível, em uma linguagem de programação, definir um ponteiro para qualquer tipo específico definido na linguagem. Conforme mencionado, uma linguagem funcional, ou aplicativa, é uma na qual a principal forma de computação é a aplicação de funções para parâmetros fornecidos. A programação pode ser feita em uma linguagem funcional sem os tipos de variáveis usados nas linguagens imperativas, sentenças de atribuição e 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 von Neumann seja projetado e permita a execução eficiente de programas em linguagens funcionais. Dentre aqueles que já reclamaram desse fato, o mais eloquente foi John Backus (1978), o principal projetista da versão original do Fortran.

Apesar do fato 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 do que o uso de uma funcional. Logo, muitos 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 no processo de desenvolvimento de software quanto no projeto de linguagens de programação.

Uma razão importante para essa pesquisa foi a mudança no custo maior da computação, de hardware para software, à medida que os custos de hardware declinavam e os de programação aumentavam. Aumentos na produtividade dos programadores eram relativamente pequenos. Além disso, problemas maiores e mais complexos eram resolvidos por computadores. Em vez de simplesmente resolver conjuntos de equações para simular rotas de satélites, como no início dos anos 1960, programas estavam sendo escritos para tarefas grandes e complexas, como controlar grandes estações de refinamento 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 chamados de projeto descendente (*top-down*) e de refinamento passo a passo. As principais deficiências que foram descobertas nas linguagens de programação eram a incompletude da verificação de tipos e a inadequação das sentenças de controle (que requeriam um uso intenso de desvios incondicionais, também conhecidos como *gotos*).

No final dos anos 1970, iniciou-se uma mudança das metodologias de projeto de programas, da orientação aos procedimentos para uma orientação aos dados. Os métodos orientados a dados enfatizam a modelagem de dados, focando 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 usadas para a implementação. A primeira a fornecer suporte limitado para a abstração de dados foi o SIMULA 67 (Birtwistle et al., 1973), apesar de não ter se tornado popular por causa disso. A vantagem 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 e 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 possibilidade 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 usada como muitas outras linguagens, o suporte à programação orientada a objetos é agora parte da maioria das linguagens imperativas populares, incluindo Ada 95 (ARM, 1995), Java e C++. Conceitos de orientação a objetos também encontraram seu caminho em linguagens funcionais como em CLOS (Bobrow et al., 1988) e 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. Boa parte da pesquisa vem ocorrendo em programação orientada a procedimentos nos últimos anos, especialmente na área de concorrência. Esses esforços de pesquisa trouxeram a necessidade de recursos de linguagem para criar e controlar unidades de programas concorrentes. Ada, Java e C# incluem tais capacidades, que serão discutidas 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

Linguagens de programação são normalmente 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. 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 do paradigma orientado a procedimentos usado normalmente nas linguagens imperativas, as extensões a uma linguagem imperativa necessárias para oferecer suporte à programação orientada a objetos não são tão complexas. 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.

As linguagens visuais são uma subcategoria das imperativas. A mais popular é o Visual BASIC .NET (VB.NET) (Deitel et al., 2002). Essas linguagens (ou suas implementações) incluem capacidades para geração de segmentos de códigos que podem ser copiados de um lado para outro. Elas foram chamadas de linguagens de quarta geração, apesar de esse nome já não ser mais usado. As linguagens visuais fornecem uma maneira simples de gerar interfaces gráficas de usuário para os programas. Por exemplo, em VB.NET, o código para produzir uma tela com um controle de formulário, como um botão ou uma caixa de texto, pode ser criado com uma simples tecla. Tais capacidades estão agora disponíveis em todas as linguagens .NET.

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*, dentre elas Perl, JavaScript 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 a ordem de execução específica das instruções ou sentenças deve ser incluída. Em uma linguagem baseada em regras, entretanto, estas são especificadas sem uma ordem em particular, 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 três 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/linguagens de programação híbridas. As de marcação não são de programação. Por exemplo, XHTML, a linguagem de marcação mais utilizada, es-

pecifica a disposição da informação em documentos Web. Entretanto, algumas capacidades de programação foram colocadas em extensões de XHTML e XML. Dentre essas, 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 brevemente introduzidas no Capítulo 2. Elas não podem ser comparadas a qualquer linguagem de programação completa e, dessa forma, não serão discutidas após esse capítulo.

Diversas linguagens de propósito especial têm aparecido nos últimos 50 anos. Elas vão desde a Report Program Generator (RPG), usada para produzir relatórios de negócios; até a Automatically Programmed Tools (APT), para instruir ferramentas de máquina programáveis; e a General Purpose Simulation System (GPSS), para sistemas de simulação. Este livro não discute linguagens de propósito especial, por causa de sua aplicabilidade restrita e pela dificuldade de compará-las com outras.

1.6 TRADE-OFFS NO PROJETO DE LINGUAGENS

Os critérios de avaliação de linguagens de programação descritos na Seção 1.3 fornecem um *framework* para o projeto de linguagens. Infelizmente, esse *framework* é contraditório. Em seu brilhante artigo sobre o projeto de linguagens, Hoare (1973) afirma que “existem tantos critérios importantes, mas conflitantes, que sua reconciliação e satisfação estão dentre 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 legais. Esse passo adiciona muito ao custo de execução de programas Java que contenham um grande número de referências a elementos de vetores. C não requer a verificação da faixa de índices – dessa forma, os programas em C executam mais rápido do que programas semanticamente equivalentes em Java, apesar de esses serem mais confiáveis. Os projetistas de Java trocaram eficiência de execução por confiabilidade.

Como outro exemplo de critérios conflitantes que levam diretamente aos *trade-offs* de projeto, considere o caso de APL, que inclui um poderoso conjunto de operadores para operandos do tipo matriz. Dado o grande número de operadores, um número significativo de novos símbolos teve de ser incluído em APL para apresentar 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 matrizes, 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 uma legibilidade muito pobre. Uma expressão compacta e concisa tem certa beleza matemática, mas é difícil para qualquer um, exceto o autor, entendê-la. O renomado autor Daniel McCracken (1970) afirmou em certa ocasião que levou quatro

horas para ler e entender um programa APL de quatro linhas. O projetista trocou 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 oferece suporte a 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 requer muitos comprometimentos 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 mais discutidas 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 pode 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 (se 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 requer 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 mais alto nível 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 ao usuário em níveis mais altos. Por exemplo, um sistema operacional e um compilador C fornecem um computador virtual C. 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

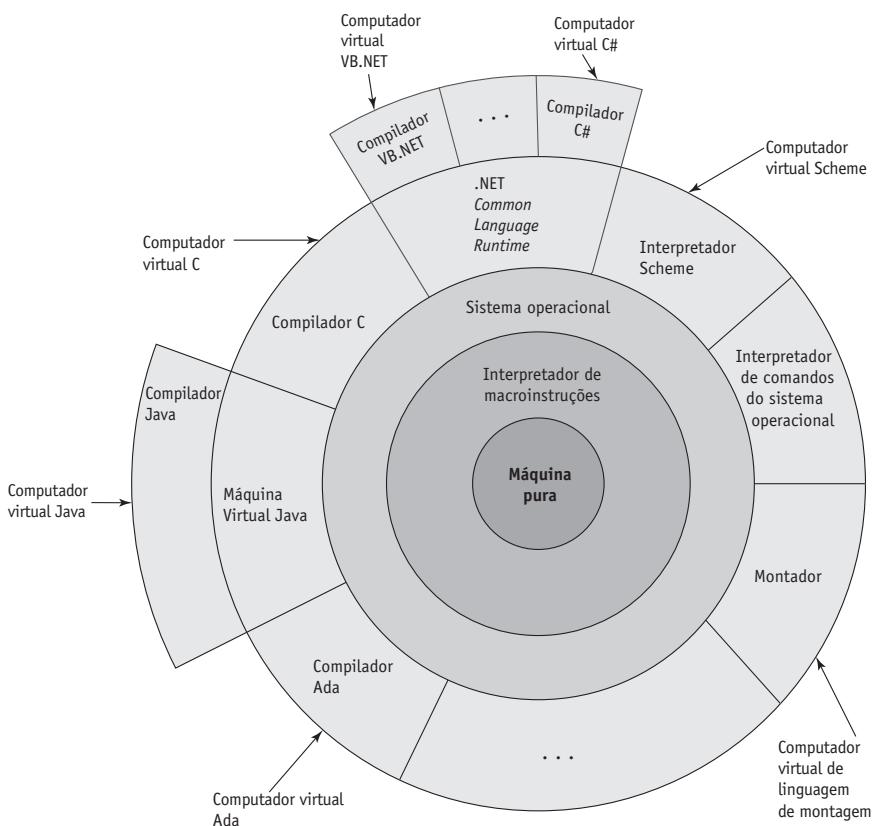


Figura 1.2 Interface em camadas de computadores virtuais, fornecida por um sistema de computação típico.

de implementação é uma aplicação de partes das teorias 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 Ada, é feita por meio de compiladores.

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

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

O analisador sintático obtém as unidades léxicas do analisador léxico e as utiliza para construir estruturas hierárquicas chamadas de **árvore de análise sintática** (*parse trees*) que 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. Tanto a análise léxica quanto 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 máquina³. 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 a um nível um pouco mais alto do que uma linguagem de montagem. O analisador semântico é parte do gerador de código intermediário, que verifica erros difíceis (ou impossíveis) de ser detectados durante a análise sintática, como erros de tipos.

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. Na verdade, alguns compiladores são incapazes de fazer quaisquer otimizações significativas. Esse tipo de compilador seria usado em situações nas quais a velocidade de execução do programa traduzido é bem menos importante do que a velocidade de com-

³ Note que as palavras *programa* e *código* são usadas como sinônimos.

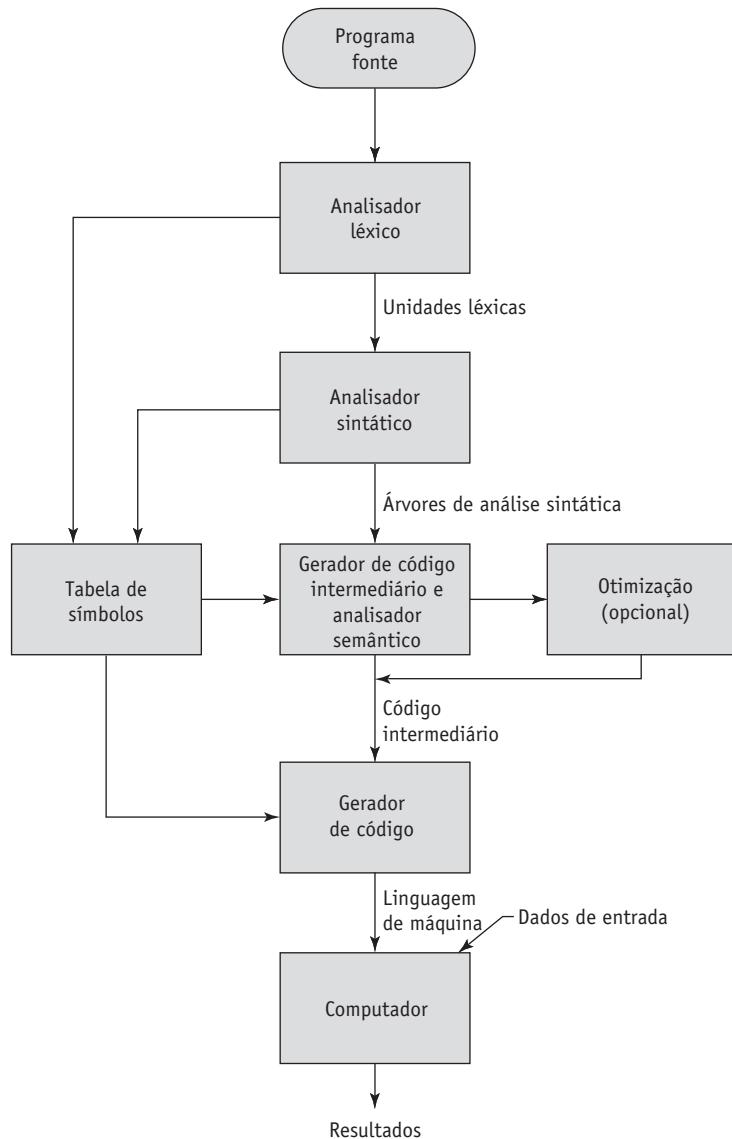


Figura 1.3 O processo de compilação.

pilação. Um exemplo de tal situação é um laboratório de computação para programadores iniciantes. Na maioria das situações comerciais e industriais, a velocidade de execução é mais importante do que a como de compilação, logo a otimização é rotineiramente desejável. Como muitos tipos de otimização são difíceis de serem feitas 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 a linguagem de máquina gerada por um compilador poder ser executada diretamente no hardware, ela praticamente sempre precisa rodar com algum outro código. A maioria dos programas de usuário também necessita de programas do sistema operacional. Dentre os mais comuns, estão os programas para entrada e saída. O compilador constrói chamadas para os programas de sistema requeridos quando eles são necessitados pelo programa de usuário. Antes de os programas em linguagem de máquina produzidos por um computador poderem ser executados, aqueles requeridos do 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. O código de usuário e de sistema juntos são chamados um **módulo de carga**, ou uma **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*).

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

A velocidade de conexão entre a memória de um computador e seu processador normalmente determina a velocidade do computador. As instruções normalmente podem ser executadas mais rapidamente do que movidas para o processador de forma que possam ser executadas. Essa conexão é chamada de **gargalo de von Neumann**; é o fator limitante primário na velocidade dos computadores que seguem a arquitetura de von Neumann e tem sido uma das motivações primárias para a pesquisa e o desenvolvimento de computadores paralelos.

1.7.2 Interpretação pura

A interpretação pura reside no oposto (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 facilmente indicar a linha do código fonte e o nome do vetor. Em contrapartida, esse método tem a 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 do que as instruções de linguagem de máquina (apesar de poderem ser bem menos sentenças do 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, em vez de a conexão entre o processador e a memória, é o gargalo de um interpretador puro.

Outra desvantagem da interpretação pura é que ela normalmente requer mais espaço. A tabela de símbolos deve estar presente durante a interpretação e o programa fonte deve ser armazenado em um formato para fácil acesso e modificação em vez de um que forneça um tamanho mínimo.

Apesar de algumas das primeiras linguagens mais simples dos anos 1960 serem puramente interpretadas (APL, SNOBOL e LISP), nos anos 1980, a abordagem já era raramente usada em linguagens de alto nível. Entretanto, nos últimos anos, a interpretação pura teve uma volta 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 nível para uma linguagem intermediária projetada para facilitar

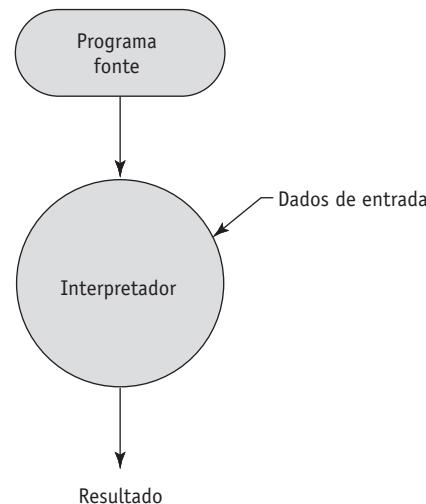


Figura 1.4 Interpretação pura.

a interpretação. Os **Sistemas de implementação híbridos** são mais rápidos do que a interpretação pura, porque as sentenças da linguagem fonte são decodificadas apenas uma vez.

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

Perl é implementado como um sistema híbrido. Os programas em Perl eram 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 *bytecode*, fornece portabilidade para qualquer má-

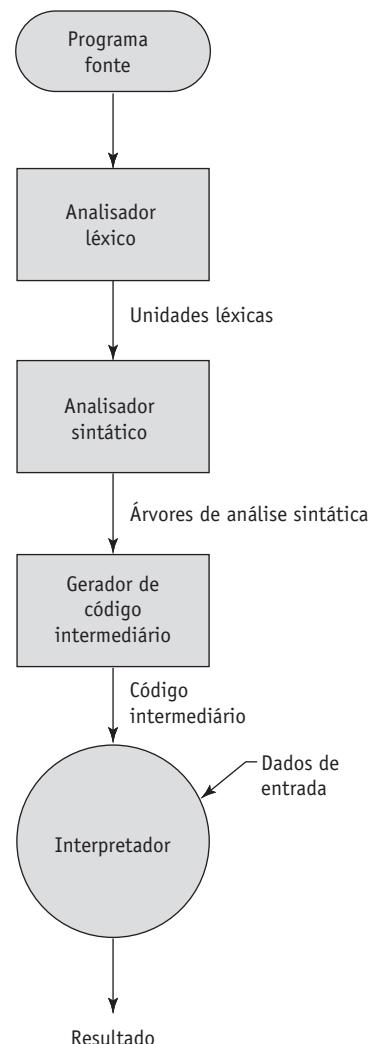


Figura 1.5 Sistema de implementação híbrido.

quina 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 para código de máquina de forma a possibilitar uma execução mais rápida.

Um sistema de implementação *Just-in-Time* (JIT) inicialmente traduz os programas para 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. 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 estudo (relativamente) livre de erros ser alcançado, os programas são compilados para aumentar sua velocidade de execução.

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"
```

faz ele 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. No último caso, o desenvolvimento e a manutenção de software é enormemente melhorada. Logo, as características de uma linguagem de programação não são a única medida da capacidade de desenvolvimento de um sistema. Agora, descrevemos 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 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 fazia mais difícil de aprender e usar. Entretanto, o UNIX é agora usado por meio de uma interface gráfica com o usuário (GUI) que roda 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 de JavaScript), J# (a versão da Microsoft de Java) ou C++ gerenciado.

O NetBeans é um ambiente usado primariamente para o desenvolvimento de aplicações Web usando Java, mas também oferece suporte a 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 diversas razões: aumenta nossa capacidade de usar diferentes construções ao escrever programas, permite que escolhamos linguagens para os projetos de forma mais inteligente e torna mais fácil o aprendizado de novas linguagens.

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

Dentre 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, no qual uma longa lista de *trade-offs* deve ser levada em consideração na escolha 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 50 anos?
4. Que linguagem de programação tem dominado as aplicações de negócios nos últimos 50 anos?
5. Que linguagem de programação tem dominado a Inteligência Artificial nos últimos 50 anos?
6. Em que linguagem o UNIX é escrito?
7. Qual é a desvantagem de ter muitas características 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 um critério de projeto primário?
11. Que sentença de controle primitiva é usada para construir sentenças de controle mais complicadas em linguagens que não as têm?
12. Que construção de uma linguagem de programação fornece abstração de processos?
13. O que significa para um programa ser confiável?
14. Por que verificar os tipos dos parâmetros de um subprograma é importante?
15. O que são apelidos?
16. O que é o tratamento de exceções?
17. Por que a legibilidade é importante para a facilidade de escrita?
18. Como o custo de compiladores para uma linguagem está relacionado ao projeto dela?

19. Qual tem sido a influência mais forte no projeto de linguagens de programação nos últimos 50 anos?
20. Qual é o nome da categoria de linguagens de programação cuja estrutura é dita pela arquitetura de computadores de von Neumann?
21. Que duas deficiências das linguagens de programação foram descobertas como um resultado da pesquisa em desenvolvimento de software dos anos 1970?
22. Quais são os três recursos fundamentais de uma linguagem orientada a objetos?
23. Qual foi a primeira linguagem a oferecer suporte aos três recursos fundamentais da programação orientada a objetos?
24. Dê um exemplo de dois critérios de projeto de linguagens que estão em conflito direto um com o outro.
25. Quais são os três métodos gerais de implementar uma linguagem de programação?
26. Qual produz uma execução de programas mais rápida, um compilador ou um interpretador puro?
27. Que papel a tabela de símbolos tem em um compilador?
28. O que faz um ligador?
29. Por que o gargalo de von Neumann é importante?
30. Quais são as vantagens de implementar uma linguagem com um interpretador puro?

CONJUNTO DE PROBLEMAS

1. Você acredita que nossa capacidade de abstração é influenciada por nosso domínio de linguagens? Defenda sua opinião.
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 dar a favor da ideia de uma única linguagem para todos os domínios de programação?
4. Que argumentos você pode dar contra a ideia de uma única linguagem para todos os domínios de programação?
5. Nomeie 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 um símbolo de fechamento de chaves para marcar o término de todas as sentenças compostas. Quais são os argumentos a favor e contra essa decisão de projeto?
8. Muitas linguagens distinguem 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. Quais recursos principais uma linguagem de programação perfeita deveria incluir, em sua opinião?

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? (Você não necessariamente saberá isso sem pesquisar).
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 o 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 múltiplas 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.

Capítulo 2

Evolução das Principais Linguagens de Programação

- 2.1** Plankalkül de Zuse
- 2.2** Programação de hardware mínima: pseudocódigos
- 2.3** O IBM 704 e Fortran
- 2.4** Programação funcional: LISP
- 2.5** O primeiro passo em direção à sofisticação: ALGOL 60
- 2.6** Informatizando os 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 dos 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** Combinando 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** Uma linguagem baseada em C para o novo milênio: C#
- 2.20** Linguagens híbridas de marcação/programação

Este capítulo descreve o desenvolvimento de uma coleção de linguagens de programação, explorando o ambiente no qual cada uma delas foi projetada e focando nas contribuições da linguagem e na motivação para seu desenvolvimento. Descrições gerais de linguagens não são incluídas; em vez disso, discutimos alguns dos novos recursos introduzidos por cada linguagem. De interesse, em particular, estão 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 que não serão familiares a muitos leitores. Aqueles que acharem isso um obstáculo podem preferir postergar a leitura deste capítulo até que o resto do livro tenha sido estudado.

A escolha sobre quais linguagens discutir foi subjetiva, e alguns leitores irão notar de maneira desapontada a ausência de uma ou mais de suas linguagens favoritas. Entretanto, para manter essa cobertura histórica em um tamanho razoável, foi necessário deixar de fora algumas linguagens que alguns apreciam muito. As escolhas foram feitas baseadas 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 tarde.

Ao longo do capítulo, as versões iniciais das linguagens são geralmente discutidas em ordem cronológica. Entretanto, versões subsequentes das linguagens aparecem com sua versão inicial, em vez de em seções posteriores. Por exemplo, o Fortran 2003 é discutido na seção com o 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. Nenhum deles é descrito neste capítulo; o objetivo é simplesmente ilustrar a aparência de programas nessas linguagens. Leitores familiarizados com qualquer uma 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 (o exemplo de LISP é discutido 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 de exemplo. Além disso, no programa do 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.

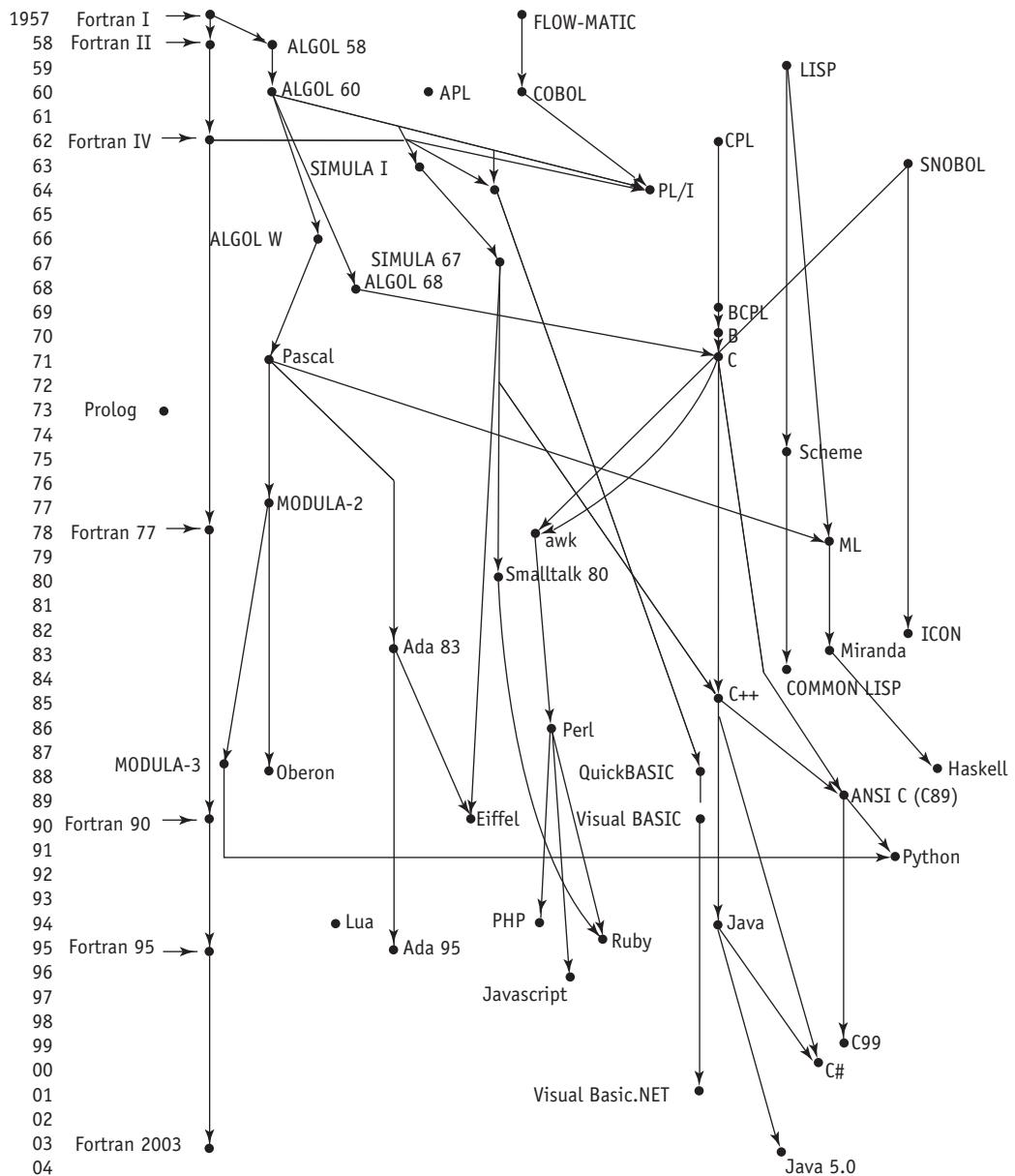


Figura 2.1 Genealogia das principais linguagens de programação de alto nível.

2.1 PLANKALKÜL DE ZUSE

A primeira linguagem de programação discutida neste capítulo é altamente não usual em diversos aspectos. Primeiro, ela nunca foi implementada. Segundo, apesar de ter sido desenvolvida em 1945, sua descrição não foi pu-

blicada até 1972. Como poucas pessoas estavam familiarizadas com a linguagem, algumas de suas capacidades não apareceram em outras 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, a guerra havia destruídos 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 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 armazenar o bit mais significativo da parte normalizada da fração de um valor de ponto flutuante.

Além dos tipos escalares usuais, Plankalkül incluía vetores e registros. Os registros podiam incluir registros aninhados. Apesar de a linguagem não ter um comando de desvio incondicional (*goto*) explícito, ela incluía uma sentença iterativa similar ao **for** de Ada. Ela também tinha o comando **fin** com um índice que especificava um salto a partir 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 do que qualquer outro escrito antes de 1945. Estavam incluídos programas para ordenar vetores de números; testar a conectividade de um dado grafo; realizar operações de inteiros e de ponto flutuante, incluindo a raiz quadrada; e realizar análise sintática em fórmulas lógicas que tinham parê-

teses 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 correntes. 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 no meio do século XIX. A última linha de cada sentença em Plankalkül continha os nomes dos tipos para as variáveis mencionadas na primeira linha. Essa notação é bastante intimidadora vista pela primeira vez.

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 bastante 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 em 1945, praticamente isolado.

2.2 PROGRAMAÇÃO DE HARDWARE MÍNIMA: 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 lentos, não confiáveis, caros e com memórias extremamente pequenas, as máquinas daquela época eram difíceis de programar por causa da 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. Dentre os proble-

mas, 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 faz com que os programas sejam de difícil leitura. 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 em outra que não o final desse 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 a 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 com as linguagens de máquina e serviram de motivação para a invenção de montadores e de linguagens de montagem. Além disso, a maioria dos problemas de programação da época era numérica e requeria operações aritméticas de ponto flutuante e de 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 de tais 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, era 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 assumir 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. Alguns dos códigos de operação eram

```
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 simplificou a programação, mas ao custo do tempo de execução – a interpretação de programas em Short Code era aproximadamente 50 vezes mais lenta do que a execução de código de máquina.

2.2.2 Speedcoding

Em outros lugares, sistemas de interpretação estavam sendo desenvolvidos para estender linguagens de máquina para 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 para 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 logarítmico. Desvios condicionais e incondicionais e conversões de entrada e saída também faziam parte da arquitetura virtual. Para se 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 poderia ser feita em 12 instruções Speedcoding. Backus afirmava que problemas que levariam duas semanas para serem programados em código de máquina poderiam ser programados em poucas horas usando 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” nomeados 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 mesmo isso ser uma grande melhoria em relação ao código de máquina, pois fazia com que os programas fonte fossem muito menores. 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 relocalizáveis para resolver 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 alojar armazenamento (Wilkes et al., 1951, 1957). Esse era, na verdade, um avanço importante e fundamental.

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

2.3 O 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 do Fortran. Pode-se argumentar que, se não fosse a IBM com o 704 e o Fortran, logo seria 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 teriam de ser simuladas em software, um processo que consumia muito tempo. 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 de interpretação, ao menos para a computação científica. A inclusão de hardware de ponto flutuante removeu o esconderijo para o custo da interpretação.

Apesar de o 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, por seu compilador Autocode para o computador Manchester Mark I. Glennie desenvolveu o compilador em Fort Halstead, no Royal Armaments Research Establishment, na Inglaterra. O compilador ficou operacional em setembro de 1952. Entretanto, de acordo com John Backus (Wexelblat, 1981, p. 26), o Autocode de Glennie era tão baixo nível e orientado à máquina que ele 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 logarítmico) e incluía vetores. O sistema foi implementado no computador Whirlwind do MIT, como um protótipo experimental, no verão de 1952, e em uma forma mais usável em maio de 1953. O tradutor gerava uma chamada a sub-rotina para codificar cada fórmula, ou expressão, no programa. A linguagem fonte era fácil de ler, e as instruções de máquina que foram incluídas eram aquelas para desvios. Apesar de esse trabalho preceder o trabalho no Fortran, ele nunca saiu do MIT.

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

2.3.2 Processo do projeto

Mesmo antes de o sistema 704 ser anunciado em maio de 1954, já haviam sido iniciados os planos para o 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 do Fortran, a 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 pseudo-código. Em outra rajada de otimismo, o documento afirmava que o 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 o Fortran foi desenvolvido 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; (4) devido ao alto custo dos computadores comparados com os dos programadores, a velocidade do código objeto gerado era o objetivo principal

dos primeiros compiladores Fortran. As características das primeiras versões do Fortran são diretamente oriundas desse ambiente.

2.3.3 Visão geral do Fortran I

O 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). O Fortran I incluía formatação de entrada e saída, nomes de variáveis até seis caracteres (eram apenas dois no Fortran 0), sub-rotinas definidas pelos usuários, apesar de elas não poderem ser compiladas separadamente, a sentença de seleção If e a sentença de repetição Do.

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

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

A afirmação mais audaciosa feita pelo grupo de desenvolvimento do Fortran durante o projeto da linguagem era que o código de máquina produzido pelo compilador teria cerca da metade da eficiência que poderia ser produzida à mão¹. Isso, mais do que qualquer coisa, fez os usuários potenciais ficarem céticos a seu respeito e frustrou muito do interesse no Fortran antes de seu lançamento. Para a surpresa de muitos, entretanto, o grupo de desenvolvimento do Fortran quase atingiu sua meta de eficiência. A grande parte do esforço de 18 trabalhadores-ano usada para construir o primeiro compilador foi gasta em otimização, e os resultados foram extraordinariamente efetivos.

O rápido sucesso do 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

¹ Na verdade, a equipe do Fortran acreditava que o código gerado pelo seu compilador não poderia ter menos do que a metade da rapidez com que o código de máquina escrito manualmente rodava, senão a linguagem não seria adotada pelos usuários.

significativos à linguagem, cujo mais importante foi a compilação independente de sub-rotinas. Sem a compilação independente, quaisquer mudanças em um programa requeriam que ele todo fosse recompilado. A falta de compilação independente no Fortran I, agregada à pobre confiabilidade do 704, colocou uma restrição prática no tamanho máximo dos programas de cerca de 300 a 400 linhas (Wexelblat, 1981, p. 68). Programas mais extensos têm uma pequena chance de serem compilados completamente antes da ocorrência de uma falha de máquina. A capacidade de incluir versões pré-compiladas em linguagem de máquina dos subprogramas diminuiu o processo de compilação consideravelmente.

2.3.5 Fortrancs IV, 77, 90, 95 e 2003

Um Fortran III foi desenvolvido, mas nunca amplamente distribuído. O Fortran IV, entretanto, tornou-se uma das linguagens de programação mais utilizadas de seu tempo. Ele evoluiu no período de 1960 até 1962 e foi padronizado como Fortran 66 (ANSI, 1966), apesar de esse nome ser raramente usado. O Fortran IV era uma melhoria ao Fortran II em muitos pontos. Dentre suas adições mais importantes estavam as declarações de tipo explícitas para variáveis, uma construção `If` lógica e a capacidade de passar subprogramas como parâmetros para outros subprogramas.

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

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

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

O 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 requeria 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 seu nome, de FORTRAN para Fortran, acompanhada pela mudança na convenção de usar letras maiúsculas para palavras-chave e para identificadores em programas Fortran. A nova convenção era que apenas a primeira letra das palavras-chave e dos identificadores deveria ter letra maiúscula.

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

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

2.3.6 Avaliação

A equipe de desenvolvimento original do 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 o Fortran seria usado em computadores não fabricados pela IBM. Na verdade, eles foram forçados a considerar a construção de compiladores Fortran para outras máquinas IBM apenas porque o sucessor do 704, o 709, foi anunciado antes que o compilador Fortran para o 704 fosse lançado. O efeito do Fortran no uso de computadores, com o 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 do Fortran I, e de todos os seus sucessores antes de 1990, que permite compiladores altamente otimizados é o fato de os tipos e o armazenamento para todas as variáveis serem fixados antes da execução. Nenhuma nova variável ou espaço pode ser alocado em tempo de execução. Esse é um sacrifício à flexibilidade em benefício da simplicidade e da eficiência, já que elimina a possibilidade de subprogramas recursivos e torna difícil a implementação de estruturas de dados que crescem ou mudam de forma dinamicamente. É claro, os tipos de programas construídos na época do desenvolvimento das primeiras versões do Fortran eram primariamente numéricos em sua natureza e simples comparados com os projetos de software recentes. Assim, o sacrifício não era muito grande.

O sucesso geral do Fortran é difícil de ser exagerado: ele mudou drasticamente a maneira como os computadores são usados. Isso, é claro, em grande parte por ter sido a primeira linguagem de alto nível muito usada. Comparativamente com conceitos e linguagens desenvolvidas depois, as primeiras versões do Fortran sofrem de uma variedade de maneiras, como o esperado, até porque não seria justo comparar o desempenho e o conforto de um Ford Modelo T 1910 com o desempenho e o conforto de um Ford Mustang 2009. Independentemente disso, apesar das inadequações do Fortran, o momento de alto investimento em software escrito na linguagem, dentre outros fatores, a mantiveram em uso por mais de meio século.

Alan Perlis, um dos projetistas do ALGOL 60, disse sobre o Fortran em 1978: “O 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 do Fortran 95
! Entrada: Um inteiro, List_Len, onde List_Len é menor do
!           que 100, seguido por valores inteiros List_Len
! Saída:   O número de valores de entrada que são maiores
!           do que a média de todos os valores de entrada
Implicit none
Integer Dimension(99) :: dInt_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 os dados de entrada em um vetor e calcula sua 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 que são maiores do 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 inventada para fornecer recursos 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 apareceu em meados dos anos 1950 em alguns lugares. Parte cresceu a partir 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. Ele foi publicado pela primeira vez em um artigo clássico que descreve um dos primeiros programas de IA, o Logical Theorist² (Teórico Lógico), e uma linguagem na qual ele poderia ser implementado (Newell e Simon, 1956), IPL-I (Information Processing Language I), que nunca foi posta em prática. A próxima versão, IPL-II, foi implementada em um computador Johnniac da Rand Corporation. 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 interpretador, no qual instruções de processamento de listas foram incluídas. 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 factível e útil.

A IBM começou a se interessar por IA no meio dos anos 1950 e escolheu a prova de teoremas como uma área de demonstração. Na época, o projeto do 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 ao Fortran, em vez de na forma de uma nova linguagem. Então, a Fortran List Processing Language (FLPL) foi projetada e implementada como uma extensão do Fortran. 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 uma 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. Dentre eles, estavam os métodos de controle de fluxo de funções matemáticas: recursão e expressões condicionais. A única linguagem de alto nível disponível na época, o Fortran I, não tinha nenhuma das duas.

Outro requisito que surgiu a partir da pesquisa sobre diferenciação simbólica foi a necessidade 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.

² O *Logical Theorist* descobriu provas para teoremas em cálculo proposicional.

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 era produzir um sistema para o processamento de listas. Ele seria usado inicialmente para implementar um programa proposto por McCarthy chamado de *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 seguinte, descreveremos o desenvolvimento do LISP puro.

2.4.3 Visão geral da linguagem

2.4.3.1 Estruturas de dados

O LISP puro 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 em 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 listas aninhadas também são especificadas com parênteses. Por exemplo, a lista

(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 seu segundo elemento a sublista (F G).

Internamente, as listas são armazenadas como estruturas de listas simplesmente encadeadas, nas quais cada nó tem dois ponteiros que apontam para alguma representação do átomo, como seu valor simbólico ou numérico, ou o ponteiro para uma sublista. Um nó para um elemento que é uma 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

³ O *Advice Taker* representava a informação com sentenças escritas em uma linguagem formal e usava um processo de inferência lógica para decidir o que fazer.

elemento da lista. Uma lista é referenciada por um ponteiro para 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 sucesso, então sua ligação é NIL, representado na Figura 2.2 como uma linha diagonal no elemento. As 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 de programar em uma linguagem imperativa.

2.4.3.3 A sintaxe de LISP

LISP é muito diferente das linguagens imperativas, tanto porque é funcional quanto porque a aparência dos programas LISP 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

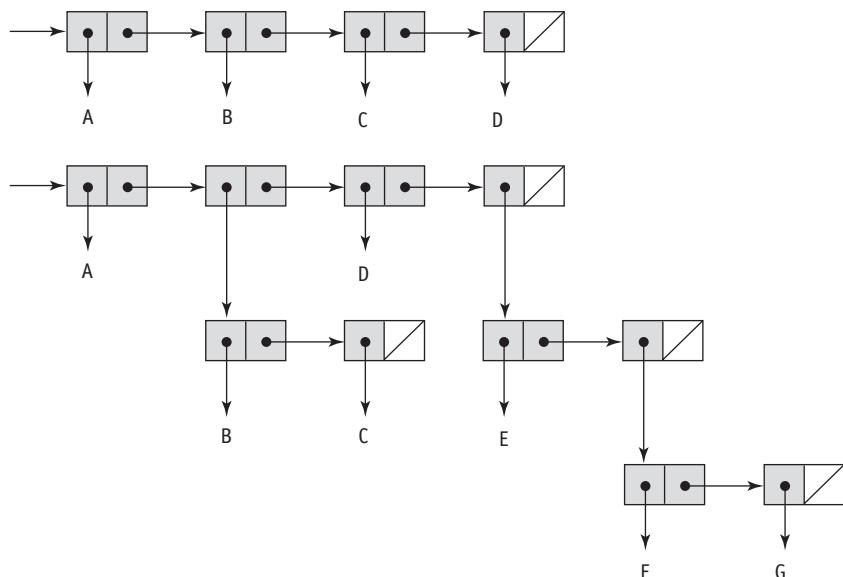


Figura 2.2 Representação interna de duas listas em LISP.

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 reputação de LISP de ser altamente ineficiente foram eliminadas. Muitas das implementações contemporâneas são compiladas, e o código resultante é muito mais rápido do que rodar o código fonte em um interpretador. Além do seu sucesso em IA, LISP foi pioneira na programação funcional, que se provou uma área extremamente ativa na pesquisa em linguagens de programação. Conforme mencionado no Capítulo 1, muitos pesquisadores de linguagens de acreditam que a programação funcional é uma abordagem muito melhor para o desenvolvimento de software do que a programação procedural usando 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 (Dybvig, 2003). 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 pequena linguagem com uma sintaxe e semântica simples, Scheme é bastante adequada para aplicações educacionais, como cursos sobre programação funcional e introduções gerais à programação. Scheme 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 familiar problema da portabilidade. COMMON LISP (Graham, 1996) foi criada em um esforço para corrigir essa situação, projetada para combinar os recursos de diferentes dialetos de LISP desenvolvidos nos anos 1980, incluindo Scheme, em uma única linguagem. Sendo um amálgama, COMMON LISP é uma linguagem grande e complexa. Sua base, entretanto, é o LISP puro, então sua sintaxe, funções primitivas e a 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 declarar uma como **special**, ela se torna uma variável com escopo dinâmico.

COMMON LISP tem um grande número de tipos e estruturas de dados, incluindo registros, vetores, números complexos e cadeias de caracteres. Ela também tem uma forma de pacotes para modular coleções de funções e dados que fornece controle de acesso.

COMMON LISP é descrita em mais detalhes no Capítulo 15.

2.4.6 Linguagens relacionadas

ML (MetaLanguage; 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.

Diferentemente de LISP e de Scheme, o tipo de cada variável e de cada expressão em ML pode ser determinado em tempo de compilação⁴. Os tipos são associados com objetos em vez de nomes. Os tipos das expressões são inferidos do contexto da expressão, conforme discutido no Capítulo 5. 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 aquela das linguagens imperativas, como Java e C++.

⁴ Embora isso seja verdade, as funções ML podem ser genéricas, e isso significa que os tipos dos parâmetros e tipos de retorno podem ser diferentes para diferentes chamadas.

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), por sua vez, é baseada em Miranda e, assim como ela, é uma linguagem puramente funcional, não tendo variáveis nem sentenças de atribuição. Outra característica que distingue Haskell é seu uso de avaliação tardia (*lazy evaluation*), ou seja, nenhuma expressão é avaliada até seu valor ser necessário, levando a algumas capacidades surpreendentes na linguagem.

Tanto ML quanto Haskell são brevemente discutidas no Capítulo 15.

2.5 O PRIMEIRO PASSO EM DIREÇÃO À SOFISTICAÇÃO: ALGOL 60

O ALGOL 60 tem tido uma grande influência nas linguagens de programação subsequentes e é de uma importância central em qualquer estudo histórico de linguagens.

2.5.1 Perspectiva histórica

O ALGOL 60 foi o resultado de esforços para projetar uma linguagem universal para aplicações científicas. No final de 1954, o sistema algébrico de Laning e Zierler estava em operação por mais de um ano, e o primeiro relatório sobre o Fortran havia sido publicado. O Fortran se tornou uma realidade em 1957, e diversas outras linguagens de alto nível haviam sido desenvolvidas. Dentre 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 a comunicação 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 essa proliferação de linguagens, 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), submeteu uma petição à ACM (Association for Computing Machinery) em 10 de maio de 1957, para formar um comitê para estudar e recomendar ações para uma linguagem de programação científica independente de linguagem de programação. Apesar de o Fortran poder ser tal candidato, ele não se tornaria uma linguagem universal, já que era de propriedade única da IBM.

Previvamente, 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 próxima 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.
- 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 os objetivos levantados, mas o processo do projeto necessitava de inúmeros comprometimentos, 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 do ALGOL 58

A linguagem projetada na reunião em Zurique foi nomeada de Linguagem Algorítmica Internacional (IAL – International Algorithmic Language). Foi sugerido durante o projeto que ela se chamassem ALGOL, do inglês ALGOrithmic Language, mas o nome foi rejeitado porque ele 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, o ALGOL 58 era um descendente do Fortran, o que é bastante natural. Ele generalizou muitos dos recursos do Fortran e adicionou novas construções e conceitos. Algumas das generalizações eram relativas ao objetivo de não amarrar 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.

O 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 subsequente incorporou. Alguns dos recursos do Fortran que foram generalizados são: os identificadores podem ter qualquer tamanho, em oposição à restrição do Fortran I de nomes de identificadores com até seis caracteres; qualquer número de dimensões de um vetor era permitido, diferentemente da limitação do Fortran I de até três dimensões; o limite inferior dos vetores podia ser especificado pelo programador, enquanto no Fortran ele era implicitamente igual a 1; sentenças de seleção aninhadas eram permitidas, o que não era o caso no Fortran I.

O ALGOL 58 adquiriu o operador de atribuição de uma maneira um tanto usual. Zuse usava o formato

expressão => variável

para a sentença de atribuição em Plankalkül. Apesar de Plankalkül não ter sido ainda publicada, alguns dos membros europeus do comitê do ALGOL 58 estavam familiarizados com a linguagem. O comitê se interessou com a forma da atribuição em Plankalkül, mas devido aos argumentos sobre limitações do conjunto de caracteres, o símbolo maior foi trocado pelo sinal de dois pontos. Então, em grande parte por causa da insistência dos americanos, toda a sentença foi modificada para ser equivalente ao formato do Fortran

variável := expressão

Os europeus preferiam a forma oposta, mas isso seria o inverso do Fortran.

2.5.4 Recepção do relatório do ALGOL 58

A publicação do relatório do ALGOL 58 (Perlis e Samelson, 1958), em dezembro de 1958, foi recebida com uma boa dose de entusiasmo. Nos Estados Unidos, a nova linguagem foi vista mais como uma coleção de ideias para o projeto de linguagens de programação do que uma linguagem padrão universal. Na verdade, o relatório do 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 no ALGOL 58 a atingir um amplo uso. (Jules era Jules I. Schwartz, um dos projetistas de JOVIAL). JOVIAL se

tornou bastante usada porque foi a linguagem científica oficial para a Força Aérea Americana por um quarto de século.

O resto da comunidade da computação nos Estados Unidos não estava 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 o ALGOL 58. A IBM começou 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 o ALGOL 58 e que a IBM o 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 o Fortran, já haviam tido sofrimento 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. Na metade de 1959, tanto a IBM quanto o SHARE haviam desenvolvido um interesse próprio pelo Fortran, levando a decisão de mantê-lo como a linguagem científica para as máquinas IBM da série 700, abandonando o ALGOL 58.

2.5.5 O processo do projeto do ALGOL 60

Durante 1959, o ALGOL 58 foi debatido intensamente 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 (do inglês – *Backus-Naur Form*). BNF é descrita em detalhes no Capítulo 3.

Em janeiro de 1960, a segunda reunião do ALGOL foi realizada, dessa vez em Paris, com o objetivo de debater as 80 sugestões submetidas para consideração. Peter Naur da Dinamarca havia se envolvido enormemente no desenvolvimento do ALGOL, apesar de não ser um membro do grupo de Zurique. Foi Naur que criou e publicou o ALGOL Bulletin. Ele gastou bastante de tempo estudando o artigo de Backus que introduzia a BNF e decidiu que ela deveria ser usada para descrever formalmente os resultados da reunião de 1960. Após ter feito algumas mudanças relativamente pequenas à 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 do ALGOL 60

Apesar de a reunião de 1960 ter durado apenas seis dias, as modificações feitas no ALGOL 58 foram drásticas. Dentre os mais importantes avanços, estavam:

- O conceito de estrutura de bloco foi introduzido, o que permitia ao programador localizar partes do programa introduzindo novos ambientes ou escopos de dados.
- Duas formas diferentes de passagem de parâmetros a subprogramas foram permitidas: por valor e por nome.
- Foi permitido aos procedimentos serem recursivos. A descrição do ALGOL 58 não era clara em relação a essa questão. Note que, apesar de essa recursão ser nova para as linguagens imperativas, LISP já fornecia funções recursivas em 1959.
- Vetores dinâmicos na pilha eram permitidos. Um vetor dinâmico na pilha é um no 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, mas 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 do 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, o ALGOL 60 foi um grande sucesso; de outras, um imenso fracasso. Foi bem-sucedido em se tornar, quase imediatamente, a única maneira formal aceitável de comunicar algoritmos na literatura em 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++ e Java.

O esforço de projeto do ALGOL 58/ALGOL 60 incluiu uma longa lista de primeiras vezes. Foi a primeira vez que um grupo internacional tentou projetar uma linguagem de programação, a primeira linguagem projetada para ser independente de máquina e também a primeira cuja sintaxe foi formalmente descrita. 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 do ALGOL 60 afetou as arquiteturas de máquina. No efeito mais contundente disso, uma extensão da linguagem foi usada como a lingua-

gem de sistema de uma série de computadores de grande escala, as máquinas Burroughs 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, o 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 sua falta de aceitação. Por um lado, alguns dos recursos se mostraram muito flexíveis – 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 o 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 era outra razão para sua falta de aceitação. A entrada e saída dependente de implementação fez os programas terem uma portabilidade ruim para outros computadores.

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

Finalmente, apesar de haver muitos outros problemas, o forte estabelecimento do Fortran entre os usuários e a falta de suporte da IBM foram provavelmente os fatores mais importantes na falha do ALGOL 60 em ter seu uso disseminado.

O esforço do 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 do ALGOL 60
Entrada: Um inteiro, listlen, onde listlen é menor do que
          100, seguido por valores inteiros listlen
Saída:   O número de valores de entrada que são maiores do
          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) and (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]);
      end
    end
  end
```

```
sum := sum + intlist[counter]
end;
comment Calcula a média;
average := sum / listlen;
comment Conta os valores que são 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 INFORMATIZANDO OS REGISTROS COMERCIAIS: COBOL

A história do COBOL é, de certa forma, o oposto da do ALGOL 60. Apesar de ter sido mais usado do que qualquer outra linguagem de programação, ele teve pouco efeito no projeto de linguagens subsequentes, exceto por PL/I. O COBOL pode ainda ser a linguagem mais usada⁵, apesar de ser difícil ter certeza disso. Talvez a razão mais importante pela qual o COBOL tem uma influência pequena é que poucos tentaram projetar uma nova linguagem de negócios desde sua aparição. Isso ocorreu em parte por causa do quão bem as capacidades do COBOL satisfazem as necessidades de sua área de aplicação. Outra razão é que uma grande parcela da computação em negócios nos últimos 30 anos ocorreu em pequenas empresas. E nelas, pouco desenvolvimento de software ocorre. Em vez disso, muito do software usado é comprado como pacotes de prateleira para várias aplicações gerais.

2.6.1 Perspectiva histórica

O início do COBOL é, de certa forma, similar ao do ALGOL 60. A linguagem também foi projetada por um comitê de pessoas que se reuniam em períodos relativamente curtos. O estado da computação de negócios na época, no caso 1959, era similar ao estado da computação científica quando o Fortran estava sendo projetado. 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 linguagem, a AIMACO, estava sendo usada pela Força Aérea americana, mas era apenas uma pequena variação do FLOW-MATIC.

⁵ No final dos anos 1990, em um estudo associado ao problema do ano 2000, foi estimado que existiam aproximadamente 800 milhões de linhas de código COBOL em uso em uma área de 35 quilômetros quadrados de Manhattan.

A IBM havia projetado uma linguagem de programação para aplicações comerciais, chamada COMTRAN (COMmercial TRANslator), mas que nunca foi implementada. Diversos outros projetos de linguagens haviam sido planejados.

2.6.2 FLOW-MATIC

Vale a pena discutir brevemente as origens do FLOW-MATIC, porque ele foi o principal progenitor do COBOL. Em dezembro de 1953, Grace Hopper na Remington-Rand UNIVAC escreveu uma proposta profética. Ela sugeria que “programas matemáticos devem ser escritos em notação matemática, programas de processamento de dados devem ser escritos em sentenças em inglês” (Wexelbal, 1981, p. 16). Infelizmente, era impossível em 1953 convencer não programadores de que um computador poderia ser feito para entender palavras em inglês. Somente em 1955 uma proposta similar teve alguma esperança de ser patrocinada pela UNIVAC, e mesmo assim precisou de um protótipo para convencer a gerência. Parte do processo de venda envolvia compilar e rodar 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 um dos fatores principais para a aceitação da proposta de Hopper.

2.6.3 O processo do projeto do COBOL

A primeira reunião formal sobre o assunto de 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 do ALGOL em Zurique). O consenso do grupo era que a linguagem, na época chamada CBL (de Common Business Language), deveria ter certas características gerais. A maioria concordou que ela deveria usar 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 lerem os programas. Finalmente, o projeto não deveria ser restrinido 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 já estava sendo feito para criar novas 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 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, o Short Range Committee foi formado.

Existiram decisões iniciais para separar as sentenças da linguagem em duas categorias – descrições de dados e operações executáveis – e que as sentenças dessas duas categorias residiriam 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 trabalharem em processamento de dados, as quais se pensava que não ficariam confortáveis com uma notação matemática. Argumentos similares eram feitos sobre incluir ou não expressões aritméticas. O relatório final do Short Range Committee, completado em dezembro de 1959, descrevia a linguagem que mais tarde foi chamada de COBOL 60.

As especificações de linguagem do COBOL 60, publicadas pela Agência de Impressão do Governo americano (Government Printing Office) em abril de 1960 (Department of Defense, 1960), foram descritas como “iniciais”. 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` do 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 um modo geral, a divisão de dados (*data division*) é a parte forte do projeto do 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 o COBOL ideal para imprimir relatórios contábeis. Talvez a fraqueza mais importante da divisão de procedimentos original do COBOL tenha sido sua falta de funções. Versões do COBOL anteriores ao padrão 1974 também não permitiam subprogramas com parâmetros.

Nosso comentário final sobre o COBOL: foi a primeira linguagem de programação cujo uso foi obrigatório pelo Departamento de Defesa Americano (DoD). Essa obrigatoriedade veio após o seu desenvolvimento inicial, já que o COBOL não foi especificamente projetado para o DoD. Apesar de seus méritos, o COBOL provavelmente não teria sobrevivido sem essa obrigatoriedade. O desempenho ruim dos primeiros compiladores sim-

plesmente o tornava muito caro. Eventualmente, é claro, os compiladores se tornaram mais eficientes e os computadores ficaram mais rápidos e baratos e tinham memórias muito maiores. Juntos, esses fatores permitiram o sucesso do COBOL, dentro e fora do DoD. Sua aparição levou à mecanização eletrônica da contabilidade, uma revolução importante por qualquer medida.

A seguir, temos exemplo de programa em COBOL. Ele lê um arquivo chamado BAL-FWD-FILE, que contém informação de inventário sobre certa coleção de itens. Dentre outras coisas, cada registro inclui o número atual de itens (BAL-ON-HAND) e o ponto de novo pedido (BAL-REORDER-POINT). O ponto de novo pedido 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 um amplo uso, mas pouco respeito. Como o COBOL, foi ignorada pelos cientistas da computação. Além disso, como o COBOL, em suas primeiras versões a linguagem era deselegante e incluía apenas um conjunto pobre de sentenças de controle.

O BASIC era muito popular em microcomputadores no final dos anos 1970 e no início dos anos 1980, basicamente por causa de duas das suas principais características. Ele é fácil de aprender, especialmente para aqueles que não são orientados à ciência, e seus dialetos menores podem ser implementados em computadores com memórias muito pequenas⁶. Quando a capacidade dos microcomputadores cresceu e outras linguagens foram implementadas, o uso do BASIC decaiu. Uma forte revitalização no uso do BASIC surgiu com a aparição do Visual Basic (Microsoft, 1991) no início dos anos 1990.

2.7.1 Processo do projeto

O 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 do Fortran e do ALGOL 60. Seus estudantes de ciências básicas tinham pouco trabalho 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 uma linguagem especialmente voltada para os estudantes da área de humanas seria projetada. Essa nova linguagem usaria terminais como o método de acesso a computadores. Os objetivos do sistema eram:

1. Deve ser fácil para estudantes que não são de ciências básicas a aprendem e usarem.
2. Deve ser prazerosa e amigável.
3. Deve agilizar os deveres de casa.
4. Deve permitir acesso livre e privado.
5. Deve considerar o tempo do usuário mais importante do 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.

⁶ Alguns dos primeiros microcomputadores incluíam interpretadores BASIC que residiam em 4096 bytes de ROM.

A combinação do segundo, terceiro e quarto objetivos levaram ao aspecto de compartilhamento de tempo do 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 do BASIC, usando acesso remoto a um computador GE 225. O projeto e a codificação do sistema operacional para o BASIC começaram no final de 1963. As 4 horas da manhã de 1º de maio de 1964, o primeiro programa usando o BASIC com compartilhamento de tempo foi digitado e executado. Em junho, o número de terminais no sistema cresceu para 11 – e, no fim do ano, para 20.

2.7.2 Visão geral da linguagem

A versão original do BASIC era bastante pequena e não era interativa: não existiam maneiras de um programa executável obter dados de entrada de um usuário. Os programas eram digitados, compilados e executados de uma maneira um tanto quanto orientada a lotes. O 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 apreciar a diferença entre tipos inteiros e de ponto flutuante, o tipo foi chamado de “*numbers*” (números). De um modo geral, era uma linguagem muito limitada, apesar de bastante fácil de aprender.

2.7.3 Avaliação

O aspecto mais importante do BASIC original foi ser a primeira linguagem amplamente usada por meio de terminais conectados a um terminal remoto⁷. Os terminais haviam recém-começado a ficar disponíveis na época. Antes disso, a maioria dos programas eram inseridos nos computadores ou por meio de cartões perfurados ou por fitas de papel.

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

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

O BASIC foi criticado pela estrutura pobre dos programas escritos na linguagem, dentre outras coisas. Pelo critério de avaliação discutido no

⁷ LISP inicialmente era empregada em terminais, mas não era usada amplamente no início dos anos 1960.

Capítulo 1, mais especificamente em relação à legibilidade e à confiabilidade, a linguagem de fato é muito pobre. As primeiras versões da linguagem 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 do BASIC nos anos 1990 foi provocado pelo surgimento do Visual BASIC (VB), que se tornou amplamente usado em grande parte porque fornece uma maneira simples de construir interfaces gráficas para interação com o usuário (GUIs). O Visual Basic .NET, ou apenas VB.NET, é uma das linguagens .NET da Microsoft. Apesar de ter se distanciado do VB, ela rapidamente substituiu a linguagem mais antiga. Talvez a diferença mais importante entre o VB e o VB.NET é que o segundo suporta completamente a programação orientada a objetos. Pensava-se que os usuários do VB migrariam para uma linguagem diferente, como o C# (veja a Seção 2.19), em vez de aprender VB.NET, principalmente porque todas as linguagens .NET têm acesso às ferramentas de construção de interfaces gráficas do VB. Entretanto, isso não ocorreu – o VB.NET é mais usado do que o C#.

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

```

REM Programa de exemplo do BASIC
REM Entrada: Um inteiro, listlen, onde listlen é menor do
REM             que 100, seguido por valores inteiros listlen
REM Saída:     O número de valores de entrada que são maiores
REM             do 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 que são maiores do 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

```



Projeto de usuário e projeto de linguagens

ALAN COOPER

Autor do best-seller *About Face: The Essentials of User Interface Design*, Alan Cooper tem ampla experiência em projetar o 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 um curso tecnólogo* em programação em uma faculdade comunitária da Califórnia. Meu primeiro emprego foi como programador na America President Lines em São Francisco (uma das empresas mais antigas de transporte oceânico dos Estados Unidos). 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 projeto de interação.

Você é muito conhecido nas áreas de projeto de linguagem e de interface com o 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 AQUELE LANÇAMENTO INICIAL DO WINDOWS

Nos anos 1980, você começou a usar o Windows e ficou entusiasmado com suas qualidades: o suporte para a interface gráfica com o 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 da Microsoft do suporte para multitarefas no Windows de forma prática, que incluía realocação dinâmica e comunicação interprocessos. 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 interpretador de comandos melhor que batizei de Tripod. O interpretador de comandos original da Microsoft, o MSDOS.exe, era um dos principais elementos bloqueadores para o sucesso inicial do Windows. O Tripod tentava resolver o problema sendo mais fácil de usar e configurar.

Quando você teve essa ideia? Ela não ocorreu até o fim de 1987, quando eu estava entrevistando um cliente corporativo, e 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 faixa 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 próprio 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 seu próprio interpretador de comandos ideal e personalizado. Assim, um programador poderia criar um interpretador de comandos poderoso e com uma ampla faixa de ação, 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.

* N. de T.: Do inglês *associate degree*.

Como você foi de escritor de um interpretador de comandos a colaborados 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. A Microsoft pegou a versão de distribuição do Ruby e adicionou Quick-BASIC a ela, criando o VB. Todas aquelas inovações originais estavam no Tripod/Ruby.

RUBY COMO INCUBADORA PARA O VISUAL BASIC

Fale sobre seu interesse nas primeiras versões do Windows e o tal recurso chamado DLL. As DLLs não eram uma coisa, eram um recurso do sistema operacional. Elas permitiam que um programador construisse 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 em-

“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 interpretador de comandos melhor.”

presa qualquer, instalar em seu computador, e ter o controle de grade como se fosse parte da linguagem, incluindo o *front-end* visual de programação.

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 o candidato mais mencionado, mas, a Microsoft tirou vantagem dessa interface dinâmica para substituir nossa pequena linguagem de interpretação de comandos pelo Quick-BASIC. Esse novo casamento de linguagem com um visual *front-end* era estático e permanente, e apesar da 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? As pontes não são construídas por engenheiros, mas por metalúrgicos. De modo similar, os programas de software não são construídos por engenheiros, mas por programadores.

2.8 TUDO PARA TODOS: PL/I

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

2.8.1 Perspectiva histórica

Como o Fortran, PL/I foi desenvolvida como um produto da IBM. No início dos anos 1960, os usuários de computadores na indústria haviam se instalado 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 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 recursos elaborados e eficientes para entrada e saída. Elas usavam COBOL, apesar de no início de 1963, quando a história da PL/I começou, a conversão de linguagem de montagem para COBOL não estar completa. 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 requeriam 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 informação de gerenciamento, os quais requeriam dados de ponto flutuante e vetores. Parecia que as instalações de computação logo precisariam de duas equipes técnicas e de computadores diferentes, de forma a oferecer suporte para duas linguagens de programação bastante diferentes⁸.

Essas percepções levaram de forma bastante natural ao conceito de projetar um computador universal que deveria ser 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 poderia ser usada tanto para

⁸ Na época, grandes instalações de computação requeriam equipes de manutenção em tempo integral tanto para hardware quanto para software de sistema.

aplicações comerciais quanto para aplicações científicas. Para atender tais aplicações, recursos para suportar a programação de sistemas e para o processamento de listas foram adicionados. Assim, a nova linguagem viria para substituir o Fortran, o COBOL, o LISP e as aplicações de sistemas das linguagens de montagem.

2.8.2 O 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 chamado de Comitê 3·3, nomeado assim porque era formado por três membros da IBM e três do SHARE. O Comitê 3·3 se encontrava três ou quatro dias, uma semana sim, uma não, para projetar a linguagem.

Como ocorreu com o Comitê Short Range para o COBOL, o projeto inicial foi planejado para ser completado em um tempo excepcionalmente curto. Aparentemente, independentemente do escopo 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 completa em dezembro, menos de três meses após a formação do comitê. Ele obteve extensões de prazo em duas ocasiões, movendo a data de finalização para janeiro e, posteriormente, para o final de fevereiro de 1964.

O conceito inicial de projeto era que a nova linguagem seria uma extensão do 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 – Nova Linguagem de Programação). 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 na 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 do 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 é agora uma linguagem quase morta, não tentaremos, mesmo que de maneira abreviada, discutir todos os recursos da linguagem ou mesmo suas construções mais controversas. Em vez disso, iremos mencionar algumas das contribuições aos conhecimentos acerca de linguagens de programação.

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

- Era permitido aos programas criar subprogramas executados concorrentemente.
- 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.

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 existia pouca experiência em projeto de linguagens na época. De um modo geral, o projeto da PL/I era baseado na premissa de que qualquer construção útil e que poderia ser implementada deveria ser incluída, com poucas preocupações sobre como um programador 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 ver 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 do problema da complexidade por conta de seu tamanho, PL/I sofria também pelo fato de ter diversas construções que atualmente são consideradas pobramente projetadas. Dentre essas estavam os ponteiros, o tratamento de exceções e concorrência, apesar de que devemos dizer que em cada um desses casos as construções não haviam aparecido em nenhuma linguagem anterior.

Em termos de uso, a PL/I deve ser considerada ao menos parcialmente bem-sucedida. Nos anos 1970, ela desfrutou de um uso significativo tanto em aplicações comerciais quanto científicas. Ela também foi bastante usada como um veículo 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 DO QUE
          100, SEGUIDO POR VALORES INTEIROS LISTLEN
SAÍDA:    UM INTEIRO, LISTLEN, ONDE LISTLEN É MENOR DO QUE
          100, SEGUIDO POR VALORES INTEIROS LISTLEN */
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 deste capítulo, porque as linguagens discutidas aqui são bastante diferentes. Nem APL nem SNOBOL foram baseadas em linguagens prévias e nenhuma delas teve muita influência em linguagens utilizadas posteriormente⁹. Alguns dos recursos interessantes de APL são discutidos mais adiante.

⁹ Entretanto, elas tiveram alguma influência em algumas linguagens que não são muito usadas (J é baseada em APL, ICON é baseada em SNOBOL, e AWK é parcialmente baseada em SNOBOL).

Tanto em relação à aparência quanto ao propósito, APL e SNOBOL são muito diferentes. Elas compartilham, entretanto, duas características fundamentais: tipagem dinâmica e alocação dinâmica de armazenamento. Variáveis em ambas as linguagens 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 da APL

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

APL tem vários operadores poderosos, os quais criaram um problema para os implementadores. As primeiras formas de uso da APL foram por meio de terminais de impressão IBM. Tais terminais tinham elementos de impressão especiais que forneciam o estranho conjunto de caracteres requerido 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 fornece uma alta expressividade, mas também faz os programas APL serem de difícil leitura. Isso levou as pessoas a pensarem em APL como uma linguagem mais bem usada para programação “descartável”. Apesar de os programas serem escritos rapidamente, eles poderiam ser descartados após seu uso porque seriam de difícil manutenção.

APL está ativa por cerca de 45 anos e ainda é usada, apesar de não amplamente, e não mudou muito em todo esse tempo de vida.

2.9.2 Origens e características do 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), especificamente para o processamento de texto. O coração do 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 do que linguagens alternativas, ela não é mais usada para tais programas. Entretanto, ainda é uma linguagem viva e com suporte usado para uma variedade de tarefas de processamento de textos em diferentes áreas de aplicação.

2.10 O INÍCIO DA ABSTRAÇÃO DE DADOS: SIMULA 67

Apesar de o 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 o 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). Eles estavam inicialmente interessados em usar computadores para simulação, mas também trabalhavam em pesquisa operacional. SIMULA I foi projetado exclusivamente para a simulação de sistemas e implementado pela primeira vez no final de 1964 em um computador UNIVAC 1107.

Quando a implementação do SIMULA I estava completa, Nygaard e Dahl começaram os esforços para estender a linguagem com novos recursos e modificações de algumas das construções existentes para torná-la útil para aplicações de propósito geral. O resultado desse trabalho era o SIMULA 67, cujo projeto foi apresentado publicamente pela primeira vez em março de 1967 (Dahl e Nygaard, 1967). Discutiremos apenas o SIMULA 67, apesar de alguns de seus recursos interessantes também estarem disponíveis no SIMULA I.

2.10.2 Visão geral da linguagem

O SIMULA 67 é uma extensão do ALGOL 60, com sua estrutura de bloco e suas sentenças de controle. A principal deficiência do 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 chamados de **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 fornecer suporte às corrotinas no SIMULA 67, a construção “classe” foi desenvolvida, determinando um avanço importante já que foi como começou o conceito de abstração de dados. A definição de uma estrutura de dados e as rotinas que manipulam suas instâncias são empacotadas em uma mesma unidade gerando a ideia básica de uma classe. Além disso, uma definição de classe é apenas um modelo para uma estrutura de dados e, dessa forma, é distinta de uma instância de classe, de maneira que um programa pode criar e usar qualquer número de instâncias de uma classe específica. Instâncias de classes podem conter dados locais e incluir código, executado em tempo de criação, podendo inicializar alguma estrutura de dados da instância de classe.

Uma discussão mais aprofundada de classes e instâncias de classes é apresentada no Capítulo 11. É interessante notar que o importante conceito de abstração de dados não foi desenvolvido e atribuído à construção classe até 1972, quando Hoare (1972) reconheceu a conexão.

2.11 PROJETO ORTOGONAL: ALGOL 68

O ALGOL 68 foi fonte de diversas ideias novas no projeto de linguagens, algumas adotadas por outras linguagens. O incluímos 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 apareceu em 1962, apesar de demorar seis anos até que a próxima iteração de projeto tenha sido publicada. A linguagem resultante, o ALGOL 68 (van Wijngaarden et al., 1969) era drasticamente diferente de seus predecessores.

Uma das inovações mais interessantes do ALGOL 68 foi em relação a um de seus critérios de projeto primários: ortogonalidade. O uso da ortogonalidade resultou em diversos recursos inovadores no ALGOL 68, um deles descrito na seção seguinte.

2.11.2 Visão geral da linguagem

Um resultado importante da ortogonalidade no ALGOL 68 foi a inclusão dos tipos de dados definidos pelo usuário. Linguagens anteriores, como o 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 obviamente ela não poderia fornecer uma estrutura de dados adequada para cada necessidade.

A abordagem do ALGOL 68 para as estruturas de dados era fornecer alguns tipos primitivos e permitir que o usuário os combinasse em 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 que ele projete abstrações de dados que se encaixem com os problemas em particular de uma maneira muito forte. Todos os aspectos de tipos de dados são discutidos no Capítulo 6.

O ALGOL 68 também introduziu um tipo de vetores dinâmicos que serão chamados de “implícitos dinâmicos do monte” no Capítulo 5. Um vetor dinâmico é um 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

O ALGOL 68 inclui um número significativo de recursos. Seu uso da ortogonalidade, o qual alguns podem argumentar que era demais, foi revolucionário.

Entretanto, o ALGOL 68 repetiu uma das sinas do ALGOL 60, um importante fator para sua popularidade limitada: a linguagem era 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), teria de aprender a nova metalinguagem, chamada de gramáticas de van Wijngaarden. 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 procedimentos 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 do ALGOL 68. O ALGOL 68 atingiu 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 atingiu uma boa facilidade de escrita com a inclusão de um grande número de construções fixas. O ALGOL 68 estendeu a simplicidade elegante do ALGOL 60, enquanto PL/I simplesmente atraiu os recursos de diversas linguagens em um mesmo recipiente para atingir seus objetivos. É claro, precisamos manter em mente que o objetivo da linguagem PL/I era fornecer uma ferramenta unificada para uma ampla classe de problemas; por outro lado, o ALGOL 68 era focado em uma classe: aplicações científicas.

PL/I atingiu uma aceitação muito maior do que o ALGOL 68, principalmente pelos esforços promocionais da IBM e pelos problemas de entendimento e de implementação do 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. O ALGOL 68 não desfrutava de tal benfeitor.

2.12 ALGUNS DOS PRIMEIROS DESCENDENTES DOS ALGOLS

Todas as linguagens imperativas, incluindo aquelas que oferecem suporte à programação orientada a objetos, devem algo de seu projeto ao ALGOL 60 e/ou ao 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 con-

tinuar o desenvolvimento do 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 ao ALGOL 60 (Wirth e Hoare, 1966). A maioria do grupo rejeitou a proposta como um avanço muito pequeno em relação ao ALGOL 60. Em vez disso, uma revisão muito mais complexa foi realizada – o que, no fim das contas, tornou-se o ALGOL 68. Wirth, com membros desse grupo, não acreditava que o relatório do ALGOL 68 deveria ter sido publicado, devido à complexidade tanto da linguagem quanto da metalinguagem usada para descrevê-la. Tal posicionamento se provou válido, já que os documentos do ALGOL 68 (e dessa forma a linguagem) foram tidos como desafiadores para a comunidade de computação.

A versão de Wirth e Hoare do ALGOL 60 foi nomeada ALGOL-W. Ela foi implementada na Universidade de Stanford e usada basicamente como um veículo educacional, mas apenas em algumas universidades. As principais contribuições do ALGOL-W eram 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 do ALGOL 60. Ambos são discutidos no Capítulo 9. A sentença **case** é discutida no Capítulo 8.

O próximo grande esforço de projeto de Wirth, mais uma vez baseado no ALGOL 60, foi seu mais bem-sucedido: Pascal¹⁰. A definição do 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 ao Pascal vieram de linguagens anteriores. Por exemplo, os tipos de dados definidos pelo usuário foram introduzidos no ALGOL 68, a sentença **case** no ALGOL-W, e os registros do Pascal são similares àqueles do COBOL e da linguagem PL/I.

2.12.1.2 Avaliação

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

Como o Pascal foi projetado como uma linguagem de ensino, ele não tinha diversos recursos essenciais para muitos tipos de aplicação. O melhor

¹⁰ Pascal foi nomeada em homenagem a Blaise Pascal, um filósofo e matemático francês do século XVII que inventou a primeira máquina mecânica de adição em 1642 (dentre outras coisas).

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 do Pascal, tanto para o ensino de programação quanto para outras aplicações, era baseada em sua excepcional combinação de simplicidade e expressividade. Apesar de existirem algumas inseguranças na linguagem, o Pascal é, mesmo assim, relativamente seguro, especialmente quando comparado com Fortran ou C. Em meado dos anos 1990, a popularidade do Pascal estava 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 no Pascal.

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

```
{Programa de exemplo em Pascal
Entrada: Um inteiro, listlen, onde listlen é menor do que
          100, seguido por valores inteiros listlen
Saída:   O número de valores de entrada que são maiores do
          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 que são maiores do 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 previamente, mas tem sido muito usada por um longo período de 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 do 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 desenvolvida 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) e a primeira versão foi escrita em linguagem de montagem. A primeira linguagem de alto nível implementada no UNIX foi B, que era baseada no 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, que levam 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 de pontos. Variáveis que eram usadas como operandos não precedidas de pontos eram consideradas inteiras. Uma alternativa a isso seria o uso de símbolos diferentes para os operadores de ponto flutuante.

Esse problema, com diversos outros, levou ao desenvolvimento de uma nova linguagem tipada baseada em B. Originalmente chamada de NB, mas posteriormente nomeada de 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 pelo 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 era 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, 1989), que incluía muitos dos recursos que os implementadores haviam incorporado na linguagem. Esse

¹¹ Essa linguagem é geralmente chamada de “K&R C”.

padrão foi atualizado em 1999 (ISO, 1999), com mudanças significativas à linguagem. A versão 1989, chamada por muito tempo de ANSI C, passou a se chamar C89. Iremos nos referir à versão de 1999 como C99.

2.12.2.2 Avaliação

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 um 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 os 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 seu grande aumento em 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 do que
          100, seguido por valores inteiros listlen
Saída:   O número de valores de entrada que são maiores do
          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 que são maiores do 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

Explicando de uma forma simples, a programação lógica é o uso de uma notação lógica formal para comunicar processos computacionais para 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, a resolução de nomes, desenvolvida 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 do 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 *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 *framework* simples para discutir a linguagem.

A base de dados de um programa Prolog consiste em dois tipos de sentenças: fatos e regras. Exemplos de sentenças factuais são:

```
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 uma regra é

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

A regra afirma que se pode deduzir que x é o avô ou a avó de z se for verdade que x é o pai ou a mãe de y e que y é o pai ou a mãe de z , para alguns valores específicos para as variáveis x , y e z .

A base de dados Prolog pode ser consultada interativamente com sentenças-objetivo, um exemplo do qual é

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

Tal sentença pergunta se bob é o pai (`father`) de $darcie$. Quando tal consulta, ou objetivo, é apresentada para o sistema Prolog, ele 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, ele mostra “`false`”.

2.13.3 Avaliação

Nos anos 1980, um grupo relativamente pequeno de cientistas da computação acreditava que a programação lógica fornecia a melhor esperança 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 abordagens não imperativas, os programas escritos em linguagens lógicas têm provado ser ineficientes se comparados aos programas imperativos equivalentes. Segundo, foi determinado que essa abordagem é efetiva 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 em maiores detalhes 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, quase metade das aplicações de computadores no DoD eram sistemas embarcados, nos quais a parte de hardware do computador é embarcada no dispositivo que ele controla ou para o qual ele 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 pelo DoD. Cada contratado para um projeto de defesa podia definir uma linguagem nova e diferente para cada contrato¹². 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, o Exército, a Marinha e a Força Aérea dos Estados Unidos propuseram, cada uma delas independentemente, em 1974, o desenvolvimento de uma única linguagem de alto nível para sistemas embarcados.

2.14.2 Processo de projeto

Ao notar esse amplo interesse, Malcolm Currie, diretor de pesquisa e engenharia de defesa, em janeiro de 1975, formou o High-Order Language Working Group (HOLWG – Grupo de Trabalho para Linguagem de Alto Nível), 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, França e Alemanha Ocidental. Seu objetivo inicial era:

- 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), 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 Steelman (Homem de Aço) em junho de 1978 (Departament of Defense, 1978).

Após um tedioso processo, as propostas submetidas para a linguagem foram reduzidas a quatro finalistas, todas baseadas no Pascal. Em maio de 1979, a proposta de projeto da Cii Honeywell/Bull foi escolhida a partir

¹² Isso ocorria devido ao uso disseminado de linguagens de montagem para sistemas embarcados, cuja maioria usava processadores especializados.

dos quatro finalistas como o projeto que seria usado. A equipe de projeto da Cii Honeywell/Bull, único competidor estrangeiro, era liderada por Jean Ichbiah.

Na primavera de 1979, Jack Cooper, do Comando Material da Marinha, recomendou o nome para a nova linguagem, Ada, o qual 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 conduzidos 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 sugeriu pequenas modificações em vez de mudanças drásticas e rejeições completas. Baseada nos relatórios sobre a linguagem, a próxima versão da especificação de requisitos, o documento Stoneman (Homem de Pedra) (Department 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.

Pacotes na linguagem Ada 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, conforme descrito no Capítulo 11.

A linguagem Ada inclui diversos recursos para o tratamento de exceções, os quais permitem que os programadores ganhem o controle após ter sido detectada a ocorrência de uma exceção, ou erros em tempo de execução, dentro de uma variedade de exceções possíveis. O tratamento de exceções é discutido no Capítulo 14.

As unidades de programas podem ser genéricas em Ada. Por exemplo, é possível escrever um procedimento de ordenação que usa 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 faixa de unidades de programas que podem ser reutilizadas, em vez de duplicadas, pelos programadores. Tipos genéricos são discutidos nos Capítulos 9 e 11.

A linguagem Ada também fornece a execução concorrente de unidades de programa especiais, chamadas tarefas, usando o mecanismo *rendezvous*. *Rendezvous* é o nome de um método de sincronização e comunicação intertarefas. Concorrência é o assunto discutido no Capítulo 13.

2.14.4 Avaliação

Talvez os aspectos mais importantes do projeto da linguagem Ada a serem considerados são:

- 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 alguém poder 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 esperado isso, o desenvolvimento de um compilador para a linguagem Ada era uma tarefa difícil. Apenas em 1985, quase quatro anos após o projeto da linguagem estar completo, é que compiladores Ada realmente usáveis começaram a aparecer.

O criticismo mais sério em relação à Ada em seus primeiros anos foi que a linguagem era muito grande 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 o qual ela foi projetada. Por outro lado, outros a consideram o epítome do projeto de linguagens de sua época. De fato, até mesmo Hoare no fim das contas suavizou sua visão da linguagem.

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

```
-- Programa de exemplo em Ada
-- Entrada: Um inteiro, listlen, onde listlen é menor do que
--           100, seguido por valores inteiros listlen
-- Saída:   O número de valores de entrada que são maiores
--           do 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
-- Read input data into an array and compute the sum
  for Counter := 1 .. List_Len loop
    Get (Int_List(Counter));
    Sum := Sum + Int_List(Counter);
  end loop;
-- Calcula a média
  Average := Sum / List_Len;
-- Count the number of values that are > average
  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

Dois dos mais importantes novos recursos de Ada 95 são brevemente descritos nos parágrafos seguintes. No restante deste livro, iremos usar o nome Ada 83 para a versão original e Ada 95 (seu nome atual) para a versão posterior quando for importante distinguir entre as duas. Em discussões sobre recursos comuns em 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 àqueles herdados de uma classe base. Isso fornece herança, um ingrediente chave em linguagens de programação orientadas a objetos. A vinculação dinâmica de chamadas a definições de subprogramas é realizada por meio de 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 principal da programação orientada a objetos. Tais recursos de Ada 95 são discutidos no Capítulo 12.

O mecanismo de *rendezvous* de Ada 83 fornecia apenas um meio 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. Os novos recursos de Ada 95 para concorrência e dados compartilhados são discutidos no Capítulo 13.

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

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 visão de prever a disponibilidade futura de computadores de mesa poderosos. Lembre-se de que os primeiros sistemas de microcomputadores não foram comercializados até o meio dos anos 1970, e eles estavam apenas remotamente relacionados às máquinas vislumbradas por Kay, as quais deveriam executar 1 milhão ou mais 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 humano-computador. Os computadores do final dos anos 1960 eram orientados a lote 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 Papert, nos quais 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, a qual Kay havia ajudado a projetar. Flex era baseada primariamente no SIMULA 67. O Dynabook usava o paradigma em 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 um

teclado quanto tocando a tela com seus dedos. Após o projeto preliminar do Dynabook lhe dar um doutorado, o objetivo de Kay se tornou ver tal máquina construída.

Kay conseguiu entrar no Xerox Palo Alto Research Center (Xerox PARC – Centro de Pesquisa da Xerox em Palo Alto) e apresentar suas ideias sobre o Dynabook. Isso o levou a ser empregado lá e, subsequentemente, ajudar 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 do hardware 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 o 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 casavam com a visão original de Alan Kay.

2.15.2 Visão geral da linguagem

O mundo de Smalltalk é populado por nada além de objetos, de constantes inteiros até grandes sistemas de software complexos. Toda a computação em Smalltalk é feita pela mesma técnica uniforme: enviando 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 que o processamento solicitado foi completado. A diferença fundamental entre uma mensagem e uma chamada a subprograma é que 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 ao código de um subprograma. Normalmente, os dados a serem processados pelo subprograma são enviados a ele como um parâmetro¹³.

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

A sintaxe do Smalltalk não é parecida com qualquer outra linguagem de programação, em grande parte por causa do uso de mensagens, em vez de expressões lógicas e aritméticas e sentenças de controle convencionais. Uma das construções de controle do Smalltalk é ilustrada no exemplo da próxima subseção.

¹³ Obviamente, uma chamada de método também pode passar dados a serem processados pelo método chamado.

2.15.3 Avaliação

Smalltalk teve um grande papel na promoção de dois aspectos separados da computação: interfaces gráficas com o usuário e programação orientada a objetos. O sistema de janelas que é agora o método dominante de interfaces com o usuário em sistemas de software cresceu a partir do Smalltalk. 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 serem do SIMULA 67, elas alcançaram a maturidade apenas em 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 desenhar 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 COMBINANDO RECURSOS IMPERATIVOS E ORIENTADOS A OBJETOS: C++

As origens da linguagem C foram discutidas na Seção 2.1, as do Simula 67 foram discutidas na Seção 2.10 e as do Smalltalk foram discutidas na Seção 2.15. C++ tem diversos recursos de linguagem, emprestados do Simula 67,

sobre a linguagem C para oferecer suporte aos recursos em que o Smalltalk foi pioneiro. C++ evoluiu a partir do C com uma série de modificações para melhorar seus recursos imperativos e adicionar construções para 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 no C incluíam a de verificação de tipos e conversão de parâmetros de funções e classes, as quais estavam relacionadas àquelas de SIMULA 67 e 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 do C com Classes. O primário era fornecer uma linguagem na qual os programas pudessem ser organizados da mesma forma que no SIMULA 67 – ou seja, com classes e herança. Um segundo objetivo importante era que deveriam existir penalidades pequenas ou nenhuma em termos de desempenho ao C. Por exemplo, a verificação de faixa de índices de vetores não foi considerada por causa de uma desvantagem significativa de desempenho, em relação ao C, que poderia resultar disso. Um terceiro objetivo do C com Classes era que ele poderia ser usado para quaisquer aplicações para as quais C poderia, então praticamente nenhum dos recursos de C seriam removidos, nem aqueles considerados inseguros.

Em 1984, a linguagem foi estendida com inclusão de métodos virtuais, os quais 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 ela é descrita em Stroustrup (1984).

Em 1985, a primeira implementação disponível apareceu: um sistema chamado Cfront, o qual traduzia programas C++ em C. Essa versão do Cfront e a de C++ que a ferramenta implementava foram chamadas de *Release 1.0*. Essa versão é descrita em Stroustrup (1986).

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

O *Release 3.0* de C++ evoluiu entre 1989 e 1990. Ele adicionou *templates*, os quais 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, a qual incluía uma nova versão de C++, chamada de Managed C++ (ou C++ Gerenciado), 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 ao C# na Seção 2.19. Como o .NET não suporta herança múltipla, o MC++ também não o faz.

2.16.2 Visão geral da linguagem

C++ fornece duas construções que definem tipos: classes e estruturas, com poucas diferenças entre as duas. Na prática, estruturas que incluem definições de métodos são geralmente usadas. Herança múltipla é suportada. Em C++, os métodos são geralmente chamados de funções membro.

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

Operadores em C++ podem ser sobre carregados – ou seja, o usuário pode criar novos operadores para os já existentes em tipos definidos pelo usuário. Métodos em C++ também podem ser sobre carregados, e isso significa que o usuário pode definir mais de um método com o mesmo nome, desde que o número ou tipos dos parâmetros seja diferente.

A vinculação dinâmica em C++ é fornecida por métodos virtuais. Esses métodos definem operações dependentes do tipo, usando métodos sobre carregados, dentro de uma coleção de classes que são relacionadas por herança. Um ponteiro para um objeto de uma classe A pode também apontar para objetos de classes que têm a classe A como um ancestral. Quando esse ponteiro aponta para um método virtual sobre carregado, 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++ inclui tratamento de exceções, significativamente diferente daquele de Ada. Uma diferença é que exceções detectáveis por hardware não podem ser tratadas. As construções de tratamento de exceção de Ada e C++ são discutidas no Capítulo 14.

2.16.3 Avaliação

C++ rapidamente se tornou (e se mantém) uma linguagem muito popular. Um fator para sua popularidade é a disponibilidade de compiladores bons e baratos. Outro é que C++ é quase completamente compatível com C (os programas em C podem, com poucas mudanças, ser compilados como programas

C++) e, na maioria das implementações, é possível ligar código C++ com código em C. Por último, no momento em que C++ apareceu pela primeira vez, quando a programação orientada a objetos começou a receber mais atenção, C++ era a única linguagem disponível adequada para grandes projetos comerciais de software.

No lado negativo, como C++ é uma linguagem muito grande e complexa, ela sofre deficiências similares àquelas da linguagem PL/I. C++ herdou muitas das inseguranças de C, tornando-a menos segura do que linguagens como Ada e Java. Os recursos de orientação a objetos de C++ são descritos em detalhes no Capítulo 12.

2.16.4 Uma linguagem relacionada: Eiffel

Eiffel é outra linguagem híbrida, que contém tanto recursos imperativos quanto orientados a objetos (Meyer, 1992). Eiffel foi projetada por uma pessoa, Bertrand Meyer, francês, que vive na Califórnia, nos EUA. A linguagem inclui recursos para oferecer suporte a tipos abstratos de dados, herança e vinculação dinâmica, de forma que oferece suporte completo à programação orientada a objetos. Talvez o recurso que mais distingue Eiffel é o uso integrado de asserções para garantir o “contrato” entre subprogramas e seus chamadores. É uma ideia que nasceu com Plankalkül, mas que foi ignorada pela maioria das linguagens projetadas desde então. É natural comparar Eiffel com C++. Eiffel é menor, mais simples e mais segura, mas tem quase a mesma expressividade e facilidade de escrita. As razões para a rápida popularidade de C++, enquanto Eiffel tem um uso muito mais limitado, não são difíceis de determinar. C++ era a maneira mais fácil para organizações de desenvolvimento de software se moverem para a programação orientada a objetos, porque, em muitos casos, seus desenvolvedores já conheciam C. Eiffel não desfrutou de tal caminho fácil em termos de adoção. Além disso, para os primeiros anos nos quais C++ se espalhou, o sistema Cfront estava disponível e era barato. C++ tinha o aval do prestigioso *Bell Labs*, enquanto Eiffel era mantida por Bertran Meyer e sua companhia de software relativamente pequena, a Interactive Software Engineering.

2.16.5 Outra linguagem relacionada: Delphi

Delphi (Lischner, 2000) é uma linguagem híbrida, similar a C++ porque foi criada por meio da adição de suporte a orientação a objetos, dentre outras coisas, a uma linguagem imperativa existente, Pascal. Muitas das diferenças entre C++ e Delphi são resultado das linguagens predecessoras e das culturas de programação que as envolviam e das quais elas são derivadas. Como C é uma linguagem poderosa, mas potencialmente insegura, C++ também casa com essa descrição, ao menos nas áreas de verificação de faixas de índices de vetores, aritmética de ponteiros e seus numerosos tipos de coerção. Da mesma forma, como Pascal é mais elegante e mais seguro do que C, Delphi é

mais elegante e seguro do que C++. Delphi também é uma linguagem menos complexa. Por exemplo, ela não permite sobrecarga de operadores definida pelo usuário, subprogramas genéricos e classes parametrizadas, os quais são parte de C++. Delphi, como Visual C++, fornece uma interface gráfica com o usuário (GUI) para o desenvolvedor e maneiras simples para criar interfaces GUI para aplicações escritas em Delphi. A linguagem Delphi foi projetada por Anders Hejlsberg, que havia desenvolvido o sistema Turbo Pascal. Ambos foram comercializados e distribuídos pela Borland. Hajlsbert foi também o projetista líder de 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 flexibilidade de C++, mas em uma linguagem menor, mais simples e mais segura.

2.17.1 Processo de projeto

Java, como muitas linguagens de programação, foi projetada para uma aplicação que parecia não ter uma linguagem existente satisfatória. Em 1990, a Sun Microsystems determinou que existia 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 para tal linguagem. Pode não parecer que a confiabilidade seria um fator importante no software para um forno de micro-ondas. Se um forno tem um problema de software, provavelmente não será um grave risco para ninguém e provavelmente não levará a grandes casos na justiça. Entretanto, se o sistema de software de um modelo em particular 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 pensavam ser uma necessidade. C++ suportava programação orientada a objetos, mas a equipe da Sun o julgava muito grande e complexo, 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 pro-

jeto foi guiado pelo objetivo fundamental de fornecer simplicidade e confiabilidade maiores do que acreditavam serem 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 foram comercializados. Quando a World Wide Web se tornou bastante usada, iniciando em 1993, por causa dos novos navegadores gráficos, descobriu-se que Java era uma ferramenta útil de programação para a Web. Em particular, os *applets* Java, programas relativamente pequenos cuja saída pode ser incluída e mostrada em documentos Web, rapidamente se tornaram populares do meio para o fim da década de 1990. Em seus primeiros anos de uso público, a Web era a aplicação de Java 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 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++ construirem classes que encapsulam vetores, de forma a adicionar recursos como verificação de índice de faixa, 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 a instâncias de classes. Todos os objetos são alocados no monte. Embora ponteiros e referências possam parecer bastante semelhantes, existem algumas diferenças semânticas importantes. Ponteiros apontam para locais de memória, referências apontam para objetos. Isso torna a aritmética de referências sem sentido, eliminando tal prática passível de erros. A distinção entre um valor de ponteiro e o valor para o qual ele aponta é responsabilidade do programa em muitas linguagens, nas quais os ponteiros precisam ser explicitamente desreferenciados. As referências são sempre implicitamente desreferenciadas, quando necessário. Dessa forma, elas se comportam mais como variáveis escalares ordinárias.

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 o primeiro oferece suporte à herança múltipla diretamente em suas definições de classes. Algumas pessoas acreditam que a herança múltipla leva a mais complexidade e confusão do que traz benefícios. Java oferece suporte apenas para herança simples de classes, apesar de alguns dos benefícios de herança múltipla poderem ser obtidos pelo uso de interfaces.

Dentre 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 exclusivo. 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 libera 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 fizeram bem em remover o excesso e/ou recursos inseguros de C++. Por exemplo, a eliminação de metade das coerções de atribuição feitas em C++ é um passo a 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 o escopo de aplicações que podem ser escritas na linguagem, assim como as bibliotecas de classes para interfaces gráficas com o 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 poderia ser traduzida para uma forma intermediária e “executada” em qualquer plataforma que tivesse 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 do que a execução de código de máquina. A versão inicial do interpre-

tador Java, chamado de Máquina Virtual Java (JVM), era ao menos 10 vezes mais lento do que os programas compilados em C equivalentes. Entretanto, muitos programas Java são agora traduzidos para código de máquina antes de serem executados, usando compiladores Just-in-Time (JIT). Isso torna a eficiência de programas Java competitiva com a dos programas escritos em linguagens convencionalmente compiladas, como C++.

O uso de Java aumentou mais rapidamente do que o de qualquer outra linguagem de programação. Inicialmente, isso ocorreu por causa de seu valor na programação de documentos Web dinâmicos. Outro fator é o sistema de compilação/interpretação para Java ser gratuito e fácil de obter na Web. É claro que uma das razões para a rápida ascensão de Java é que os programadores gostam de seu projeto. Sempre existiram alguns desenvolvedores que pensavam que a linguagem C++ era muito grande e complexa para ser prática e segura. Java ofereceu a eles uma alternativa com muito do poder de C++, mas em uma linguagem mais simples, segura. Java é agora usada em uma variedade de áreas de aplicação.

A versão mais recente, que apareceu em 2004 e foi chamada de Java 1.5, mas depois renomeada para Java 5.0, inclui algumas adições significativas. Tais adições incluem uma classe de enumerações, tipos genéricos e uma nova construção de iteração.

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

```
/ Programa de exemplo em Java
// Entrada: Um inteiro, listlen, onde listlen é menor do que
//           100, seguido por valores inteiros listlen
// Saída:    O número de valores de entrada que são maiores
//           do 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 do que a média */
    for (counter = 0; counter < listlen; counter++)
        if (intlist[counter] > average) result++;
/* Imprimir o resultado */
    System.out.println(
        "\nNumber 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 25 anos. As primeiras eram usadas por meio de uma lista de comandos, chamada de *script*, em um arquivo a ser interpretado. 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. Nessa base, 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 usadas 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 no Bell Labs (Aho et al., 1988). A awk começou como uma linguagem de geração de relatórios, mas 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. Apesar de ser geralmente chamada de linguagem de *scripting*, ela é mais similar a 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 no restante do 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 um arroba (@) e todos os nomes de dispersões (*bashes*) – dispersões são brevemente descritas abaixo – começam com sinais de percentual (%). Essa convenção torna os nomes em programas mais legíveis do 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 distintos – apesar de crípticos, 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 separam de outros vetores das linguagens imperativas comuns. Primeiro, eles têm tamanho dinâmico, ou seja, podem crescer e encolher conforme necessário durante a execução. Segundo, vetores podem ser esparsos, ou seja, pode haver espaços em branco entre os elementos. Esses espaços em branco não ocupam espaço em 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 para um número, é usado zero e não existe mensagem de aviso ou erro para o usuário. Esse efeito 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 era 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. Perl é usada como uma 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 do que
#           100, seguido por valores inteiros listlen
# Saída:   O número de valores de entrada que são maiores
#           do 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 do 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 no meio dos anos 1990, após a aparição dos primeiros navegadores gráficos. A necessidade de computação associada com documentos HTML, os quais por si só eram completamente estáticos, rapidamente se tornou crítica. A computação 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. Ambas abordagens foram agora substituídas em grande parte por novas tecnologias, primariamente linguagens de *scripting*.

JavaScript (Flanagan, 2002), originalmente chamado LiveScript, foi desenvolvida na Netscape. 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. JavaScript também é usada com o framework de desenvolvimento Web Rails.

Apesar de seu nome, JavaScript é relacionada com Java apenas pelo uso de uma sintaxe similar. Java é fortemente tipada, mas JavaScript é dinamicamente tipada (veja 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 e para vinculação dinâmica de chamadas a métodos.

Um dos usos mais importantes de JavaScript é para a 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 de um documento HTML e interpretado por um navegador Web.

```
// example.js
// Entrada: Um inteiro, listlen, onde listlen é menor do que
//           100, seguido por valores numéricos listlen
// Saída:   O número de valores de entrada que são maiores
//           do 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 do que a média
    result = 0;
    for (counter = 0; counter < listLen; counter++) {
        if (intList[counter] > average)
            result++;
    }
}
```

```
for (counter = 0; counter < listLen; counter++)
    if (intList[counter] > average) result++;
// Display the results
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 (Converse e Park, 2000) foi desenvolvido por Rasmus Lerdorf, membro do Grupo Apache, em 1994. Sua motivação 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 desenvolvido, distribuído e suportado 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 for requisitado por um navegador. O código PHP normalmente produz código HTML como saída, o que 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 à 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. PHP fornece suporte para muitos sistemas de gerenciamento de bancos de dados. Isso o torna uma excelente linguagem para construir programas que precisam de acesso Web a bases de dados.

2.18.4 Origens e características de Python

Python (Lutz e Ascher, 2004) é uma linguagem de *scripting* orientada a objetos interpretada relativamente recente. Seu projeto original foi feito por Guido van Rossum no Stichting Mathematisch Centrum, na Holanda, no iní-

cio dos anos 1990. Seu desenvolvimento é feito agora pela Python Software Foundation (Fundação de Software Python). 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 distribuição padrão da indústria para Windows está disponível em www.activestate.com/Products/ActivePython. Implementações para outras plataformas estão disponíveis em www.python.org, o qual fornece também informações sobre Python.

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. 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.

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 era a falta de satisfação de seu projetista com Perl e Python. Apesar de tanto Perl quanto Python oferecerem suporte à programação orientada a objetos, nenhuma delas é uma linguagem puramente orientada a objetos, ao menos no sentido de que ambas têm tipos primitivos (não objetos) e possibilitam o uso de funções.

O recurso característico primário de Ruby é que se trata 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 a tipagem dinâmica é usada. 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 \$_.

Como no caso de Python, qualquer usuário pode estender ou modificar Ruby, que foi a primeira linguagem de programação projetada no Japão a atingir um uso relativamente amplo nos Estados Unidos.

2.18.6 Origens e características de Lua

Lua 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 extensibilidade como um de seus objetivos primários. Dentre as linguagens que influenciaram seu projeto estão Scheme, Icon e Python.

Lua é similar a JavaScript no sentido em que não oferece suporte a 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, no caso de Lua, a linguagem pode ser estendida para oferecer suporte à programação orientada a objetos.

Como em Scheme, as funções em Lua são valores de primeira classe. Além disso, a linguagem oferece suporte para *closures*. Essas capacidades permitem que ela seja usada 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 coleção 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 de fornecer apenas o necessário e maneiras simples de estender a linguagem para permitir que ela se encaixe em uma variedade de áreas de aplicação. Muito de sua extensibilidade é derivada de sua estrutura de dados, a tabela, a qual pode ser customizada usando o conceito de metatabellas 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 para um código intermediário e interpretada. Ela pode ser facilmente embarcada em outros sistemas, em parte por causa do tamanho pequeno de seu interpretador, apenas cerca de 150Kbytes.

Durante 2006 e 2007, a popularidade de Lua cresceu rapidamente, em decorrência de seu uso na indústria de jogos. A sequência de linguagens de *scripting* que apareceram nos últimos 20 anos já produziu diversas linguagens bastante usadas. Lua, a última a chegar, está se tornando rapidamente uma delas.

2.19 UMA LINGUAGEM BASEADA EM C PARA O NOVO MILÊNIO: C#

C#, ao lado da nova plataforma de desenvolvimento .NET¹⁴, foi anunciada pela Microsoft em 2000. Em janeiro de 2002, versões de produção de ambas estavam disponíveis.

2.19.1 Processo de projeto

O C# é baseado em C++ e Java, mas também inclui algumas ideias do Delphi e do Visual BASIC. Seu projetista líder, Anders Hejlsberg, também projetou o Turbo Pascal e o Delphi, o que explica as partes Delphi da herança do 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#, Visual Basic .NET, C++ gerenciado, J# .NET e JScript .NET, usam o Common Type System (CTS – Sistema de Tipos Comum). 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. As cinco linguagens .NET são compiladas para o mesmo formato intermediário, Intermediate Language (IL – Linguagem Intermediária)¹⁵. Diferentemente de Java, entretanto, a IL nunca é interpretada. Um compi-

¹⁴ O sistema de desenvolvimento .NET é brevemente discutido no Capítulo 1.

¹⁵ Inicialmente, a IL era chamada de MSIL (de Microsoft Intermediate Language – Linguagem Intermediária da Microsoft), mas muitas pessoas acharam o nome extenso.

lador Just-in-Time é usado para converter IL em código de máquina antes de esse ser executado.

2.19.2 Visão geral da linguagem

Muitos acreditam que um dos avanços mais importantes de Java em relação a C++ está na exclusão de alguns recursos. Por exemplo, C++ oferece suporte a herança múltipla, ponteiros, estruturas, tipos `enum`, sobrecarga de operadores e uma sentença *goto*, mas Java não inclui nenhum desses recursos¹⁶. Os projetistas de C# discordaram com essas remoções, o que fez com que esses recursos, exceto herança múltipla, fossem trazidos de volta na nova linguagem.

Dando crédito aos projetistas de C#, 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. Estruturas (*structs*) em C# são discutidas no Capítulo 12 – “Suporte à Programação Orientada a Objetos”. C# dá um passo à frente ao melhorar a sentença `switch` usada em C, C++ e Java. Nessas linguagens, não existe um desvio implícito no final dos segmentos selecionáveis de código, que causava inúmeros erros de programação. Em C#, quaisquer segmentos `case` não vazios devem terminar com uma sentença de desvio incondicional. A sentença `switch` de C# é discutida no Capítulo 8 – “Estruturas de Controle no Nível de Sentenças”.

Apesar de C++ incluir ponteiros para funções, eles compartilham a falta de segurança inerente nos ponteiros para variáveis de C++. C# inclui um novo tipo, os *delegates*, referências a métodos que são tanto orientadas a objetos quanto seguras em relação a tipos. *Delegates* são usados para implementar manipuladores de eventos e *callbacks*¹⁷. *Callbacks* são implementadas em Java por meio de interfaces; em C++, ponteiros para métodos são usados.

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

¹⁶ N. de R. T.: Tipos `enum` estão disponíveis em Java desde a versão 1.5, de 2004.

¹⁷ Quando um objeto chama um método de outro objeto e precisa ser notificado quando esse método tiver completado sua tarefa, o método chamado chama seu chamador novamente, o que é denominado um *callback*.

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 12¹⁸.

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

2.19.3 Avaliação

C# foi criada como um avanço 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 passo atrás, C# inclui algumas construções que a movem à frente de suas antecessoras. Alguns de seus recursos certamente serão adotados por linguagens de programação em um futuro próximo.

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

```
// Programa de exemplo em C#
// Entrada: Um inteiro, listlen, onde listlen é menor do que
//           100, seguido por valores inteiros listlen.
// Saída:   O número de valores de entrada que são maiores
//           do 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 ...
        }
    }
}
```

¹⁸ Esse recurso foi adicionado à linguagem Java em sua versão 5.0.

```
// Calcula a média
    average = sum / listlen;
// Conta o número de valores que são maiores do 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 seguintes subseções introduzem duas linguagens híbridas, XSLT e JSP.

2.20.1 XSLT

XML (eXtensible Markup Language – Linguagem de Marcação Extensível) é uma linguagem de metamarcação usada para definir linguagens de marcação. Linguagens de marcação derivadas de XML são usadas para definir documentos de dados, chamados de documentos 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 que possam ser efetivamente visualizados ou impressos. Em muitos casos, as transformações são para XHTML, 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 XHTML 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. 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 XML¹⁹, usando as transformações descritas no XSLT. O 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 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, permindo 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 que roda 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 a documentos Web dinâmicos e fornecer outras necessidades de processamento para 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 mostram como ele é. A marcação JSTL no documento JSP é processada, conforme discutido no próximo parágrafo. O *servlet* produzido pelo processador JSP é executado pelo *servlet container*.

A 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 que outros de HTML e XML. Um dos elementos de controle de ação mais usados de JSTL é o **if**, que especifica uma ex-

¹⁹ O documento de saída do processador XSLT pode ser também em HTML ou texto plano.

pressão booleana como um atributo²⁰. O conteúdo do elemento **if** (o texto entre a *tag* de abertura (`<if>`) e a *tag* de fechamento (`</if>`)) é código de marcação 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 de marcação 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.

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 múltipla seleção, JSTL tem os elementos **choose**, **when** e **otherwise**. JSTL também inclui **forEach**, o qual 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 e o ambiente de algumas das linguagens de programação mais importantes. Este capítulo deve ter dado ao leitor uma boa perspectiva em relação às questões atuais do projeto de linguagens. Esperamos ter preparado o terreno para uma discussão aprofundada dos recursos importantes das linguagens contemporâneas.

NOTAS BIBLIOGRÁFICAS

Talvez a fonte mais importante de informações históricas sobre o desenvolvimento de linguagens de programação seja *History of Programming Languages*, editada por Richard Wexelblat (1981). Ele contém a perspectiva histórica do desenvolvimento e do ambiente 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 de uma *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 o Fortran

²⁰ Um atributo em HTML, embarcado na *tag* de abertura de um elemento, fornece informações adicionais sobre o elemento.

(inclusive). O artigo inclui programas de exemplo para demonstrar os recursos de muitas dessas linguagens.

Outro livro de grande interesse é o *Programming Languages: History and Fundamentals*, de Jean Sammet (1969), um trabalho de 785 páginas repleto de detalhes de 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 em Plankalkül.
3. Como eram implementados os pseudocódigos do início dos anos 1950?
4. 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 que apareceram pela primeira vez no computador IBM 704 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 o Fortran foi projetado?
9. Qual foi a fonte de todas as sentenças de fluxo de controle do Fortran I?
10. Qual foi o recurso mais significativo adicionado ao Fortran I para chegar ao Fortran II?
11. Quais sentenças de controle de fluxo foram adicionadas ao Fortran IV para chegar ao Fortran 77?
12. Que versão do Fortran foi a primeira a ter quaisquer tipos de variáveis dinâmicas?
13. Que versão do 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 o LISP foi desenvolvido? 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 o ALGOL 60?
19. Em que versão do ALGOL a estrutura de bloco apareceu?
20. Que elemento de linguagem que faltava ao ALGOL 60 fez com que suas chances de uso disseminado fossem prejudicadas?
21. Que linguagem foi projetada para descrever a do ALGOL 60?
22. Em que linguagem o COBOL foi baseado?
23. Em que ano o processo de projeto do COBOL começou?
24. Que estrutura de dados apareceu no COBOL que foi originada em Plankalkül?
25. Que organização foi a maior responsável pelo sucesso inicial do COBOL (em termos de uso)?
26. Para que grupo de usuários foi focada a primeira versão do BASIC?

27. Por que BASIC foi uma linguagem importante no início dos anos 1980?
28. PL/I foi projetada para substituir que duas outras linguagens?
29. Para que 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 no ALGOL 68, geralmente creditada ao 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 o tornam menos seguro do 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 de 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. O que as linguagens Ada e COBOL têm em comum?
45. Qual foi a primeira aplicação para Java?
46. Que característica de Java é mais evidente em JavaScript?
47. Como o sistema de tipos de PHP e JavaScript diferem daquele de Java?
48. Que estrutura de vetor é incluída em C#, mas não em C, C++ ou Java?
49. Quais são as duas linguagens que a versão original de Perl pretendia substituir?
50. Para qual área de aplicação JavaScript é mais usada?
51. Qual é o relacionamento entre JavaScript e PHP, em termos de utilização?
52. A estrutura de dados primária de PHP é uma combinação de que duas outras estruturas de dados de outras linguagens?
53. Que estrutura de dados Python usa em vez de vetores?
54. Que características Ruby compartilha com Smalltalk?
55. Que característica dos operadores aritméticos de Ruby os tornam únicos entre aqueles de outras linguagens?
56. Que estruturas de dados são construídas em Lua?
57. Lua é normalmente compilada, puramente interpretada ou impuramente interpretada?
58. Que recurso das classes do Delphi é incluído em C#?
59. Que deficiência da sentença **switch** do C é sanada com as mudanças feitas por C# a essa construção?
60. Qual é a plataforma primária na qual o C# é usado?
61. Quais são as entradas para um processador XSLT?
62. Qual é a saída de um processador XSLT?
63. Que elemento da JSTL é relacionado a um subprograma?
64. Por que um documento JSP é convertido por um processador JSP?
65. Os *servlets* são executados?

CONJUNTO DE PROBLEMAS

1. Que recursos de Plankalkül você acha que teriam maior influência no Fortran 0 se os projetistas do 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. Como um projeto de pesquisa, compare as facilidades do Fortran 0 com as do sistema de Laning e Zierler.
5. Qual dos três objetivos originais do comitê de projeto do ALGOL, na 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, na sua opinião, por que o ALGOL 60 não se tornou uma linguagem amplamente usada.
9. Por que, na sua opinião, o 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 do que o 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 do argumento de que as linguagens muito complexas também são muito perigosas, e que devemos manter todas as linguagens pequenas e simples?
17. Você pensa que o projeto de linguagem por comitê é uma boa ideia? Justifique sua resposta.
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 do Fortran.
19. Construa uma tabela identificando todas as principais evoluções das linguagens, constando 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 do J++ e do C# da Microsoft e do Java da Sun. Leia alguns desses documentos, disponíveis em seus respectivos sites Web, e escreva uma análise das discordâncias existentes.
21. As linguagens de *scripting* têm evoluído as estruturas de dados de forma a substituir os vetores tradicionais. Explique a sequência cronológica desses avanços.
22. Dê duas razões para que a interpretação pura seja um método de implementação aceitável para diversas das linguagens de *scripting* recentes.

23. Perl 6, quando chegar, provavelmente será uma linguagem significativamente ampliada. Tente estimar se uma linguagem como Lua também crescerá continuamente ao longo de seu tempo de vida. Justifique sua resposta.
24. Por que, em sua opinião, aparecem novas linguagens de *scripting* mais frequentemente do que novas linguagens compiladas?
25. Dê 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 que use um vetor de estruturas que armazenem informações de estudantes, incluindo o nome, a idade, a média das notas como um 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**.

Capítulo 3

Descrevendo Sintaxe e 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 Descrevendo o significado de programas: semântica dinâmica

Este capítulo cobre os tópicos *sintaxe* e *semântica*. Após a definição dos termos, é apresentada uma discussão detalhada do 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, a descrição de precedência e a 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. Alguém poderia facilmente 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. O ALGOL 60 e o ALGOL 68 foram apresentados pela primeira vez usando 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 sofreram em função disso. Em contrapartida, algumas linguagens sofreram 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 dentro da organização que emprega o projetista, antes de os 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 procurarem em um manual de referência da linguagem. Livros-texto e cursos entram nesse processo, mas os manuais de linguagem são normalmente as únicas fontes oficialmente impressas 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 programas. 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 é executada. Caso contrário, o controle continua após a construção `while`. O controle retorna implicitamente para a expressão booleana para repetir o processo.

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 a sentença realiza.

Descrever a sintaxe é mais fácil do que descrever a semântica, especialmente porque uma notação aceita universalmente está disponível para a descrição de 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 linguagens 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 mais baixo nível. Essas pequenas unidades são chamadas de **lexemas** e sua descrição 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, dentre 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 *símbolo*. 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 de operação aritmética + tem um lexema possível. Considere a seguinte sentença Java:

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

Os lexemas e *tokens* dessa sentença são

<i>Lexemas</i>	<i>Tokens</i>
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 formalmente L usando o método de reconhecimento, precisamos construir um mecanismo R, chamado de dispositivo de reconhecimento, capaz de ler cadeias de caracteres a partir do alfabeto Σ . R pode indicar se uma determinada cadeia de entrada está ou não em L. Dessa forma, R iria aceitar ou rejeitar a cadeia fornecida. Tais dispositivos são como filtros, separando sentenças legais 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, por propósitos práticos, infinita, isso pode parecer um processo 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 ele 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 um 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 tentativa e erro. Por exemplo, para determinar a sintaxe correta de uma sentença em particular usando um compilador, o programador pode apenas submeter uma versão que ele estima estar correta e esperar para ver se o compilador 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 primárias e inovadoras em 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 chamadas de **gramáticas**, usadas para descrever a sintaxe das linguagens de programação.

3.3.1 Forma de *Backus-Naur* e gramáticas livres de contexto

No meio 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

Do meio para o final dos anos 1950, 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 são chamadas de livres de contexto e linguagens regulares, que acabaram sendo 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 linguagens de programação completa, com pequenas exceções, pode ser descrita por gramáticas livres de contexto. Como Chomsky era um linguista, seu interesse primário 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 do ALGOL 58. Um artigo marcante descrevendo o ALGOL 58 foi apresentado por John Backus, um proeminente membro do grupo ACM-GAMM, em uma conferência internacional em 1959 (Backus, 1959). O artigo introduzia uma nova notação formal para especificar a sintaxe de linguagens de programação, que foi levemente modificada por Peter Naur para a descrição do ALGOL 60 (Naur, 1960). O método revisado de descrição de sintaxe ficou conhecido como **Forma de Backus-Naur**, ou simplesmente **BNF**.

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

Apesar de o uso de BNF no relatório do 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 **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. BNF é uma metalinguagem para linguagens de programação. 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 símbolo no lado esquerdo da seta, chamado de **lado esquerdo** (LHS – do inglês *left-hand side*), é a abstração que está sendo definida. O texto no lado direito da seta é a definição de LHS. É chamado de **lado direito** (RHS – do inglês *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 **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 de uma 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. Múltiplas definições podem ser escritas como uma única regra, separadas pelo símbolo |, que significa um 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 ou 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 pela qual as diferentes construções devem aparecer, estruturas aninhadas a qualquer profundidade e mesmo expressar precedência e associatividade de operadores.

3.3.1.4 Descrevendo listas

Listas de tamanho variável em matemática são geralmente escritas usando uma elipse (...); 1, 2, ... é um exemplo. BNF não inclui elipses, então 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 BNF, a alternativa é a recursão. Uma regra é recursiva se seu lado esquerdo aparecer 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**. A geração de uma sentença é chamada de **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 + C ; <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, incluindo `<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 substi-

tuição são chamadas de **derivações mais à esquerda**. A derivação continua até que a forma sentencial não contenha mais nenhum não terminal. Essa forma sentencial, consistindo em apenas 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 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 são 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 atrativos das gramáticas é que elas naturalmente descrevem 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 previamente.

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.

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 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>
  
```

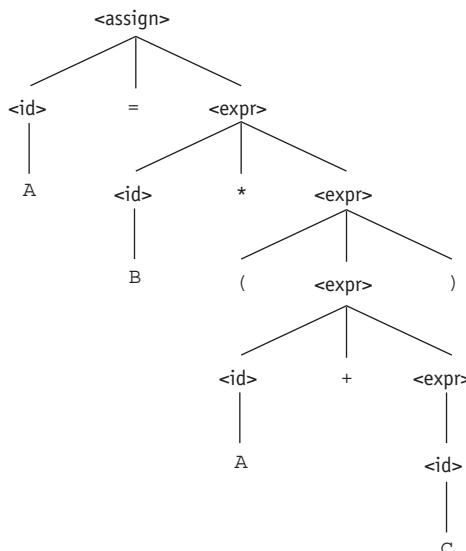


Figura 3.1 Uma árvore de análise sintática para a sentença simples $A = B * (A + C)$.

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 do 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.

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 unicamente. Esse problema é discutido em dois exemplos específicos nas seções seguintes.

Existem diversas características de uma gramática que são úteis para determinar se uma gramática é ambígua¹. Dentre essas estão: (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.

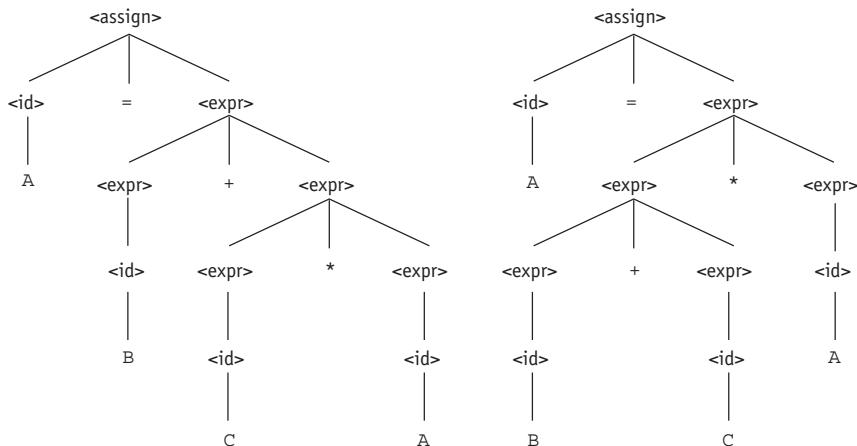


Figura 3.2 Duas árvores diferentes de análise sintática para a mesma sentença, $A = B + C * A$.

¹ Note que não é possível determinar matematicamente se uma gramática arbitrária é ambígua.

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 (é adição e depois multiplicação, ou vice-versa na expressão acima?). Essa questão pode ser respondida pela atribuição de diferentes níveis de precedência para os operadores. Por exemplo, se para o operador $*$ for atribuída uma precedência mais alta do que o $+$ (pelo projetista da linguagem), a multiplicação será feita 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 pode ser determinado 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 uma 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 $A + B * C$, $*$ seria a mais baixa na árvore, indicando que esse operador deve ser usado primeiro. Entretanto, na expressão $A * B + C$, $+$ seria o mais baixo.

Uma gramática para as expressões simples que estamos discutindo pode ser escrita de forma a ser tanto não ambígua quanto especificar uma precedência consistente 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 correto de $+$. A seguir, podemos definir `<term>` para gerar operadores $*$, usando `<term>` como o operando da esquerda e um novo não terminal, `<factor>`,

como o da direita. Agora, * irá sempre estar mais abaixo na árvore de análise sintática, simplesmente porque ele está mais longe do símbolo inicial do 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 no Exemplo 3.4 gera a mesma linguagem que os 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 as árvores de análise sintática e derivações é bastante próxima: ambas podem ser construídas, uma a partir da outra. Cada derivação com uma gramática 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 previamente dada. 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

```

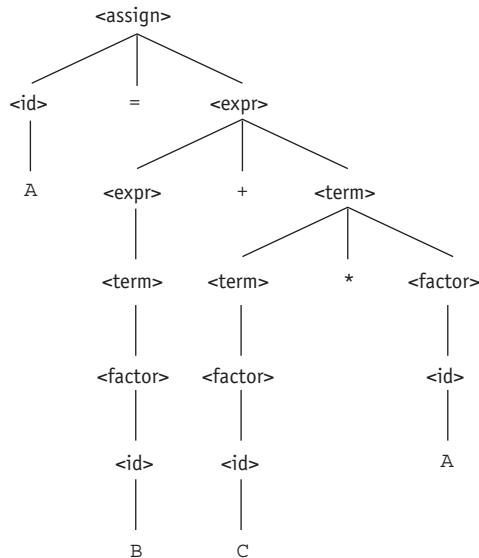


Figura 3.3 A árvore de análise sintática única para $A = B + C * A$ usando uma gramática não ambígua.

3.3.1.9 Associatividade de operadores

Quando uma expressão inclui dois operadores que têm a mesma precedência (como * e / normalmente têm) – por exemplo, $A / B * C$ – uma regra semântica é necessária para especificar qual dos operadores deve ter precedência². Essa regra é chamada de *associatividade*.

² Uma expressão com duas ocorrências do mesmo operador tem a mesma questão; por exemplo, $A / B / C$.

Como era o caso com a precedência, uma gramática para expressões pode corretamente definir a associatividade de operadores. Considere o seguinte exemplo de uma sentença de atribuição:

$A = B + C + A$

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 do que o operador de adição da direita. Essa é a ordem correta se a adição deve ser deixada 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 – assim, $(A + B) + C = A + (B + C)$.

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 adicionar 11 números juntos, onde um deles é 10^7 e os outros 10 são 1. Se os números pequenos (os 1s) são adicionados 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 primeiro adicionados juntos 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 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.

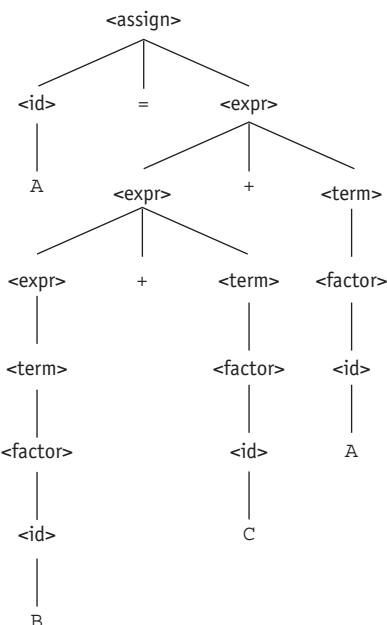


Figura 3.4 Uma árvore de análise sintática para $A = B + C + A$ ilustrando a associatividade da adição.

Quando uma regra gramatical tem seu lado esquerdo aparecendo também no início de seu lado direito, a regra é dita **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 tais algoritmos forem usados, a gramática deve ser modificada para remover a recursão à esquerda. Isso, por sua vez, desabilita 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 obrigue isso.

Na maioria das linguagens que o provêm, 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-then-else`

As regras BNF para um `if-then-else` em Ada são:

```
<if_stmt> → if <logic_expr> then <stmt>
           | if <logic_expr> then <stmt> else <stmt>
```

Se tivéssemos também $\langle \text{stmt} \rangle \rightarrow \langle \text{if_stmt} \rangle$, essa gramática seria ambígua. A forma sentencial mais simples que ilustra essa ambiguidade é

```
if <logic_expr> then if <logic_expr> then <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
  then if denom == 0
    then quotient = 0;
  else quotient = num / denom;
```

O problema é que, se a árvore de análise sintática na Figura 3.5 for usada como base para a tradução, a cláusula senão (`else`) é executada quando `done`

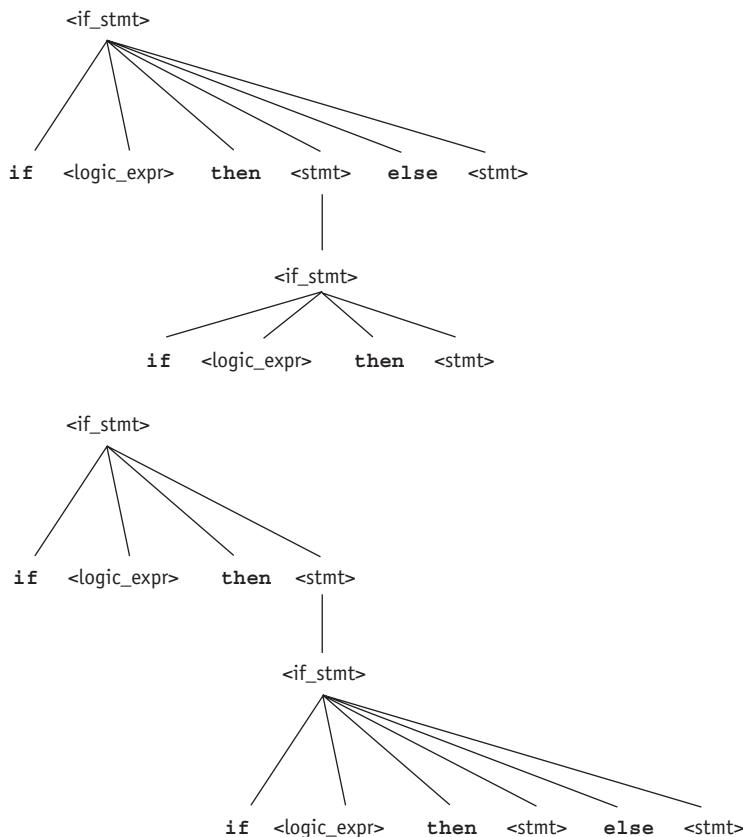


Figura 3.5 Duas árvores de análise sintática para a mesma forma sentencial.

não for verdadeira, o que provavelmente não é o que o autor da construção pretende. Examinaremos os problemas práticos associados com 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 o **then** mais próximo que ainda não está casado. Dessa forma, não pode existir uma sentença **if** sem um **else** entre um **then** e seu **else** correspondente. Então, para essa situação, as sentenças devem ser distinguidas entre aquelas que estão casadas e as que não estão, onde as sentenças que não estão casadas são **ifs** sem **else**. Todas as outras sentenças são casadas. O problema com a primeira gramática é que ela trata todas as sentenças como se tivessem uma importância sintática igual – ou seja, como se todas casassem.

Para refletir 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> then <matched> else <matched>
           | qualquer sentença fora if
<unmatched> → if <logic_expr> then <stmt>
               | if <logic_expr> then <matched> else <unmatched>
```

Existe apenas uma árvore de análise sintática possível, usando essa gramática, para a seguinte forma sentencial:

```
if <logic_expr> then if <logic_expr> then <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 que elas não sejam 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 pode ser repetida indefinidamente ou deixada de fora. Essa extensão permite que listas sejam construídas 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 eles ajudam a descrever. Nos casos em que esses metassímbolos também são símbolos terminais na linguagem sendo descrita, as instâncias que são símbolos terminais devem ser destacadas (sublinhadas ou colocadas entre aspas).

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 ao projetar o processo do analisador sintático para garantir a associatividade correta. Isso é discutido mais detalhadamente no Capítulo 4.

Algumas versões de EBNF permitem que um valor numérico seja anexado à chave direita para indicar um limite superior para o número de vezes em

que as partes envoltas pelas chaves possam ser repetidas. Além disso, algumas versões usam um sinal de adição (+) sobrescrito para indicar uma ou mais re-petições. Por exemplo,

`<compound> → begin <stmt> {<stmt>} end`

e

`<compound> → begin {<stmt>}+ end`

são equivalentes.

Nos últimos anos, algumas variações na BNF e na EBNF têm aparecido. Dentre elas, estão:

- Em vez da seta, um ponto e vírgula é usado e a RHS é colocada na pró-xima linha.
- Em vez de uma barra vertical para separar RHSs alternativas, elas são colocadas em linhas separadas.
- Em vez de colchetes para indicar que algo é opcional, o subscrito _{opt} é usado. Por exemplo,

`DeclaradorDeConstrutor → NomeSimples (ListaDeParâmetrosFormaisopt)`
• Em vez de usar o símbolo | em uma lista de elementos entre parênteses para indicar uma escolha, as palavras “one of” são usadas. Por exemplo,

`OperadorDeAtribuição → one of = *= /= %= += -= <<= >= = ^ = | =`

Existe um padrão para EBNF, o ISO/IEC 14977:1996 (1996), mas ele 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 requer aspas em todos os símbolos terminais; ele também especifica inúmeras outras regras de notação.

3.3.3 Gramáticas e reconhecedores

Neste capítulo, sugerimos que existia um forte relacionamento 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 algorítmicamente construído. Alguns sistemas de software foram desenvolvidos 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 analisadores léxicos é chamado yacc (*yet another compiler-compiler*; Johnson, 1975). Existem agora muitos sistemas disponíveis.

NOTA HISTÓRICA

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 usando uma gramática livre de contexto.

Ela é uma extensão de uma gramática livre de contexto, que permite certas regras de linguagens serem descritas de maneira conveniente, 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

Existem algumas características da estrutura das linguagens de programação que são difíceis de descrever com BNF – e algumas impossíveis. Como um exemplo de uma 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. Apesar de essa restrição poder ser especificada 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 muito grande para ser útil, porque o tamanho da gramática determina o tamanho do analisador sintático.

Como um 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.

Por causa dos 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 (veja 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 livres de contexto nas quais foram adicionados atributos, funções de computação de atributos e funções de predicado. **Atributos**, associados com símbolos de gramáticas (os símbolos terminais e não terminais), são similares a variáveis no sentido de que podem ter valores atribuídos a eles. **Funções de computação de atributos**, algumas vezes chamadas de funções semânticas, são associadas com 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 com 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 é uma com os seguintes recursos adicionais:

- Associado a cada símbolo X da gramática está um conjunto de atributos $A(X)$. O conjunto $A(X)$ é formado de 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 os **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 $X_0 \rightarrow X_1 \dots X_n$, os atributos sintetizados de X_0 são computados com funções semânticas da forma $S(X_0) = f(A(X_1), \dots, A(X_n))$. 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. Atributos herdados dos símbolos X_j , $1 \leq j \leq n$ (na regra acima) são computados com uma função semântica na forma $I(X_j) = f(A(X_0), \dots, A(X_n))$. 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 são geralmente restritos a funções na forma $I(X_j) = f(A(X_0), \dots, A(X(j-1)))$. Essa forma previne um atributo herdado de depender 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 $\{A(X_0), \dots, A(X_n)\}$ e um conjunto de valores de atributos literais. As únicas derivações permitidas com uma gramática de atributos são aquelas nas quais cada predicado associado com 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, a árvore é dita como **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, usada para armazenar nomes de variáveis e seus tipos. O conteúdo da tabela de símbolos é informado com base nas sentenças de declaração anteriores. Inicialmente, assumindo 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 uma 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 existem 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 distinguir cada um deles. Nem os subscritos nem os colchetes fazem parte da linguagem descrita.

Regra sintática: $\text{proc_def} \rightarrow \text{procedure } \langle\text{proc_name}\rangle[1]$
 $\qquad\qquad\qquad \langle\text{proc_body}\rangle \text{ end } \langle\text{proc_name}\rangle[2];$
Predicado: $\langle\text{proc_name}\rangle[1].\text{string} == \langle\text{proc_name}\rangle[2].\text{string}$

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, consideraremos um exemplo maior de uma 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 detalhadas a seguir. Os únicos nomes de variáveis são A, B e C. O lado direito das atribuições podem ser uma variável ou uma expressão a 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. O tipo da expressão quando os tipos dos operandos não são o mesmo 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 casar 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 seguintes parágrafos:

- *actual_type* – Um atributo sintetizado associado com os 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) filho(s) do não terminal `<expr>`.
- *expected_type* – Um atributo herdado associado com o 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.

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,

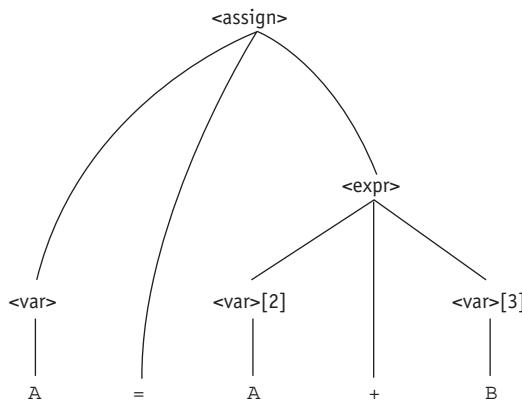


Figura 3.6 Uma árvore de análise sintática para $A = A + B$.

EXEMPLO 3.6 Uma gramática de atributos para sentenças de atribuição simples

1. Regra sintática: $\text{<assign>} \rightarrow \text{<var>} = \text{<expr>}$
 Regra semântica: $\text{<expr>.expected_type} \leftarrow \text{<var>.actual_type}$
2. Regra sintática: $\text{<expr>} \rightarrow \text{<var>[2]} + \text{<var>[3]}$
 Regra semântica: $\text{<expr>.actual_type} \leftarrow$
 - if ($\text{<var>[2].actual_type} = \text{int}$) and
 $\text{<var>[3].actual_type} = \text{int}$)
 - then int
 - else real
 - end if
 Predicado: $\text{<expr>.actual_type} == \text{<expr>.expected_type}$
3. Regra sintática: $\text{<expr>} \rightarrow \text{<var>}$
 Regra semântica: $\text{<expr>.actual_type} \leftarrow \text{<var>.actual_type}$
 Predicado: $\text{<expr>.actual_type} == \text{<expr>.expected_type}$
4. Regra sintática: $\text{<var>} \rightarrow A \mid B \mid C$
 Regra semântica: $\text{<var>.actual_type} \leftarrow \text{look-up}(\text{<var>.string})$

A função `lookup` busca um dado nome de variável na tabela de símbolos e retorna o tipo de dessa variável.

números entre colchetes são adicionados após os rótulos de nós repetidos, de forma que possam ser referenciados de maneira não ambígua.

3.4.6 Computando 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 pode ser realizado em uma ordem completamente descendente, da raiz para as folhas.

Alternativamente, pode 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. $\langle \text{var} \rangle.\text{actual_type} \leftarrow \text{look-up}(A)$ (Rule 4)
2. $\langle \text{expr} \rangle.\text{expected_type} \leftarrow \langle \text{var} \rangle.\text{actual_type}$ (Rule 1)
3. $\langle \text{var} \rangle[2].\text{actual_type} \leftarrow \text{look-up}(A)$ (Rule 4)
 $\langle \text{var} \rangle[3].\text{actual_type} \leftarrow \text{look-up}(B)$ (Rule 4)
4. $\langle \text{expr} \rangle.\text{actual_type} \leftarrow \text{ou int ou real}$ (Rule 2)
5. $\langle \text{expr} \rangle.\text{expected_type} == \langle \text{expr} \rangle.\text{actual_type}$
é ou TRUE ou FALSE (Regra 2)

A árvore na Figura 3.7 mostra o fluxo dos valores de atributo no exemplo da Figura 3.6. Linhas sólidas são usadas para a árvore de análise sintática; linhas tracejadas mostram o fluxo de atributos na árvore.

A árvore na 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.

3.4.7 Avaliação

Verificar as regras de semântica estática de uma linguagem é uma parte essencial de todos os compiladores. Mesmo se um desenvolvedor de compiladores nunca ouviu falar de uma gramática de atributos, ele pode precisar usar as ideias fundamentais dessas gramáticas para projetar as verificações de regras de semântica estática para seu compilador.

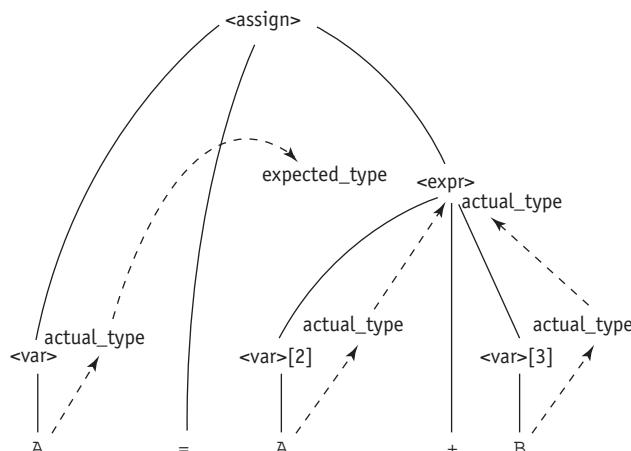


Figura 3.7 O fluxo de atributos na árvore.

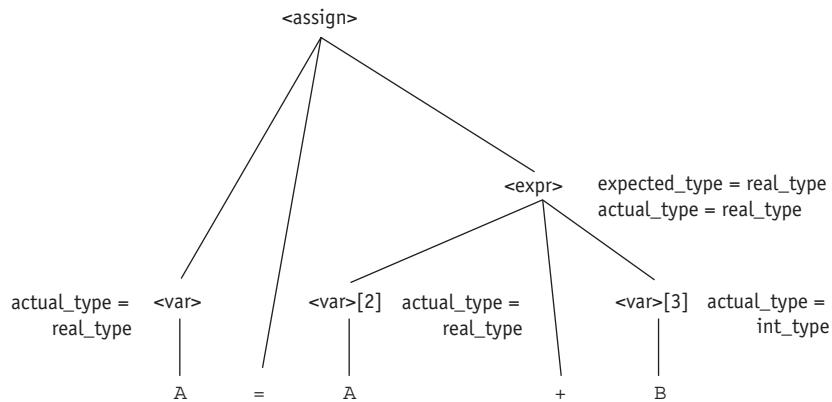


Figura 3.8 Uma árvore de análise sintática completamente atribuída.

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 seu tamanho e complexidade. 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 custosos 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 DESCREVENDO O 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. Por causa do poder e da 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 a respeito da necessidade de uma metodologia e notação para descrever semântica. Os programadores obviamente precisam saber exatamente o que as sentenças de uma linguagem fazem antes de usá-las em seus programas. Desenvolvedores de compiladores devem saber o que as construções da linguagem significam para projetar implementações corretas para elas. 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 desenvolveriam 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 são raramente 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 ele foca 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 rodá-lo em uma máquina. Os efeitos na máquina são vistos como sequências de mudanças em seu estado, onde 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 a pessoa está fazendo é usando semântica operacional para determinar o significado da construção.

Existem diversos problemas com o 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 é 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 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 muito baixo nível 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 manualmente.

O processo básico da semântica operacional é usual. O conceito é bastante usado em livros-texto de programação e 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

<i>Sentença C</i>	<i>Significado</i>
<code>for (expr1; expr2; expr3) {</code>	<code>expr1;</code>
...	<code>loop: if expr2 == 0 goto out</code>
<code>}</code>	<code>...</code>
	<code>expr3;</code>
	<code>goto loop</code>
	<code>out: . . .</code>

O leitor humano de tal descrição é o computador virtual e assume que ele seja capaz de “executar” corretamente as instruções na definição e reconhecer os efeitos da “execução”.

A linguagem intermediária e sua máquina virtual associada usada para as descrições formais de semântica operacional são, com frequência, 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 des-

crever 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 leve generalização dessas três sentenças de atribuição permite que expressões aritméticas e sentenças de atribuição mais gerais sejam descritas.

```
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 usando essa 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 do PL/I, infelizmente, é tão complexa que não serve para nenhum propósito prático.

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 mais baixo nível. Essa abordagem pode levar a circularidades, em que conceitos são indiretamente definidos em termos deles próprios. Os métodos descritos nas duas seções seguintes são muito mais formais, no sentido de que eles são baseados em matemática e lógica, não em linguagens de programação.

3.5.2 Semântica denotacional

A **semântica denotacional** é o método 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 deno-

tacional para descrever a semântica de linguagens de programação é 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 qual as construções de linguagem de programação são traduzidas para construções mais simples, o que se torna 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

Usamos uma construção de linguagem muito simples, a representação em cadeias de caracteres de números binários, para introduzir o método denotacional. 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 apóstrofes 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 para um valor matemático antes de poder ser usada como um valor no programa.

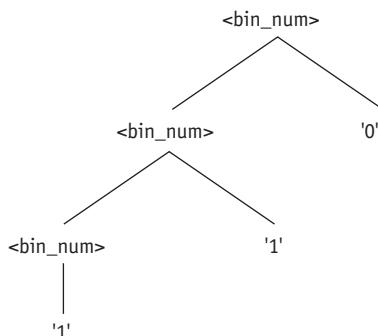


Figura 3.9 Uma árvore de análise sintática do número binário 110.

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 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 com as duas primeiras regras gramaticais. As outras duas são, em um sentido, regras computacionais, porque combinam um símbolo terminal, com o qual um objeto pode ser associado, com um não terminal, para 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, tal regra sintática precisaria de uma função que computasse o significado do LHS, o que 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:

$$\begin{aligned}
 M_{\text{bin}}('0') &= 0 \\
 M_{\text{bin}}('1') &= 1 \\
 M_{\text{bin}}(<\text{bin_num}> '0') &= 2 * M_{\text{bin}}(<\text{bin_num}>) \\
 M_{\text{bin}}(<\text{bin_num}> '1') &= 2 * M_{\text{bin}}(<\text{bin_num}>) + 1
 \end{aligned}$$

Os significados, ou objetos denotados (que, nesse caso, são números decimais), podem ser anexados aos nós da árvore de análise sintática acima, resultando na árvore da Figura 3.10. Isso é uma semântica dirigida por sintaxe. As entidades sintáticas são mapeadas para objetos matemáticos com significado concreto.

Em parte porque precisaremos disso depois, mostraremos um exemplo similar para descrever o significado de literais decimais sintáticos. Nesse caso,

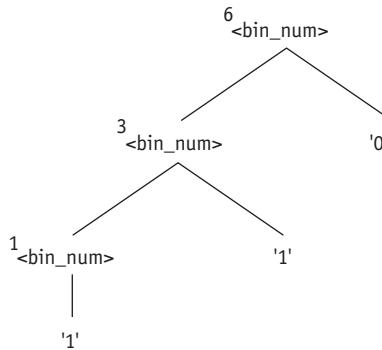


Figura 3.10 Uma árvore de análise sintática com objetos denotados para 110.

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.

$$\begin{aligned} \langle \text{dec_num} \rangle \rightarrow & '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' \\ & | \langle \text{dec_num} \rangle ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9') \end{aligned}$$

Os mapeamentos denotacionais para essas regras sintáticas são

$$\begin{aligned} M_{\text{dec}}('0') &= 0, M_{\text{dec}}('1') = 1, M_{\text{dec}}('2') = 2, \dots, M_{\text{dec}}('9') = 9 \\ M_{\text{dec}}(\langle \text{dec_num} \rangle '0') &= 10 * M_{\text{dec}}(\langle \text{dec_num} \rangle) \\ M_{\text{dec}}(\langle \text{dec_num} \rangle '1') &= 10 * M_{\text{dec}}(\langle \text{dec_num} \rangle) + 1 \\ \dots \\ M_{\text{dec}}(\langle \text{dec_num} \rangle '9') &= 10 * M_{\text{dec}}(\langle \text{dec_num} \rangle) + 9 \end{aligned}$$

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 algorit-

mos 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:

$$s = \{ \langle i_1, v_1 \rangle, \langle i_2, v_2 \rangle, \dots, \langle i_n, v_n \rangle \}$$

Cada i é o nome de uma variável, e os v s associados são os valores atuais dessas variáveis. Qualquer um desses v s podem ter o valor especial **undef**, que indica 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 (ij, s) é vj (o valor que faz par com ij 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. Assumimos 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 $Z \cup \{\text{error}\}$ 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 as sentenças de linguagens de programação, usamos o símbolo Δ 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, $<\text{binary_expr}>. <\text{left_expr}>$ se refere ao nó filho da esquerda de $<\text{binary_expr}>$.

```


$$M_e(<\text{expr}>, s) \Delta= \begin{cases} \text{case } <\text{expr}> \text{ of} \\ \quad <\text{dec\_num}> \Rightarrow M_{\text{dec}}(<\text{dec\_num}>, s) \\ \quad <\text{var}> \Rightarrow \text{if } \text{VARMAP}(<\text{var}>, s) == \text{undef} \\ \quad \quad \quad \text{then error} \\ \quad \quad \quad \text{else } \text{VARMAP}(<\text{var}>, s) \\ \quad <\text{binary\_expr}> \Rightarrow \\ \quad \quad \text{if } (M_e(<\text{binary\_expr}>. <\text{left\_expr}>, s) == \text{undef} \text{ OR} \\ \quad \quad M_e(<\text{binary\_expr}>. <\text{right\_expr}>, s) == \text{undef}) \\ \quad \quad \text{then error} \\ \quad \quad \text{else if } (<\text{binary\_expr}>. <\text{operator}> == '+') \\ \quad \quad \quad \text{then } M_e(<\text{binary\_expr}>. <\text{left\_expr}>, s) + \\ \quad \quad \quad M_e(<\text{binary\_expr}>. <\text{right\_expr}>, s) \\ \quad \quad \text{else } M_e(<\text{binary\_expr}>. <\text{left\_expr}>, s) * \\ \quad \quad \quad M_e(<\text{binary\_expr}>. <\text{right\_expr}>, s) \end{cases}$$


```

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:

```


$$M_a(x = E, s) \Delta= \begin{cases} \text{if } M_e(E, s) == \text{error} \\ \quad \text{then error} \\ \quad \text{else } s' = \{<i_1, v_1'\>, <i_2, v_2'\>, \dots, <i_n, v_n'\>, \text{ where} \\ \quad \quad \text{for } j = 1, 2, \dots, n \\ \quad \quad \quad \text{if } i_j == x \\ \quad \quad \quad \text{then } v_j' = M_e(E, s) \\ \quad \quad \quad \text{else } v_j' = \text{VARMAP}(i_j, s) \end{cases}$$


```

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, assumimos 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

```


$$M_l(\text{while } B \text{ do } L, s) \Delta= \begin{cases} \text{if } M_b(B, s) == \text{undef} \\ \quad \text{then error} \\ \quad \text{else if } M_b(B, s) == \text{false} \\ \quad \quad \text{then } s \\ \quad \quad \text{else if } M_{sl}(L, s) == \text{error} \\ \quad \quad \quad \text{then error} \\ \quad \quad \quad \text{else } M_l(\text{while } B \text{ do } L, M_{sl}(L, s)) \end{cases}$$


```

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, assumindo que não ocorreram erros. Essencialmente, o laço foi convertido de iteração para recursão, onde o controle da recursão é definido matematicamente por outras funções recursivas de mapeamento de estado. A recursão é mais fácil de descrever com rigor matemático do que a iteração.

Uma observação importante neste momento é que essa definição, como laços de programas reais, podem não computar coisa alguma por causa da não finalização.

3.5.2.6 Avaliação

Objetos e funções, como aquelas usadas nas construções anteriores, podem ser definidas 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 *framework* para pensar sobre programação de uma maneira altamente rigorosa.

Conforme mencionado, a semântica denotacional pode ser usada como uma ajuda para o projeto de linguagens. Por exemplo, sentenças para as quais a descrição em semântica denotacional seja complexa e difícil pode indicar ao projetista que tais sentenças também serão difíceis para que os usuários as entendam e que um projeto alternativo pode ser necessário.

Devido à complexidade das descrições denotacionais, elas são de pouco uso para os usuários das linguagens. No entanto, elas 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 ser rastreada e acompanhada desde o século XIX (Frege, 1892).

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 de 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 de mudanças de estado que ocorrem quando o programa é executado. O significado de um programa é baseado nos relacionamentos entre variáveis e constantes de um programa, os quais são o mesmo 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 foca na verificação de programas e na descrição de semântica axiomática.

A semântica axiomática foi definida em conjunto com o desenvolvimento de uma abordagem 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 precede imediatamente uma sentença de programa descreve as restrições nas variáveis do programa naquele ponto. Uma asserção imediatamente após uma sentença descreve as novas restrições nessas variáveis (e possivelmente em outras) após a execução 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 que pré-condições para sentenças são computadas para pós-condições dadas, apesar de ser possível considerá-las no sentido oposto. Assumimos que todas as variáveis são do tipo inteiro. Como um exemplo simples, considere a seguinte sentença de atribuição e a pós-condição:

```
sum = 2 * x + 1 {sum > 1}
```

As asserções de pré e pós-condições são apresentadas em chaves para distinguí-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 seguintes subseções, focamos em provas de corretude de sentenças e de programas, que é um uso de semântica axiomática. O conceito 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 dessa 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 a 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.

Uma **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

$$\frac{S_1, S_2, \dots, S_n}{S}$$

a qual define que se S_1, S_2, \dots, S_n forem verdadeiras, a verdade de S pode ser inferida. 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, um axioma ou uma regra de inferência deve estar disponível 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 assumimos 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 exatamente o seu significado. Para 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, P , é definida pelo axioma

$$P = Q_{x \rightarrow E}$$

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:

$$\begin{aligned} b / 2 - 1 &< 10 \\ b &< 22 \end{aligned}$$

Logo, a pré-condição mais fraca para a sentença de atribuição dada e sua pós-condição é $\{b < 22\}$. Lembre que o axioma de atribuição é garantidamente 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

$$\{Q_{x \rightarrow E}\} x = E \{Q\}$$

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:

$$\begin{aligned} 2 * y - 3 &> 25 \\ y &> 14 \end{aligned}$$

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

$$\begin{aligned}x + y - 3 &> 10 \\y &> 13 - x\end{aligned}$$

Lembre-se de que a semântica axiomática foi desenvolvida para provar a corretude de programas. Nesse sentido, é natural nesse ponto imaginarmos como o axioma para sentenças de atribuição pode ser usado para provar alguma coisa. Aqui está como isso pode ser feito: 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 sentença lógica

$$\{x > 3\} \quad x = x - 3 \quad \{x > 0\}$$

Usando o axioma de atribuição em

$$x = x - 3 \quad \{x > 0\}$$

produz $\{x > 3\}$, que é a pré-condição dada. Logo, provamos a sentença lógica de exemplo.

A seguir, considere a sentença lógica

$$\{x > 5\} \quad x = x - 3 \quad \{x > 0\}$$

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, uma regra de inferência, chamada de regra de consequência, é necessária. A forma da regra de consequência é

$$\frac{\{P\} S \{Q\}, P' \Rightarrow P, Q \Rightarrow Q'}{\{P'\} S \{Q'\}}$$

O símbolo \Rightarrow significa “implica”, e S pode ser qualquer sentença de um programa. A regra pode ser descrita assim: se a sentença lógica $\{P\} S \{Q\}$ for verdadeira, a asserção P' implica na asserção P , e a asserção Q implica na asserção Q' . Então, pode-se inferir 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

$$\frac{\{x > 3\} \quad x = x - 3 \quad \{x > 0\}, \quad (x > 5) \Rightarrow \{x > 3\}, \quad (x > 0) \Rightarrow (x > 0)}{\{x > 5\} \quad x = x - 3 \quad \{x > 0\}}$$

O primeiro termo do antecedente ($\{x > 3\} \quad x = x - 3 \quad \{x > 0\}$) foi provado com o axioma de atribuição. O segundo e terceiro termos são óbvios. Logo, pela regra de 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 a pré-condição depende dos tipos de sentenças particulares na sequência. Nesse caso, a pré-condição apenas 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 e pós-condições

$$\begin{array}{l} \{P1\} S1 \{P2\} \\ \{P2\} S2 \{P3\} \end{array}$$

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 obter a pré-condição da sequência, a pré-condição da segunda deve ser computada. Essa nova assertão 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

$$\begin{array}{l} \{P3_{x2 \rightarrow E2}\} x2 = E2 \{P3\} \\ \{(P3_{x2 \rightarrow E2})_{x1 \rightarrow E1}\} x1 = E1 \{P3_{x2 \rightarrow E2}\} \end{array}$$

Logo, a pré-condição mais fraca para a sequência $x1 = E1; x2 = E2$ com a pós-condição P3 é $\{(P3_{x2 \rightarrow E2})_{x1 \rightarrow E1}\}$.

Por exemplo, considere a seguinte sequência e pós-condição:

$$\begin{array}{l} y = 3 * x + 1; \\ x = y + 3; \\ \{x < 10\} \end{array}$$

A pré-condição para a segunda sentença de atribuição é

$y < 7$

a qual é usada como a pós-condição para a primeira sentença: a pré-condição para a primeira sentença de atribuição pode agora ser computada:

$$\begin{array}{l} 3 * x + 1 < 7 \\ x < 2 \end{array}$$

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

$$\frac{\{B \text{ and } P\} S1 \{Q\}, \{(not B) \text{ and } P\} S2 \{Q\}}{\{P\} \text{ if } B \text{ then } S1 \text{ else } S2 \{Q\}}$$

Essa regra indica que as sentenças de seleção devem ser provadas tanto quando a expressão booleana de controle for verdadeira quanto quando 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 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**

$y = y - 1 \{y > 0\}$

NOTA HISTÓRICA

Uma quantidade significativa de trabalho tem sido feita considerando a possibilidade de usar descrições denotacionais de linguagens para gerar compiladores automaticamente (Jones, 1980; Milos et al., 1984; Bodwin et al., 1982). Esses esforços mostram que o método é factível, mas o trabalho nunca progrediu até um ponto no qual pudesse ser usado para gerar compiladores úteis.

Isso produz $\{y - 1 > 0\}$ ou $\{y > 1\}$, que 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 $\{y > 1\} \Rightarrow \{y > -1\}$, 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 são os pré-testes lógicos, ou laços

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 pré-determinado. Em um caso onde 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 induutivo pode ser usado para alguns laços. O passo principal na indução é encontrar uma hipótese induutiva. 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

$$\frac{(\text{I and B}) (\text{S}\{\text{I}\})}{\{\text{I}\} \text{ while B do S end } \{\text{I and (not B)}\}}$$

onde I é a invariante de laço de repetição. Isso parece simples, mas não é. A complexidade está em encontrar um invariante de laço de repetição apropriado.

A descrição axiomática de um laço **while** é escrita como

$\{\text{P}\} \text{ while B do S end } \{\text{Q}\}$

A invariante de laço de repetição deve satisfazer alguns requisitos para ser útil. Primeiro, a pré-condição mais fraca para o **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 movem 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 de *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:

$\text{P} \Rightarrow \text{I}$
 $\{\text{I and B}\} \text{ S } \{\text{I}\}$
 $(\text{I and (not B)}) \Rightarrow \text{Q}$
the loop terminates

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 induutiva 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, com a esperança 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(statement, postcondition) = precondition`

Uma função `wp` é geralmente chamada de um **transformador de predicado**, porque recebe um predicado, ou asserção, como um parâmetro e retorna outro.

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 zero iterações, a pré-condição mais fraca é, obviamente,

```
{y = x}
```

Para uma iteração, é

```
wp(y = y + 1, {y = x}) = {y + 1 = x}, ou {y = x - 1}
```

Para duas iterações, é

```
wp(y = y + 1, {y = x - 1}) = {y + 1 = x - 1}, ou {y = x - 2}
```

Para três iterações, é

```
wp(y = y + 1, {y = x - 2}) = {y + 1 = x - 2}, ou {y = x - 3}
```

Está agora óbvio que $\{y < x\}$ será suficiente para os casos de uma ou mais iterações. Combinando isso com $\{y = x\}$ para o caso de zero iterações, temos $\{y \leq x\}$, 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 $P = I$, $P \Rightarrow I$. O segundo requisito é que deve ser verdade que

$\{I \text{ and } B\} S \{I\}$

Em nosso exemplo, temos

$$\{y \leq x \text{ and } y < x\} \quad y = y + 1 \quad \{y \leq x\}$$

Aplicando o axioma de atribuição a

$$y = y + 1 \quad \{y \leq x\}$$

obtemos $\{y + 1 \leq x\}$, que é equivalente a $\{y < x\}$, o que é implicado por $\{y \leq x \text{ and } y < x\}$. Logo, a sentença anterior está provada.

A seguir, devemos ter

$$\{I \text{ and } (\text{not } B)\} \Rightarrow Q$$

Em nosso exemplo, temos

$$\begin{aligned} \{(y \leq x) \text{ and not } (y < x)\} &\Rightarrow \{y = x\} \\ \{(y \leq x) \text{ and } (y = x)\} &\Rightarrow \{y = x\} \\ \{y = x\} &\Rightarrow \{y = x\} \end{aligned}$$

Logo, isso é obviamente verdade. A seguir, o término do laço deve ser considerado. Nesse exemplo, a questão é se o laço

$$\{y \leq x\} \text{ while } y < x \text{ do } y = y + 1 \text{ end } \{y = x\}$$

termina. Lembrando que se presume que x e y são variáveis inteiros, 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 quanto 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 usando a abordagem usada na indução matemática, considere a seguinte sentença de laço:

$$\text{while } s > 1 \text{ do } s = s / 2 \text{ end } \{s = 1\}$$

Como antes, usamos o axioma de atribuição para tentar encontrar uma invariante e uma pré-condição para o laço. Para zero iterações, a pré-condição mais fraca é $\{s = 1\}$. Para uma iteração, ela é

$$\text{wp}(s = s / 2, \{s = 1\}) = \{s / 2 = 1\}, \text{ ou } \{s = 2\}$$

Para duas iterações, é

$$\text{wp}(s = s / 2, \{s = 2\}) = \{s / 2 = 2\}, \text{ ou } \{s = 4\}$$

Para três iterações, é

$\text{wp}(s = s / 2, \{s = 4\}) = \{s / 2 = 4\}$, ou $\{s = 8\}$

A partir desses casos, podemos ver claramente que a invariante é

$\{s \text{ é uma potência não negativa de } 2\}$

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

$\{s > 1\} \text{ while } s > 1 \text{ do } s = s / 2 \text{ end } \{s = 1\}$

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. Por causa da regra de consequência, usar uma pré-condição mais forte do 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.

Por causa da 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, provar a corretude de programas com laços **while** usando semântica axiomática pode ser difícil.

3.5.3.7 Provas de programas

Esta seção fornece validações para dois programas simples. O primeiro exemplo de uma 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, que leva a

```
{y = B AND x = A}
```

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 factorial.

```
{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 criatividade é necessária aqui, a qual pode ser auxiliada por um breve estudo do código. O laço computa a função factorial na ordem da última multiplicação primeiro; ou seja, $(n - 1) * n$ é feita primeiro, assumindo que n é maior que 1. Então, parte da invariante pode ser

```
fact = (count + 1) * (count + 2) * . . . * (n - 1) * n
```

Mas também precisamos nos assegurar de que `count` é sempre não negativo, o que podemos fazer adicionando isso à asserção acima, para obter

```
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 em I . A próxima questão é

$\{I \text{ and } B\} S \{I\}$

$I \text{ and } B$ is

```
((fact = (count + 1) * . . . * n) AND (count >= 0)) AND  
(count <> 0)
```

que é reduzida para

```
(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-a 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 nesse P . Então, pela regra de consequência,

$\{I \text{ AND } B\} S \{I\}$

é verdadeira. Finalmente, o último teste de I é

$I \text{ AND } (\text{NOT } B) \Rightarrow Q$

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 factorial. 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-a como a pós-condição para a primeira atribuição no código

```
{P} count = n { (1 = (count + 1) * . . . * n) AND  
 (count >= 0) }
```

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, $\{n \geq 0\}$. Logo, provamos que programa está correto.

3.5.3.8 Avaliação

Para definir a semântica de uma linguagem de programação completa usando o 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 provou ser uma tarefa difícil. Uma solução óbvia para esse problema é projetar a linguagem com o método axiomático em mente, de forma que apenas sentenças para as quais axiomas ou regras de inferência podem ser escritos são incluídas. Infelizmente, tal linguagem seria bastante pequena e simples, dado o estado da ciência da semântica axiomática.

A semântica axiomática é uma ferramenta poderosa para a pesquisa na área de prova de corretude de programas, e ela fornece um *framework* excelente no qual se pode pensar acerca de programas, tanto durante sua construção quanto posteriormente. Sua utilidade em descrever o significado de linguagens de programação tanto para os usuários de linguagens quanto para os desenvolvedores de compiladores é, entretanto, 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 apenas são métodos descritivos concisos, mas as árvores de análise sintática que podem ser associadas com suas ações de geração dão 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, os quais descrevem as regras de semântica estática.

Existem três métodos principais de descrição semântica: operacional, denotacional e axiomática. 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 para esses objetos matemáticos por meio de funções recursivas. Semântica axiomática, baseada em lógica formal, foi criada como uma ferramenta para provar a corretude de programas.

NOTAS BIBLIOGRÁFICAS

A descrição de sintaxe usando gramáticas livres de contexto e BNF é extensamente discutida em Cleaveland e Uzgalis (1976).

A pesquisa em semântica axiomática começou com Floy (1967) e foi complementada por Hoare (1969). A semântica de uma grande parte de Pascal foi descrita por Hoare e Wirth (1973) usando esse método. As partes que eles não completaram envolviam efeitos colaterais de funções e sentenças **goto**. Essas foram as partes 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 aos três métodos de descrição semântica discutidos neste capítulo podem ser encontradas em Marcotty et al. (1976). Outra boa referência para muito do material deste capítulo é Pagan (1981). A forma das funções de semântica denotacional deste capítulo é similar à usada 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 gramática recursiva à esquerda.
7. Que 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 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 o principal uso das gramáticas de atributos?
13. Explique os principais usos de uma 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. Que 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 em 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 abordagem 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 se diferem?

CONJUNTO DE 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 + precedência sobre * e para forçar + a ser associativo à direita.
4. Reescreva a BNF do Exemplo 3.4 para adicionar os operadores unários de Java **++** e **--**.
5. Escreva uma descrição BNF das expressões booleanas de Java, incluindo os três operadores **&&**, **||** e **!** e as expressões relacionais.
6. Usando a gramática no Exemplo 3.2, mostre uma árvore de análise sintática e uma derivação mais a esquerda para cada uma das seguinte sentenças:
 - a. **A = A * (B + (C * A))**
 - b. **B = C * (A * C + B)**
 - c. **A = A * (B + (C))**
7. Usando a gramática no Exemplo 3.4, mostre uma árvore de análise sintática e uma derivação mais a esquerda para cada uma das seguinte 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:
- $$\begin{aligned} <S> &\rightarrow <A> \\ <A> &\rightarrow <A> + <A> \mid <\text{id}> \\ <\text{id}> &\rightarrow a \mid b \mid c \end{aligned}$$
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:
- $$\begin{aligned} <S> &\rightarrow <A> <C> \\ <A> &\rightarrow a <A> \mid a \\ &\rightarrow b \mid b \\ <C> &\rightarrow c <C> \mid c \end{aligned}$$
11. Considere a seguinte gramática:
- $$\begin{aligned} <S> &\rightarrow <A> a b \\ <A> &\rightarrow <A> b \mid b \\ &\rightarrow a \mid a \end{aligned}$$
- Quais das sentenças abaixo estão na linguagem gerada por essa gramática?
- baab
 - bbbbab
 - bbaaaaaa
 - bbaaab
12. Considere a seguinte gramática:
- $$\begin{aligned} <S> &\rightarrow a <S> c \mid <A> \mid b \\ <A> &\rightarrow c <A> \mid c \\ &\rightarrow d \mid <A> \end{aligned}$$
- Quais das sentenças abaixo estão na linguagem gerada por essa gramática?
- abcd
 - acccbd
 - acccbcc
 - acd
 - accc
13. Escreva uma gramática para a linguagem com cadeias que têm n cópias da letra a seguida pelo mesmo número de cópias da letra b, onde $n > 0$. Por exemplo, as cadeias ab, aaaabbbb, eaaaaaaaaabbbbbbbb 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 para EBNF.
16. Converta a BNF do Exemplo 3.3 para EBNF.
17. Converta a seguinte EBNF para BNF:
- $$\begin{aligned} S &\rightarrow A \{ bA \} \\ A &\rightarrow a [b]A \end{aligned}$$
18. Qual é a diferença entre um atributo intrínseco e um atributo sintetizado não intrínseco?
19. Escreva uma gramática de atributo cuja base da BNF é aquela do Exemplo 3.6 na Seção 3.4.5, mas cujas regras da linguagem são as seguintes: os tipos de dados

- não podem ser misturados em expressões, mas as sentenças de atribuição não precisam ter os mesmos tipos em ambos os lados do operador de atribuição.
20. Escreva uma gramática de atributo cuja base da BNF é aquela do Exemplo 3.2 e cujas regras de tipo são 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, dê uma definição de 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 para provar a corretude de um laço de repetição com pré-teste lógico da forma **while** B **do** S **end**
27. Prove que $(n + 1)^* \dots ^* n = 1$
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}
```

Capítulo 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

Uma investigação séria do projeto de compiladores requer ao menos um semestre de estudo intensivo, incluindo o projeto e a implementação de um compilador para uma linguagem de programação simples, mas realista. A primeira parte de tal curso é dedicada à análise 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 análise léxica e sintática neste livro. Primeiro, os analisadores sintáticos são baseados diretamente nas gramáticas discutidas no Capítulo 3, então 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, dentre 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 tanto análise léxica quanto sintática. Logo, as análises léxica e sintática são tópicos importantes para os desenvolvedores de software, mesmo que nunca precisem escrever um compilador. Além disso, alguns cursos de ciência da computação não requerem mais que os estudantes 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.

Este capítulo começa com uma introdução à análise léxica, com um exemplo simples. Então, o problema geral de análise sintática é discutido, incluindo as duas principais abordagens para análise sintática e a complexidade do processo. A seguir, introduzimos 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 usando o processo de análise sintática LR.

4.1 INTRODUÇÃO

O Capítulo 1 introduz três abordagens para a implementação de linguagens de programação: compilação, interpretação pura e implementação híbrida. A abordagem 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* embarcados 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, graças à popularidade de Java e uma coleção de linguagens de *scripting* recentes. Tradicionalmente, sistemas híbridos têm resultado na execução muito mais lenta de programas do que nos 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, o qual traduz código intermediário para 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.

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, introduzida no Capítulo 3. Usar 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 a 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 diversas razões pelas quais a análise léxica é separada da sintática:

1. Simplicidade – Técnicas para análise léxica são menos complexas do que as necessárias para análise sintática, então 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 analisador sintático menor e mais limpo.
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 essencialmente um casador de padrões, que 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 de casamento de padrões foi 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 encontrou seu caminho 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 é informado para um compilador como sendo uma única cadeia de caracteres. O analisador léxico coleta caracteres e os agrupa logicamente, atribuindo código 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 uma sentença de atribuição:

```
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 primeiros tempos dos compiladores, os analisadores léxicos processavam um arquivo de programa de entrada completamente e produziam um arquivo de *tokens* e *lexemas*. Agora, entretanto, a

maioria dos analisadores léxicos são subprogramas que localizam o próximo lexema na entrada, determinam seu código de *token* associado e o retornam para o chamador, o analisador sintático. A única visão do programa de entrada vista 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 eles 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. Finalmente, analisadores léxicos detectam erros sintáticos em *tokens*, como literais de ponto flutuante mal formados, e informam tais erros ao usuário.

Existem três abordagens para construir um analisador léxico:

1. Escrever uma descrição formal dos padrões de *tokens* da linguagem usando uma linguagem descritiva relacionada a expressões regulares¹. Essas descrições são usadas como entrada para uma ferramenta de software que gera automaticamente um analisador léxico. Existem muitas ferramentas disponíveis para isso. A mais antiga e acessível dentre essas, 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 implementa 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 também incluir ações que o analisador léxico deve realizar quando a transição ocorrer.

Diagramas de estados da forma usada pelos analisadores léxicos são representações de uma classe de máquinas matemáticas chamadas de **autômatos finitos**. Autômatos finitos podem ser projetados para reconhecer uma classe de linguagens chamadas de **linguagens regulares**. Gramáticas regulares são dispositivos geradores para 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 simplesmente incluir estados e transições para cada um dos padrões de *tokens*. Entretanto, os resultados dessa abordagem são diagramas muito grandes e

¹ Essas expressões regulares são a base para os recursos de casamento de padrões que agora fazem parte de muitas linguagens de programação, seja diretamente ou por meio de uma biblioteca de classes.

complexos, já que cada nó no diagrama de estados precisaria de uma transição para cada caractere no conjunto de caracteres da linguagem que está sendo analisada. Verificaremos maneiras para simplificar isso.

Suponha que precisássemos de um analisador léxico que reconhecesse apenas expressões aritméticas, incluindo nomes de variáveis e literais inteiros como operandos. Os nomes de variáveis consistem em cadeias de letras maiúsculas, letras minúsculas e dígitos, mas devem começar com uma letra. Os nomes não têm limitação de tamanho. A primeira observação a ser feita é que existem 52 caracteres diferentes (quaisquer letras maiúsculas ou minúsculas) que podem começar um nome, o que requereria 52 transições a partir do estado inicial do diagrama de transições. Entretanto, um analisador léxico está interessado apenas em determinar que é um nome, sem se preocupar com qual nome específico é. Logo, definimos uma classe de caracteres chamada LETTER (letra) para todas as 52 letras e usamos uma transição no primeiro caractere de qualquer nome.

Outra oportunidade para simplificar o diagrama de transição é com os *tokens* representando literais inteiros. Existem 10 caracteres que podem iniciar um *lexema* de literal inteiro. Isso iria requerer 10 transições a partir do estado inicial do diagrama de 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.

Observe que a maioria das linguagens de programação permite dígitos em nomes de programas, após a letra inicial. Para a transição a partir do nó seguinte ao primeiro caractere de um nome, podemos 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, que 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`, como caracteres de espaço em branco entre lexemas em nossa linguagem. 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 causa disso, uma função chamada `getNextBlank` é usada para ignorar espaços em branco.

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 na 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 na Figura 4.1, incluindo uma função principal para propósitos de teste:

```
/* front.c - um analisador léxico e analisador sintático
   simples para expressões aritméticas simples */

#include <stdio.h>
#include <ctype.h>

/* Declarações globais */
/* Variáveis */

int charClass;
char lexeme [100];
char nextChar;
int lexLen;
int token;
```

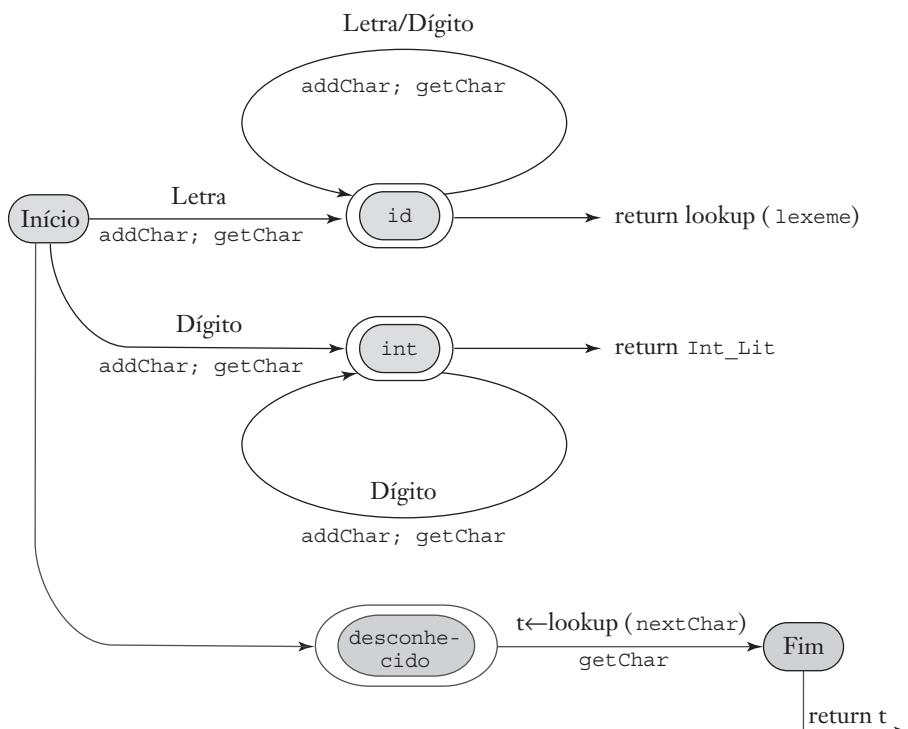


Figura 4.1 Um diagrama de estados para reconhecer nomes, parênteses e operadores aritméticos.

```
int nextToken;
FILE *in_fp, *fopen();

/* Declarações de Funções */
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() {

    /* Abrir o arquivo de dados de entrada e processar 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;
    }
}
```

```

        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 ao
   vetor 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 da en-
   trada e determinar sua classe de caracteres */
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 ela
   retorne um caractere diferente de espaço em
   branco */
void getNonBlank() {
    while (isspace(nextChar))
        getChar();
}

/
/*****************/
/* lex - um analisador léxico simples para expressões
   aritméticas */
int lex() {
    lexLen = 0;
    getNonBlank();
    switch (charClass) {

        /* Reconhecer identificadores */
        case LETTER:
            addChar();
            getChar();
            while (charClass == LETTER || charClass == DIGIT) {
                addChar();
                getChar();
            }
            nextToken = IDENT;
            break;

        /* Reconhecer literais inteiros */
        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 seguinte expressão:

$(\text{sum} + 47) / \text{total}$

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 proibitivamente 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 que nomes são palavras reservadas. O uso dessa abordagem considera as palavras reservadas exceções na categoria de nomes de *tokens*.

Um analisador léxico geralmente é responsável pela construção inicial da tabela de símbolos, que age 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 no-

mes. Por exemplo, se um nome for de uma variável, o tipo da variável é um dos atributos do nome que será armazenado na tabela de símbolos. Os nomes são normalmente 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 introduz 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 programas informados. Em alguns casos, a árvore de análise sintática é apenas construída implicitamente, significando que talvez apenas um percurso da árvore seja gerado. Mas, em todos os casos, as informações necessárias para criar a árvore de análise sintática são criadas durante a análise. Tanto árvores de análise sintática quanto derivações incluem todas as informações sintáticas necessárias para um processador de linguagem.

Existem dois objetivos distintos da análise sintática: primeiro, verificar o programa de entrada para determinar se ele está sintaticamente correto. Quando um erro for encontrado, o analisador deve produzir uma mensagem de diagnóstico e se recuperar. A recuperação significa que ele deve 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 criar 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 categorizados de acordo com a direção na qual eles constroem as árvores de análise sintática. As duas amplas classes de analisadores sintáticos são os analisadores **descendentes**, nos quais a árvore é construída a partir da raiz em direção às folhas, e **ascendentes**, a partir das folhas em direção à raiz.

Neste capítulo, usaremos um pequeno conjunto de convenções de notação para símbolos gramaticais e cadeias para tornar a discussão menos poluída. Para linguagens formais, as convenções 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 ($\alpha, \beta, \delta, \gamma$).

Para linguagens de programação, os símbolos terminais são as construções sintáticas de pequena escala, a que 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 usados 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. Isso corresponde a uma derivação mais à esquerda. Um percurso em pré-ordem de uma árvore de análise sintática começa com a raiz. Cada nó é visitado antes de seus ramos serem seguidos. Os ramos de um nó em particular são seguidos na ordem da esquerda para direita.

Em termos de derivação, um analisador sintático descendente pode ser descrito como segue. Dada uma forma sentencial que é 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 esquerda é $xA\alpha$, onde, 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 corrente é

$xA\alpha$

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 $xbB\alpha$, $xcBb\alpha$ ou $xa\alpha$. Esse é o problema de decisão de análise sintática para analisadores sintáticos descendentes.

Diferentes algoritmos de análise sintática descendente usam informações diferentes para tomar decisões de análise sintática. Os analisadores sintáticos descendentes mais comuns escolhem a RHS correta 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. A RHS que tiver esse *token* no final esquerdo da cadeia que ela gera é a correta. Então, na forma sentencial $xA\alpha$, o analisador sintático usaria qual-

quer *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, todas as três RHSs das regras A começam com símbolos terminais diferentes. O analisador sintático pode facilmente escolher a RHS correta baseado no próximo *token* de entrada, que pode ser a, b, ou c nesse exemplo. De um modo geral, a escolha da RHS correta não é tão direta assim, porque algumas das 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 poderosas. Ambas trabalham no mesmo subconjunto de gramáticas. 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 introduz 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. Em termos da derivação, um analisador sintático ascendente pode ser descrito como segue. Dada uma forma sentencial direita α ,² o analisador sintático deve determinar que subcadeia de α é a RHS da regra na gramática deve ser reduzida para seu LHS para 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 a RHS a ser reduzida ao LHS correspondente para obter a penúltima forma sentencial na derivação. O processo de encontrar a RHS correta para reduzir é complicado pelo fato de uma forma sentencial direita poder incluir mais de uma RHS da gramática da linguagem sendo analisada sintaticamente. A RHS correta é chamada de **manipulador (handle)**.

Considere a seguinte gramática e derivação:

$$\begin{aligned} S &\rightarrow aAc \\ A &\rightarrow aA \mid b \end{aligned}$$

$$S \Rightarrow aAc \Rightarrow aaAc \Rightarrow aabc$$

² Uma forma sentencial direita é uma forma sentencial que aparece em uma derivação mais à direita.

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 uma RHS, b. Quando o analisador sintático substitui b por sua 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 diversas 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* na entrada que não foram analisados ainda.

Os algoritmos mais comuns de análise sintática ascendente estão na família LR, onde 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 trabalham 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 que está sendo analisada. A reanálise sintática é necessária quando o analisador sintático tiver cometido um erro no processo de análise. Voltar o analisador sintático também requer que a parte com problemas da árvore de análise sintática que está sendo construída (ou seu percurso) deve ser desmantelada e reconstruída. Os algoritmos $O(n^3)$ normalmente não são úteis para processos práticos, como a análise sintática para um compilador, porque são muito lentos. Nesse tipo de situação, os cientistas da computação buscam algoritmos mais rápidos, apesar de não tão 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 descrevem linguagens de programação (na verdade, conforme discutido no Capítulo 3, a classe inteira das gramáticas livres de contexto não é adequada 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 do que os algoritmos $O(n^3)$.

4.4 ANÁLISE SINTÁTICA DESCENDENTE RECURSIVA

Esta seção introduz 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 uma 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.

A EBNF é idealmente adequada para analisadores sintáticos descendentes recursivos. Lembre-se, do Capítulo 3, de que as principais extensões da EBNF eram as chaves, que especificam que o que está envolto entre elas pode aparecer zero ou mais vezes, e os colchetes, que 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 dois exemplos:

```
<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 na gramática. A responsabilidade do subprograma associado com um não terminal em particular 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 força 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 lin-

guagem. Isso pode ser facilmente feito quando a análise sintática descendente recursiva é usada.

Na seguinte função descendente recursiva de exemplo, `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 uma única RHS é relativamente simples. Para cada símbolo terminal na RHS, esse símbolo é comparado com `nextToken`. Se eles não casam, 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 -, obtenha o próximo
       token e analise 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 tarde 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 assume 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 ou pelo operador de adição ou pelo de subtração. Essa é a linguagem gerada pelo não terminal `<expr>`. Logo, ela primeiro 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 do que a maioria, porque sua regra tem

apenas uma RHS. Além disso, ela não inclui qualquer código para detecção ou recuperação de erros de sintaxe, porque não existem erros detectáveis associados com a regra gramatical.

Um subprograma de análise sintática descendente recursiva para um não terminal cuja regra tenha mais de uma RHS começa com código para determinar qual RHS deve ser analisada sintaticamente. Cada RHS é examinada (em tempo de construção do compilador) para determinar o conjunto de símbolos terminais que podem aparecer no início de sentenças que ela pode gerar. As casar esses conjuntos com o próximo *token* de entrada, o analisador sintático pode escolher a RHS correta.

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 termo */
    factor();

/* Desde que o próximo token seja + ou -, obtenha o próximo
   token e analise sintaticamente o próximo termo */
    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 suas duas 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, uma mensagem de diagnóstico deve ser produzida quando um erro é detectado. 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 a RHS é (<expr>), chame lex para passar o parêntese
esquerdo, chame expr e verifique pelo parêntese
direito */
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 esquerdo */
    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 a 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** de 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() {
/* Certifique-se de que o primeiro token é 'if' */
  if (nextToken != IF_CODE)

```

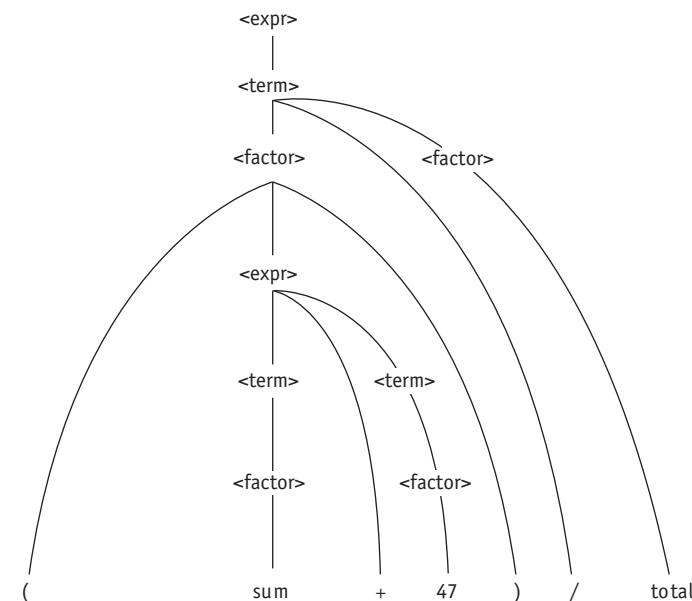


Figura 4.2 Árvore de análise sintática para $(\text{sum} + 47) / \text{total}$.

```

        error();
    else
    {
        /* Chame lex para obter o próximo token */
        lex();
        /* Verifique se existe o parêntese esquerdo */
        if (nextToken != LEFT_PAREN)
            error();
        else {
            /* Chame boolexpr para analisar sintaticamente a expressão
               booleana */
            boolexpr();
            /* Verifique se existe o parêntese direito */
            if (nextToken != RIGHT_PAREN)
                error();
            else {
                /* Chame statement para analisar sintaticamente a cláusula
                   then */
                statement();
                /* Se um else for o próximo, analise sintaticamente a
                   cláusula else */
                if (nextToken == ELSE_CODE) {
                    /* Chame 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, as quais 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 escolher usar a estratégia de análise sintática descendente recursiva para um compilador ou para outra ferramenta de análise de programas, devem ser consideradas as limitações da abordagem, em termos de restrições gramaticais. Esta seção discute tais restrições e 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 sua RHS. A ativação do subprograma A do analisador então imediatamente chama a si mesmo mais uma vez, e novamente, e assim por diante. É fácil ver que isso não leva a nada.

A recursão à esquerda na regra $A \rightarrow A + B$ é chamada de **recursão à esquerda direta** porque ela ocorre em uma regra. A recursão à direita direta pode ser eliminada de uma gramática com o seguinte processo:

Para cada não terminal, A,

1. Agrupe as regras-A como $A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$
onde nenhum dos β 's começa com A
2. Substitua as regras-A originais por

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

Note que ϵ 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 sua LHS da forma sentencial.

Considere a seguinte gramática de exemplo e a aplicação do processo acima:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T^* F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

Para as regras E, temos $\alpha_1 = + T$ e $\beta = T$, então substituímos as regras E por

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \epsilon \end{aligned}$$

Para as regras T, temos $\alpha_1 = * F$ e $\beta = F$, então substituímos as regras T por

$$\begin{aligned} T &\rightarrow F T' \\ T' &\rightarrow * F T' \mid \epsilon \end{aligned}$$

Como não existe recursão à esquerda nas regras F, elas permanecem as mesmas, então a gramática substituta completa é

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \epsilon \\ T &\rightarrow F T' \\ T' &\rightarrow * F T' \mid \epsilon \\ F &\rightarrow (E) \mid id \end{aligned}$$

Essa gramática gera a mesma linguagem que a original, mas não é recursiva à esquerda.

Como era o caso com a gramática de expressões escrita usando EBNF na Seção 4.1.1, 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 coloca o mesmo problema que a recursão à esquerda direta. Por exemplo, suponha que tivéssemos

$$\begin{aligned} A &\rightarrow B \ a \ A \\ B &\rightarrow A \ b \end{aligned}$$

Um analisador sintático descendente recursivo para essas regras faria com que o subprograma A chamassem imediatamente o subprograma para B, que chama o subprograma A. Então, o problema é o mesmo da recursão à esquerda direta. O problema da recursão à esquerda não está confinado 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 coberto 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 desabilita a análise sintática descendente. Outra é se o analisador sintático puder sempre escolher a RHS correta 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 de uma gramática não recursiva à esquerda que indica se isso pode ser feito, o **teste de disjunção par a par**, que requer a habilidade de computar um conjunto com base nos lados direitos de um dado símbolo não terminal em uma gramática. Tais conjuntos, chamados FIRST, são definidos como

$$\text{FIRST}(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\} \ (\text{If } \alpha \Rightarrow^* \epsilon, \epsilon \text{ is in FIRST}(\alpha))$$

No qual \Rightarrow^* 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

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

$$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$$

(A intersecção dos dois conjuntos, $\text{FIRST}(\alpha_i)$ e $\text{FIRST}(\alpha_j)$, deve ser vazia.)

Em outras palavras, se um não terminal A tiver mais de uma RHS, o primeiro símbolo terminal que pode ser gerado em uma derivação para cada uma delas deve ser único àquela RHS. Considere as seguintes regras:

$$\begin{aligned}A &\rightarrow aB \mid bAb \mid Bb \\B &\rightarrow cB \mid d\end{aligned}$$

Os conjuntos FIRST para as 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 vendo apenas o primeiro símbolo terminal da entrada (*token*) gerado pelo não terminal. Agora considere as regras

$$\begin{aligned}A &\rightarrow aB \mid BAab \\B &\rightarrow aB \mid b\end{aligned}$$

Os conjuntos FIRST para as 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 analisada sintaticamente olhando para o próximo símbolo da entrada, porque se ele for um a, pode ser qualquer uma das RHSs. Essa questão é mais complexa se uma 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

$$<\text{variable}> \rightarrow \text{identifier} \mid \text{identifier} [<\text{expression}>]$$

Essa regra diz que uma variável (*<variable>*) é tanto 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 ambas as RHSs começam com o mesmo terminal, chamado *identifier*. Esse problema pode ser aliviado com o processo de **fatoração à esquerda**.

Agora, tomaremos uma visão informal sobre a fatoração à esquerda. Considere nossas regras para *<variable>*. Ambas as RHSs começam com *identifier*. As partes que seguem *identifier* nas duas RHSs são a cadeia vazia ϵ e $[<\text{expression}>]$. As duas regras podem ser substituídas por

$$<\text{variable}> \rightarrow \text{identifier} <\text{new}>$$

onde *<new>* é definido como

$$<\text{new}> \rightarrow \epsilon \mid [<\text{expression}>]$$

Não é difícil ver 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 está sendo usada como base para um analisador sintático descendente recursivo, uma alternativa à fatoração à esquerda está disponível. 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 colocando-o entre colchetes, como em

`<variable> → identifier [[<expression>]]`

Nessa regra, os colchetes externos são metassímbolos que indicam o que está dentro deles como opcional. Os colchetes internos são símbolos terminais da linguagem de programação sendo 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). 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 introduz 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:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

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:

$$\begin{aligned} E &\Rightarrow \underline{E} + T \\ &\Rightarrow E + \underline{T} * F \\ &\Rightarrow E + T * \underline{id} \\ &\Rightarrow E + \underline{F} * id \\ &\Rightarrow E + \underline{id} * id \\ &\Rightarrow T + id * id \\ &\Rightarrow \underline{F} + id * id \\ &\Rightarrow id + id * id \end{aligned}$$

A parte sublinhada de cada forma sentencial nessa derivação é a RHS reescrita como sua LHS correspondente para obter a forma sentencial prévia. O processo de análise sintática ascendente produz o reverso 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 resta o símbolo inicial, que nessa gramática é E. Em cada passo, a tarefa do analisador sintático ascendente é encontrar a RHS específica, o manipulador, na forma sentencial que deve ser reescrita para obter a próxima (prévia) forma sentencial. Conforme mencionado anteriormente, uma forma sentencial à direita pode incluir mais de uma RHS. Por exemplo, a forma sentencial à direita

$E + T^* id$

inclui três RHSs, E + T, T e id. Apenas uma dessas é o manipulador. Por exemplo, se a RHS E + T fosse escolhida para ser reescrita 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 apenas se, $S \Rightarrow_{rm}^* \alpha Aw \Rightarrow_{rm} \alpha\beta w$.

Nessa definição, \Rightarrow_{rm} especifica um passo de derivação mais à direita, e \Rightarrow_{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 intuição acerca dos manipuladores.

Definição: β é uma **frase** da forma sentencial à direita γ se, e somente se, $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1 \beta \alpha_2$.

Nessa definição, \Rightarrow^+ significa um ou mais passos de derivação.

Definição: β é uma **frase simples** da forma sentencial à direita γ se, e somente se, $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$.

Se essas duas definições forem comparadas cuidadosamente, está claro que se 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 podem parecer 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,

a cadeia de todas as folhas da árvore de análise sintática parcial que tem como raiz um nó interno em particular da árvore de análise sintática completa. Uma frase simples é uma 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 uma RHS da gramática.

A razão para discutir frases e frases simples é que o manipulador de qualquer forma sentencial mais à direita é sua frase simples mais à esquerda. Então, agora temos uma forma intuitiva de encontrar o manipulador de quaisquer formas sentenciais à direita, assumindo que temos a gramática e podemos desenhar uma árvore de análise sintática. Essa abordagem para encontrar manipuladores é impraticável para um analisador sintático (se você já tem uma árvore de análise sintática, por que você precisa 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, primeiro item a ser reescrito

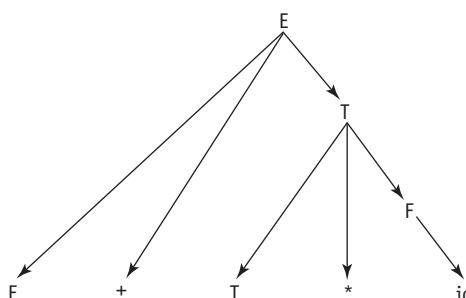


Figura 4.3 Uma árvore de análise sintática para $E + T^* id$.

na sentença para 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, a derivação mais à direita completa 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 eles especificam. Uma parte integral de todo o analisador sintático ascendente é uma pilha. A ação deslocar (*shift*) move o próximo símbolo de entrada para a pilha do analisador sintático. Uma ação de redução (*reduce*) substitui uma RHS (o manipulador) no topo da pilha do analisador sintático por sua LHS correspondente. Cada analisador sintático para uma linguagem de programação é um **autômato de pilha** (PDA). 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 nomeado dessa forma porque usa uma pilha como memória. 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 é 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 a que seria se estivesse em outra pilha, porque o PDA nunca vê mais do que o 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, o qual grava chamadas a subprogramas (dentre outras coisas), que correspondem aos não terminais da gramática.

4.5.3 Analisadores sintáticos LR

Muitos algoritmos diferentes de análise sintática ascendentes foram inventados. A maioria são variações de um processo chamado LR. Analisadores sintáticos LR usam um programa relativamente pequeno e uma tabela de análise sintática. 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 requeria 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) requerem muito menos recursos para produzir a tabela de análise sintática necessária do que o algoritmo LR canônico,

e (2) trabalham em classes menores de gramáticas do que o algoritmo LR canônico.

Existem diversas vantagens com o uso de analisadores sintáticos LR:

1. Eles podem ser construídos para todas as linguagens de programação.
2. Eles podem detectar erros de sintaxes 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 se poderia efetivamente procurar para a esquerda do manipulador sob suspeita até o final da pilha de análise sintática para determinar se ele era 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, nos 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:

$S_0 X_1 S_1 X_2 \dots X_m S_m$ (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

$(S_0 X_1 S_1 X_2 \dots X_m S_m, a_i a_{i+1} \dots a_n \$)$

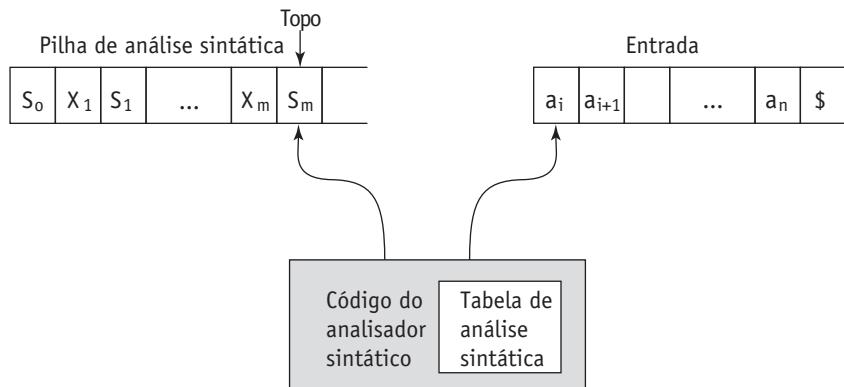


Figura 4.4 A estrutura de um analisador sintático LR.

Note que a cadeia de entrada tem um cifrão no seu final, do lado direito. O sinal é colocado lá durante a inicialização do analisador sintático. Ele é usado para o término normal do analisador sintático. Usando 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 maioria do que o analisador sintático faz. Ela tem símbolos de estado como os rótulos de linhas, e os símbolos terminais da gramática como os rótulos das colunas. Dado um estado atual do analisador sintático, que é 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 a LHS da regra cuja RHS é a mesma do manipulador. Duas outras ações são possíveis: aceitar (*accept*), ou seja, o analisador sintático completou com sucesso 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 os rótulos das 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 completada, 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 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 a LHS da regra usada na redução.

Considere a tradicional gramática 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 o deslocamento. R4 significa reduzir usando a regra 4; S6 significa deslocar o próximo símbolo da entrada na pilha e inserir o estado S_6 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.

Tabelas de análise sintática LR podem ser construídas usando uma ferramenta de software como o yacc³ (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

Estado	Action							Goto		
	id	+	*	()	\$	E	T	F	
0	S5			S4			1	2	3	
1		S6				aceitar				
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 árvore de análise sintática LR para uma gramática de expressões aritméticas.

³ O termo yacc é um acrônimo para “yet another compiler compiler” (mais um compilador de compiladores).

ção 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

$(S_0, a_1 \dots a_n \$)$

As ações do analisador sintático são formalmente definidas como segue:

1. Se $\text{ACTION}[S_m, a_i] = \text{Deslocar } S$, a próxima configuração é

$(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m a_i S, a_{i+1} \dots a_n \$)$

O processo de deslocamento 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.

Por exemplo, suponha que a configuração seja $(S_0 E S_1, + id \dots \$)$. A tabela ACTION especifica S_6 como a ação em sua posição $[1, +]$. Isso resulta na configuração $(S_0 E S_1 + S_6, id \dots \$)$.

2. Se $\text{ACTION}[S_m, a_i] = \text{Reducir } A \rightarrow \beta$ e $S = \text{GOTO}[S_{m-r}, A]$, onde $r =$ tamanho de β , a próxima configuração é

$(S_0 X_1 S_1 X_2 S_2 \dots X_{m-r} S A S, a_i a_{i+1} \dots a_n \$)$

Essa é uma ação muito mais complicada. Para uma ação de redução, 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 remover o manipulador e 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. Então, na nova configuração, o símbolo do topo vem da parte GOTO da tabela, e o símbolo da gramática mais ao topo é o lado esquerdo da regra usada na redução.

Por exemplo, suponha que a configuração seja $(S_0 id S_5, + id \dots \$)$. A tabela ACTION especifica R_6 em sua posição $[5, +]$, o que indica uma redução usando a regra 6, $F \rightarrow id$. O lado direito dessa regra tem tamanho 1, então dois símbolos podem ser removidos da pilha. Isso expõe S_0 no topo da pilha, então buscamos pela linha 0, coluna F da tabela GOTO (porque F é o lado esquerdo da regra usada na redução). Nessa posição da tabela GOTO, encontramos 3, então S_3 é inserida na pilha após a inserção de F.

3. Se $\text{ACTION}[S_m, a_i] = \text{Aceita}$, a análise sintática está completa e nenhum erro foi encontrado.
4. Se $\text{ACTION}[S_m, a_i] = \text{Erro}$, 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 se familiarizar com 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 no 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 o 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 \$	Reducir 6 (use GOTO[0, F])
0F3	+ id * id \$	Reducir 4 (use GOTO[0, T])
0T2	+ id * id \$	Reducir 2 (use GOTO[0, E])
0E1	+ id * id \$	Deslocar 6
0E1+6	id * id \$	Deslocar 5
0E1+6id5	* id \$	Reducir 6 (use GOTO[6, F])
0E1+6F3	* id \$	Reducir 4 (use GOTO[6, T])
0E1+6T9	* id \$	Deslocar 7
0E1+6T9*7	id \$	Deslocar 5
0E1+6T9*7id5	\$	Reducir 6 (use GOTO[7, F])
0E1+6T9*7F10	\$	Reducir 3 (use GOTO[6, T])
0E1+6T9	\$	Reducir 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 do escopo de um livro sobre linguagens de programação. Conforme mencionado previamente, existe um número de diferentes sistemas de software disponíveis para gerar tabelas de análise sintática LR.

RESUMO

A análise sintática é uma parte comum da implementação de linguagens, independentemente da abordagem de implementação usada. A análise sintática é normalmente

baseada em uma descrição de sintaxe formal da linguagem que está sendo implementada. Uma gramática livre de contexto, também chamada de BNF, é a abordagem 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 – dentre 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, chamados de lexemas, que 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 abordagens distintas para a construção de um analisador léxico: usando uma ferramenta de software para gerar uma tabela para um analisador dirigido por tabela, construindo tal tabela manualmente e escrevendo código para implementar uma descrição de um diagrama de estados dos *tokens* da linguagem sendo implementada. O diagrama de estados para *tokens* pode ser razoavelmente pequeno se as classes de caracteres forem usadas para transições, em vez de 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.

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. Os analisadores sintáticos são descendentes, 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 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, que 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 eles 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 previnem 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 cobrirmos isso, existe um algoritmo para remover tanto a recursão à esquerda direta quanto a indireta. O outro problema é detectado com o 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, que 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 de análise sintática que é relevante para o processo. Analisadores sintáticos usam duas tabelas de: ACTION e GOTO. A parte ACTION especifica o que o analisador sintático deve fazer, dado 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

- Quais são as três razões pelas quais os analisadores sintáticos são baseados em gramáticas?
- Explique as três razões pelas quais a análise léxica é separada da análise sintática.
- Defina lexema e *token*.
- Quais são as tarefas primárias de um analisador léxico?
- Descreva brevemente as três abordagens para a construção de um analisador léxico.
- O que é um diagrama de transição de estados?
- 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?
- Quais são os dois objetivos distintos da análise sintática?
- Descreva as diferenças entre analisadores sintáticos descendentes e ascendentes.
- Descreva o problema de análise sintática para um analisador sintático descendente.
- Descreva o problema de análise sintática para um analisador sintático ascendente.
- Explique por que os compiladores usam algoritmos de análise sintática que funcionam em apenas um subconjunto de todas as gramáticas.
- Por que as constantes nomeadas são usadas, em vez de números, para códigos de *tokens*?
- Descreva como um subprograma de análise sintática descendente recursiva é escrito para uma regra com um único lado direito.
- Explique as duas características das gramáticas que as proíbem de serem usadas como a base para um analisador sintático descendente.
- O que é o conjunto FIRST para uma gramática e forma sentencial?
- 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?

CONJUNTO DE PROBLEMAS

1. Faça o teste de disjunção par a par para as seguintes regras gramaticais.
 - a. $A \rightarrow aB \mid b \mid cBB$
 - b. $B \rightarrow aB \mid bA \mid aBb$
 - c. $C \rightarrow aaA \mid b \mid caB$
2. Faça o teste de disjunção par a par para as seguintes regras gramaticais.
 - a. $S \rightarrow aSb \mid bAA$
 - b. $A \rightarrow b\{aB\} \mid a$
 - c. $B \rightarrow aB \mid a$
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.
 $S \rightarrow aAb \mid bBA \quad A \rightarrow ab \mid aAB \quad B \rightarrow aB \mid b$
 - 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.
 $S \rightarrow Abb \mid bAc \quad A \rightarrow Ab \mid aBB \quad B \rightarrow Ac \mid cBb \mid c$
 - 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 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 ações para a cadeia $(id + id)^* id$, usando a gramática e a tabela de análise sintática da seção 4.5.3.
9. 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.

10. 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.
11. 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: $S \rightarrow Aa \mid Bb \quad A \rightarrow Aa \mid Abc \mid c \mid Sb \quad B \rightarrow bb$.

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 reconhecer a seguinte lista de palavras reservadas e retorne seus códigos de *token* respectivos: **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 para Java.
7. Converta as rotinas de análise sintática descendente recursiva para `<expr>`, `<term>` e `<factor>` dadas na Seção 4.4.1 para Java.
8. Para as regras que passam no teste no Problema 1, escreva um subprograma de análise sintática descendente recursiva que analise sintaticamente a linguagem gerada pelas regras. Assuma que você tem um analisador léxico chamado *lex* e um subprograma de tratamento de erros chamado *error*, chamado sempre que um erro de sintaxe for detectado.
9. Para as regras que passam no teste no Problema 2, escreva um subprograma de análise sintática descendente recursiva que analise sintaticamente a linguagem gerada pelas regras. Assuma que você tem um analisador léxico chamado *lex* e um subprograma de tratamento de erros chamado *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.

Capítulo 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 introduz as questões semânticas fundamentais das variáveis, começando pela natureza dos nomes e das palavras especiais em 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*). Depois, os conceitos importantes de vinculação e tempos de vinculação são introduzidos. Os diferentes tempos de vinculação possíveis para atributos de variáveis definem quatro diferentes categorias de variáveis. A seguir, 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 componentes primários 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 próximas. 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 requerem 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 requer que sejam consideradas diversas questões. (Os tipos de dados são discutidos no Capítulo 6). Dentre as mais importantes, estão o escopo e o tempo de vida das variáveis.

No restante deste livro, famílias de linguagens serão referenciadas como se fossem linguagens únicas. Por exemplo, Fortran significará todas as versões do Fortran – o mesmo caso de 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. A frase **linguagens baseadas em C** será usada para nos referirmos a C, C++, Java e C#¹.

¹ Estivemos tentados a incluir as linguagens de *scripting* JavaScript e PHP como linguagens baseadas em C, mas decidimos que elas são diferentes demais de suas antecessoras para fazermos isso.

5.2 NOMES



Antes de iniciarmos nossa discussão sobre variáveis, deve ser coberto o projeto de um dos atributos fundamentais das variáveis: os nomes, também associados com subprogramas, parâmetros formais e outras construções. O termo *identificador* é muito usado como sinônimo de *nome*.

5.2.1 Questões de projeto

As questões de projeto primárias para nomes são:

- Os nomes são sensíveis à capitalização?
- 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.

O Fortran 95 permite até 31 caracteres em seus nomes. O C99 não tem limitação em seus nomes internos, mas apenas os 63 primeiros são significativos. Nomes externos (definidos fora das funções, que devem ser manipulados pelo ligador) são restritos a 31 caracteres. Nomes em Java, C# e Ada não têm limites de tamanho, e todos os caracteres são significativos. Entretanto, as implementações de Ada podem impor uma limitação de tamanho, apesar de essa não poder restringir os nomes para um tamanho menor do que 200 caracteres – obviamente, essa não é uma restrição que causa incômodos. C++ não especifica um limite de tamanho, apesar de os implementadores algumas vezes fazê-lo porque a tabela de símbolos na qual os identificadores são armazenados durante a compilação não precisam ser tão grandes e para simplificar a manutenção dessa tabela.

Os nomes, na maioria das linguagens de programação, têm o mesmo formato: uma letra seguida por uma cadeia de letras, dígitos e sublinhados (_). Apesar do uso de sublinhados ter sido disseminado nos anos 1970 e 1980, a prática é menos popular atualmente. Nas linguagens baseadas em C, ele foi substituído pela chamada de camelCase, na qual todas as palavras de um nome contendo múltiplas palavras têm sua primeira letra em maiúsculo, exceto a primeira palavra,

NOTA HISTÓRICA

As primeiras linguagens de programação usavam nomes com um caractere. Essa notação era natural porque a programação no início era principalmente matemática, e os matemáticos têm usado nomes com um único caractere há muito tempo para variáveis em suas notações formais.

O Fortran I quebrou com a tradição de nomes de um único caractere, permitindo até seis em seus nomes. O Fortran 77 ainda restringia os nomes a seis caracteres.

NOTA HISTÓRICA

Em versões do Fortran anteriores ao Fortran 90, apenas letras maiúsculas poderiam ser usadas em nomes – uma restrição desnecessária. A origem dessa restrição era o fato de que as perfuradoras de cartões tinham apenas letras maiúsculas. Como o Fortran 90, muitas implementações do Fortran 77 permitem letras minúsculas; elas simplesmente traduzem-nas para letras maiúsculas para uso interno durante a compilação.

como em `my$tack`². Note que o uso de sublinhados e de capitalização 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. Em Ruby, caracteres especiais no início do nome, @ ou @@, indicam que a variável é uma instância ou uma variável de classe, respectivamente.

Em versões do Fortran anteriores ao Fortran 90, os nomes podiam ter espaços em branco, que eram ignorados. Por exemplo, os dois nomes a seguir eram equivalentes:

```
Sum Of Salaries  
SumOfSalaries
```

Em muitas linguagens, mais notavelmente nas baseadas em C, as letras maiúsculas e minúsculas nos nomes são distintas; ou seja, são **sensíveis à capitalização**. Por exemplo, os três nomes seguintes são distintos em C++: rosa, ROSA e Rosa. Para algumas pessoas, esse é um sério detimento à legibilidade, porque os nomes que se parecem denotam entidades diferentes. Nesse sentido, a sensibilidade à capitalização 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 capitalização, apesar de Rosa e rosa parecerem similares, não existe uma conexão entre elas.

Obviamente, nem todo mundo concorda que a sensibilidade à capitalização é ruim para nomes. Em C, os problemas da sensibilidade à capitalização 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 pré-definidos incluem tanto maiúsculas quanto minúsculas. Por exemplo, o método em Java para converter uma cadeia para um valor inteiro é `parseInt`. Nesse caso, as variações `ParseInt` e `parseint` não são reconhecidas. Esse é um problema de facilidade de escrita em vez de um de legibilidade porque a necessidade de lembrar o uso de capitalização 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

Palavras especiais em linguagens de programação são usadas para tornar os programas mais legíveis ao nomearem as ações a serem realizadas. Elas tam-

² Ela é chamada de “camelo” porque as palavras escritas usando a notação normalmente têm letras maiúsculas embutidas, que se parecem com as corcovas de um camelo.

bém são usadas para separar as partes sintáticas das sentenças e programas. Nos códigos de programa de exemplo deste livro, as palavras especiais são apresentadas em negrito. Na maioria das linguagens, elas são classificadas como palavras reservadas, mas em algumas são apenas palavras-chave.

Uma **palavra-chave** é uma palavra de uma linguagem de programação especial apenas em alguns contextos. Fortran é a única linguagem bastante usada que restou cujas palavras especiais são palavras-chave. Em Fortran, a palavra `Integer`, quando encontrada no início de uma sentença e seguida por um nome, é considerada uma palavra-chave que indica que a sentença é declarativa. Entretanto, se a palavra `Integer` é seguida por um operador de atribuição, é considerada um nome de variável. Os dois usos são ilustrados a seguir:

```
Integer Apple  
Integer = 4
```

Compiladores Fortran e pessoas que estejam lendo os programas Fortran devem distinguir entre nomes e palavras especiais pelo contexto.

Uma **palavra reservada** é uma palavra especial de uma linguagem de programação que não pode ser usada como um nome. Como uma escolha de projeto de linguagem, as palavras reservadas são melhores do que as palavras-chave porque a habilidade de redefinir palavras-chave pode ser confusa. Por exemplo, em Fortran, alguém poderia ter escrito as seguintes sentenças:

```
Integer Real  
Real Integer
```

que declaram a variável de programa chamada `Real` como sendo do tipo `Integer` e a variável chamada `Integer` como sendo do tipo `Real`³. Além da estranha aparência dessas sentenças de declaração, a aparição de `Real` e `Integer` como nomes de variáveis em outros lugares do programa poderia confundir os leitores.

Existe um problema em potencial com as palavras reservadas: se a linguagem incluir um grande número de palavras reservadas, o usuário tem dificuldades para inventar nomes que não são reservados. O melhor exemplo disso é o 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`.

Algumas linguagens incluem nomes pré-definidos, que de alguma forma estão entre as palavras especiais e os nomes definidos pelo usuário. Eles têm significados pré-definidos, mas que podem ser redefinidos pelo usuário. Por exemplo, os nomes dos tipos de dados disponíveis em Ada, como `Integer` e

³ É claro, qualquer programador profissional que escrever tal código não deve esperar ter estabilidade no emprego.

`Float`, são pré-definidos. Esses nomes não são reservados; eles podem ser redefinidos por quaisquer programas Ada.

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 ser tornados visíveis para um programa. Esses nomes são pré-definidos, 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 existe muito mais acerca de uma variável do que apenas um nome.

A mudança das linguagens de máquina para as de montagem era dirigida pela necessidade de substituir endereços numéricos e absolutos de memória para dados por nomes, tornando os programas muito mais legíveis e mais fáceis de serem escritos e mantidos. Esse passo também fornecia um escape para o problema do endereçamento absoluto feito de forma manual, porque o tradutor que convertia os nomes para endereços reais também escolhia esses endereços.

Uma variável pode ser caracterizada como um conjunto de seis atributos (nome, endereço, valor, tipo, tempo de vida, escopo). Apesar de isso parecer muito complicado para um conceito aparentemente tão simples, essa caracterização fornece a maneira mais clara de explicar os vários aspectos das variáveis.

Nossa discussão dos atributos das variáveis levará a examinarmos conceitos relacionados importantes, 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 com dife-

rentes endereços em vários momentos em um 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 com 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 uma variável aparece no lado esquerdo de uma atribuição.

É possível haver múltiplas variáveis com o mesmo endereço. Quando mais de uma variável pode ser usada para acessar a mesma posição de memória, elas 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 à uma variável diferente. Por exemplo, se as variáveis `total` e `soma` são apelidos, quaisquer mudanças à `total` também modificam `soma` e vice-versa. Um leitor do programa deve sempre se lembrar que `total` e `soma` são nomes diferentes para a mesma célula de memória. Como pode existir qualquer número de apelidos em um programa, isso é muito difícil na prática. O uso de apelidos também torna a verificação de programas mais difícil.

Apelidos podem ser criados em programas de diversas formas. 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 elas 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 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 com um endereço é muito importante para o entendimento das linguagens de programação. Esse assunto é discutido na Seção 5.4.3.

NOTA HISTÓRICA

A sentença Equivalence do Fortran foi projetada unicamente para criar apelidos. Uma das motivações para tal sentença era economizar espaço de armazenamento. Como agora o Fortran tem armazenamento dinâmico e memória, é relativamente abundante, não existe mais razão para usar Equivalence. Como resultado disso, a sentença Equivalence foi depreciada no Fortran 90.

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 aritmé-

ticas para adição, subtração, multiplicação, divisão e módulo. Os tipos de dados são discutidos no Capítulo 6, chamado “Tipos de Dados”.

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, das memórias da maioria dos computadores modernos têm tamanho de um byte, com um byte normalmente tendo tamanho de 8 bits – muito pequeno para a maioria das variáveis de programas. Uma célula abstrata de memória tem o tamanho necessário pela variável com a qual está associada. Por exemplo, apesar de os valores de ponto flutuante poderem ocupar quatro bytes físicos em uma implementação de uma linguagem em particular, 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* significa uma célula abstrata de memória.

O valor de uma variável é algumas vezes chamado de **lado direito (*r-value*)** porque é requerido quando a variável é usada no lado direito de uma sentença de atribuição. Para acessar o lado direito, o lado esquerdo precisa primeiro ser determinado. Tais determinações não são sempre simples. Por exemplo, regras de escopo podem complicar enormemente as coisas, conforme discutido na Seção 5.8.

5.4 O CONCEITO DE VINCULAÇÃO



De um modo geral, uma **vinculação** é uma associação, como entre um atributo e uma entidade 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 em 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 em Java:

```
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 com essa sentença.

Um 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.

5.4.1 Vinculação de atributos a variáveis

Uma vinculação é **estática** se ela ocorre pela primeira vez antes do tempo de execução e permanece intocada 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.

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 um 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 projetadas desde meados dos anos 1960 requer a declaração explícita de todas as variáveis (Perl, JavaScript, Ruby e ML são algumas exceções). Diversas linguagens cujos projetos iniciais foram feitos antes do final dos anos 1960 – notavelmente, o Fortran e o BASIC – têm declarações implícitas⁴. Em Fortran, um identificador que aparece em um programa e não é explicitamente declarado é implicitamente declarado de acordo com a seguinte convenção: se o identificador começar com uma das letras I, J, K, L, M ou N, ou em suas versões minúsculas, ele é implicitamente declarado do tipo Integer; caso contrário, é implicitamente declarado do tipo Real.

Apesar de serem uma pequena conveniência para os programadores, as declarações implícitas podem ser prejudiciais à confiabilidade porque previnem o processo de compilação de detectar alguns erros de programação e de digitação. No Fortran, as variáveis accidentalmente deixadas sem declaração pelo programador recebem tipos padronizados e atributos inesperados, que podem causar erros sutis difíceis de ser diagnosticados. Muitos programadores agora incluem a declaração Implicit none em seus programas. Essa declaração instrui o compilador para não declarar implicitamente quaisquer variáveis, evitando os problemas em potencial de accidentalmente termos variáveis não declaradas.

Alguns dos problemas com declarações implícitas podem ser evitados obrigando os nomes para tipos específicos começarem com caracteres especiais em particular. 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*). 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 quando lê seu nome. Note que esse projeto é diferente daquele do Fortran, porque o Fortran tem tanto declarações implícitas quanto explícitas, então o tipo de uma variável não pode ser necessariamente determinado pelo formato de seu nome.

A Seção 5.4.2.3 discute outro tipo de vinculação implícita de tipos: a inferência de tipos.

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

⁴ Dialetos atuais do BASIC – como o Visual BASIC .NET – não permitem declarações implícitas.

valor a ela em uma sentença de atribuição. Quando a sentença de atribuição é executada, a variável que está recebendo um valor atribuído é vinculada ao tipo do valor da expressão no lado direito da atribuição.

As linguagens nas quais os tipos são vinculados dinamicamente são drasticamente diferentes daquelas nas quais os tipos são vinculados estaticamente. A vantagem principal da vinculação dinâmica de variáveis a tipos é que ela fornece uma flexibilidade maior 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 o dado for atribuído às variáveis após a entrada. Ao contrário, devido à vinculação de tipos estática, não é possível escrever um programa Java para processar dados sem conhecer os tipos desses dados⁵.

Em JavaScript e PHP, a vinculação de uma variável a um tipo 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 um vetor unidimensional de tamanho 2. Se a sentença

```
list = 47;
```

seguisse a atribuição de exemplo, `list` se tornaria uma variável escalar.

Existem duas desvantagens da vinculação de tipos dinâmica. Primeiro, ela faz os programas serem 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 valores de quaisquer tipos serem atribuídos 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 para o tipo incorreto. Por exemplo, suponha que em um programa JavaScript em particular, `i` e `x` estivessem armazenando valores numéricos escalares, e `y` estivesse armazenando 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;
```

⁵ Java tem algumas estruturas de dados pré-definidas que podem armazenar dados genéricos. Elas são discutidas no Capítulo 9.

Em JavaScript (ou qualquer outra linguagem que usa vinculação de tipos dinâmica), nenhum erro é detectado nessa sentença pelo interpretador – 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 essa desvantagem também está presente em certo grau em algumas linguagens que usam vinculação de tipos estática, como Fortran, C e C++, nos quais muitos casos convertem automaticamente o tipo do lado direito de uma atribuição para o 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 descriptor 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.

Na linguagem orientada a objetos pura, Ruby, as variáveis não possuem tipos – todos os dados são objetos e qualquer variável pode referenciar qualquer objeto. Logo, não pode existir verificação de tipos nos operadores de atribuição.

Por fim, as linguagens que têm vinculação de tipos dinâmica são normalmente implementadas usando 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 ao menos 10 vezes mais tempo para executar um código de máquina equivalente. É claro, 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 cara nesse ambiente. Por outro lado, as linguagens com vinculações de tipo estáticas são raramente implementadas pela interpretação pura, pois os programas nessas linguagens podem ser facilmente traduzidos para versões em código de máquina muito eficientes.



Linguagens de *scripting* e outros exemplos de soluções simples

RASMUS LERDORF

Rasmus Lerdorf nasceu na ilha de Disko, na costa da Groenlândia, em 1968. Após graduar-se em engenharia, Lerdorf desempenhou diversos trabalhos de consultoria. Em uma iniciativa para descobrir quem estava vendendo seu currículo online, ele criou a primeira iteração de PHP. Atualmente, é evangelista do movimento de código aberto e colaborador do Yahoo! em Sunnyvale, Califórnia.

ALGUMAS EXPERIÊNCIAS PASSADAS

Qual foi sua primeira experiência com computação? Meu pai e eu construímos juntos um jogo de "pong" a partir de um kit que ele pediu dos Estados Unidos, em meados de 1976. Também me lembro de ter um "Speak & Spell" da Texas Instruments, por volta de 1978, que tinha o primeiro chip único de sintetização de voz acoplado. Ganhei meu primeiro computador em 1983 ou por volta disso. Era um Commodore Vic20, com 5k de RAM e um processador 6502 de 1MHz. Eu perdia horas digitando trechos de código das revistas. No ensino médio, brincava com um Commodore PET e um UNISYS 80186 que rodavam o QNX. O QNX era de longe o sistema operacional mais legal que eu já tinha visto e foi de certa forma um anticlímax quando mais tarde obtive um computador rodando MS-DOS. Penso que minhas experiências iniciais me direcionaram ao UNIX e aos sistemas operacionais semelhantes a ele.

Você tem um emprego anterior favorito? Gostei muito do tempo que estive no Brasil, e eles também me apresentaram ao Vale do Silício quando abriram um escritório em Mountain View e me mudei para lá em 1993. Também gostei muito de trabalhar na Universidade de Toronto depois disso, ajudando-os a construir um sistema de discagem. Uma boa parte do PHP foi criada durante esse emprego.

SOBRE AS LINGUAGENS DE *SCRIPTING*

Qual é a sua definição de uma linguagem de *scripting*? É uma linguagem de alto nível que esconde o tédio e a complexidade de qualquer técnica de progra-

mação tradicional usada para solucionar uma classe de problemas específica.

Quando você estava trabalhando em sua página pessoal e, mais tarde, no sistema para a Universidade de Toronto, querendo acompanhar os visitantes de seu site, e em seguida acompanhar estudantes, que ferramentas disponíveis você considerou usar? Não me lembro de ter considerado nenhuma das soluções existentes para isso. Na época, você normalmente leria os logs de acesso puros de seu servidor Web. É claro que existiam algumas ferramentas para análise de log, em sua maioria escritas em Perl, que resumiam as coisas, mas eu queria receber um email cada vez que alguém lesse o meu currículo e saber de onde essa pessoa tinha vindo. Não existiam ferramentas especificamente para isso.

Como você decidiu que gastar tempo para criar sua própria solução era melhor que usar as ferramentas já disponíveis? A principal alternativa era sempre Perl. E, apesar do que algumas pessoas atribuíram a mim ao longo dos anos, não odeio Perl. Gosto dele e o uso razoavelmente, mas na época minhas necessidades eram simples. Eu estava rodando minha página pessoal em um servidor compartilhado com pouca memória RAM ou CPU, e criar e executar um CGI Perl para cada requisição era muito caro em termos de recursos. Precisava de um analisador sintático simples que eu poderia embutir diretamente em meu servidor Web, e Perl era muito difícil e grande para ser embutido; eu não precisava dos poderes de Perl para o que estava fazendo na época. Então, escrevi um analisador sintático pequeno e simples que eu entenderia e poderia ser facilmente embutido.

Ele começou a crescer, é claro, e uma vez que você tenha adicionado qualquer tipo de fluxo lógico, é um passo em direção a uma linguagem de programação completa. O objetivo original nunca foi escrever uma linguagem completa.

“Meu título oficial é “Técnico Yahoo!” e estou na equipe de infraestrutura que mantém e dá suporte às várias ferramentas usadas em toda a empresa. Meu foco é PHP e Apache.”

O que o PHP oferece hoje que outras linguagens de scripting (Perl, Tcl, Python, Ruby) não oferecem? PHP é direcionado a Web. Tudo o que você lê sobre PHP segue nessa direção, assim se você está tentando resolver um problema na Web é fácil aplicar PHP. Não é tão claro com outras linguagens de propósito mais geral em que você primeiro pega a linguagem, e em seguida tem de tentar descobrir a melhor maneira de aplicá-la em específico.

O futuro do PHP: qual é a próxima ideia, ou funcionalidade que você gostaria de ver executada ou adicionada no código? Precisamos de um repositório de qualidade de código PHP e extensões, porque nossa tendência anterior de empacotar tudo com PHP não foi bem em termos de escalabilidade.

Sobre soluções simples

Você disse em uma entrevista: “Eu definitivamente aprecio e respeito uma solução simples para um problema difícil.” Que processos mentais e quais atividades levam a encontrar essa solução simples? Inventar soluções para problemas é ser capaz de abordá-los de muitos ângulos. Algumas vezes, o treinamento formal e a sabedoria convencional podem atrapalhar e limitar a criatividade. Você também precisa estar preparado para falhar várias vezes.

Um problema é difícil de resolver apenas porque você não encontrou a solução simples ainda.

“Inventar soluções para problemas é ser capaz de abordá-los de muitos ângulos. Algumas vezes, o treinamento formal e a sabedoria convencional podem atrapalhar e limitar a criatividade. Você também precisa estar preparado para falhar várias vezes.”

É um pouco parecido com aqueles quebra-cabeças com palavras embaralhadas que você vê no jornal. Você olha para a palavra embaralhada e ela não faz sentido. Você tenta chegar à combinação correta de letras, mas não conseguevê-la. Então, alguém sussurra a palavra para você e subitamente ela óbvia. Agora lhe parece você olha para as letras, a palavra está lá e você não consegue entender como não a viu logo de cara. Esse é exatamente o mesmo sentimento que eu tenho quando descubro uma boa solução para um problema ou vejo que alguém já decobriu.

Quais são algumas de suas soluções simples favoritas? O mundo está repleto de tais soluções. Clipes de papel, velcro, a ponta da caneta esferográfica. Mas eu imagino que você esteja perguntando de elementos dentro do PHP. Eu acho que uma das melhores soluções foi amarrar o *get*, o *post* e os dados de *cookies* HTML diretamente às variáveis PHP. Parecia óbvio, mas na época ninguém estava fazendo e isso tornou PHP uma solução bastante agradável para os problemas da Web. O recurso vem recebendo algumas críticas por causa da possibilidade de ele ser usado de maneira imprópria, mas eu ainda o defendo e acredito ser uma solução simples.

5.4.2.3 Inferência de tipos

ML é uma linguagem de programação que oferece suporte tanto para programação funcional quanto para programação imperativa (Ullman, 1998). ML emprega um mecanismo interessante de inferência de tipos, no qual os tipos da maioria das expressões podem ser determinados sem precisar que o programador especifique os tipos das variáveis.

Antes de investigar a inferência de tipos por meio de funções ML, primeiro considere a sintaxe geral de uma função ML:

```
fun nome_função(parâmetros formais) = expressão;
```

O valor da expressão é retornado pela função⁶.

Agora podemos discutir a inferência de tipos. Considere a declaração de uma função em ML

```
fun circumf (r) = 3.14159 * r * r;
```

Essa declaração especifica uma função chamada `circumf` que recebe um ponto flutuante (`real` em ML) como argumento e produz um resultado como ponto flutuante. Os tipos são inferidos pelo tipo da constante na expressão. Da mesma forma, na função

```
fun times10 (x) = 10 * x;
```

o argumento 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, assume-se que o tipo do parâmetro e o tipo da função sejam numéricos. Em ML, o tipo numérico padrão é o tipo `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`.

⁶ A expressão pode ser uma lista de expressões, separadas por ponto e vírgula e envolta em parênteses. O valor de retorno neste caso é o da última expressão.

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 puramente funcionais Miranda e Haskell.

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 ao conjunto de células de memória disponíveis.

O **tempo de vida** de uma variável é o 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 variáveis escalares (não estruturadas) em quatro categorias, de acordo com seus tempos de vida. Essas categorias são estáticas, dinâmicas da pilha, dinâmicas do monte explícitas e dinâmicas do monte implícitas. Nas seções a seguir, discutimos os significados 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 valiosas em programação. Variáveis acessíveis globalmente são usadas ao longo da execução de um programa, tornando necessário tê-las vinculadas ao mesmo armazenamento durante essa execução. Algumas vezes, é conveniente ter subprogramas sensíveis ao histórico. Tal subprograma deve ter variáveis locais estáticas.

Outra vantagem das variáveis estáticas é a eficiência. Todo o endereçamento de variáveis estáticas pode ser direto⁷; outros tipos de variáveis ge-

⁷ Em algumas implementações, as variáveis estáticas são endereçadas por meio de um registrador-base, fazendo o acesso a elas ser o mesmo de variáveis alocadas na pilha.

ralmente requerem 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 negligenciável.

Uma desvantagem da vinculação estática ao 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 requerem grandes vetores. 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** a uma definição de variável em uma função, fazendo as variáveis definidas serem estáticas. Note que, quando um modificador **static** aparece na declaração de uma variável em uma definição de classe em C++, Java e C#, seu significado tem apenas uma conexão indireta com o conceito de tempo de vida da variável. Nesse contexto, significa que a variável é de classe, em vez de ser de instância. As variáveis de classe são criadas algum tempo antes de a classe ser instanciada pela primeira vez. Esse uso múltiplo de uma palavra reservada pode ser confuso, particularmente para quem está aprendendo a linguagem.

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 estaticamente vinculados. 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 com o 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 declarações de variáveis ocorrerem 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 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.

Para serem úteis, ao menos na maioria dos casos, os programas recursivos requerem 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, relativas às variáveis estáticas, são a sobrecarga em tempo de execução da alocação e liberação, acessos 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, em vez de em operações separadas.

O Fortran 95 permite aos implementadores usarem variáveis dinâmicas da pilha para variáveis locais, mas inclui uma sentença

`Save list`

que permite ao programador especificar algumas ou todas as variáveis (aqueles na lista) no subprograma no qual `Save` é colocado como estáticas. Em Java, C++ e C#, as variáveis definidas em métodos são, por padrão, dinâmicas da pilha. Em Ada, todas as variáveis que não são do monte e são definidas em subprogramas são 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 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 cuja organização é altamente desorganizada, por causa da 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 ou por meio de um operador (por exemplo, em Ada e 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 seu operando. Quando executado, uma variável dinâmica do monte explícita do tipo do operando é criada e um ponteiro a ela 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 elas 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 explicitamente destruí-las.

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++ requer 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, a coleta de lixo implícita é usada.

C# tem tanto objetos dinâmicos do monte explícitos quanto dinâmicos da pilha, dos quais todos são implicitamente liberados. Além disso, C# também 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 em C++, e os objetos que eles referenciam no monte não são implicitamente liberados. Ponteiros são incluídos em C# para permitir que os componentes C# interoperem com componentes C e C++. Para desencorajar seu uso, o cabeçalho de qualquer método que define um ponteiro deve incluir 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 usando 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 de referências às variáveis e a complexidade da implementação do gerenciamento

de armazenamento. Esse é essencialmente o problema do gerenciamento do monte, que é custoso 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. De certa forma, são apenas nomes que se adaptam a qualquer uso para o qual são solicitadas a atender. Por exemplo, considere a seguinte sentença de atribuição em JavaScript:

```
highs = [74, 84, 86, 90, 71];
```

Independentemente se a variável `highs` foi previamente usada no programa ou para 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 códigos altamente genéricos serem escritos. 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, dentre outros. Outra desvantagem é a perda de alguma detecção de erros pelo compilador, conforme discutido na Seção 5.4.2.2. Exemplos de variáveis dinâmicas do monte implícitas em JavaScript aparecem na Seção 5.4.2.2.

5.5 ESCOPO



Um dos fatores mais 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 nessa sentença. As regras de escopo de uma linguagem determinam como uma ocorrência em particular de um nome é associada com uma variável. Em particular, regras de escopo determinam como referências a variáveis declaradas fora do subprograma ou bloco em execução são associadas com suas declarações e, logo, com seus atributos (os blocos são discutidos na Seção 5.5.2). Logo, um claro entendimento dessas regras para uma linguagem é essencial para a habilidade de escrever ou ler programas nela.

Conforme definido na Seção 5.4.3.2, uma variável é local a uma unidade ou a um bloco de programa se ela for lá declarada. 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.

Questões de escopo para classes, pacotes e espaços de nomes são discutidas no Capítulo 11.

5.5.1 Escopo estático

O ALGOL 60 introduziu o método de vincular nomes a variáveis não locais, chamado de **escopo estático**⁸, copiado por muitas linguagens imperativas subsequentes (e por muitas linguagens não imperativas). O escopo estático é chamado assim porque o escopo de uma variável pode ser determinado estaticamente – ou seja, antes da execução. Isso permite a um leitor de programas humano (e um compilador) determinar o tipo de cada variável.

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. Nessa 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, Fortran 2003 e PHP permitem subprogramas aninhados, mas as linguagens baseadas em C não.

Nossa discussão sobre o uso de escopo estático nesta seção foca nas linguagens que permitem subprogramas aninhados. Inicialmente, assumimos que *todos* os escopos são associados com unidades de programas e que todas as variáveis não locais referenciadas são declaradas em outras unidades de programa⁹. Neste capítulo, assumimos que o uso de escopos é o único método de acessar variáveis não locais nas linguagens em discussão, o que 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. Em linguagens de escopo estático com subprogramas, esse processo pode ser visto da seguinte forma. 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 é encontrada lá, a busca continua para a próxima unidade maior que envolve o subprograma onde está sendo feita a busca (a unidade que declarou o pai de *Sub1*), e assim por diante, até que uma declaração para *x* seja encontrada ou a maior unidade de declaração tenha sido buscada sem sucesso. Nesse caso, um erro de variável não declarada é detectado. O pai estático do subprograma *Sub1*, e seu pai estático, e assim por diante até (e incluindo) o maior subprograma que envolve os demais, são chamados de

⁸ O escopo estático é chamado algumas vezes de *escopo léxico*.

⁹ Variáveis não locais não definidas em outras unidades de programa são discutidas na Seção 5.5.4.

ancestrais estáticos de Sub1. Técnicas de implementação reais para escopo estático, discutidas no Capítulo 10, são muito mais eficientes do que o processo que acabamos de descrever.

Considere o seguinte procedimento em Ada, chamado Big, no qual estão aninhados os procedimentos Sub1 e Sub2:

```
procedure Big is
    X : Integer;
    procedure Sub1 is
        X : Integer;
        begin -- de Sub1
        ...
        end; -- de Sub1
    procedure Sub2 is
        begin -- de Sub2
        ...X...
        end; -- de Sub2
    begin -- de Big
    ...
end; -- de Big
```

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 ele não está nos ancestrais estáticos de Sub2.

A presença de nomes pré-definidos, descritos na Seção 5.2.3, complica esse processo. Em alguns casos, um nome pré-definido é como uma palavra-chave, que pode ser redefinida pelo usuário. Em tais casos, um nome pré-definido é usado apenas se o programa de usuário não contém uma redefinição.

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 o procedimento Big em Ada. A variável X é declarada tanto em Big quanto em Sub1, aninhado dentro de Big. Dentro de Sub1, cada referência simples para X é para o X local. Logo, o X externo está oculto de Sub1.

Em Ada, variáveis ocultas de escopos ancestrais podem ser acessadas com referências seletivas, que incluem o nome do escopo ancestral. Em nosso

NOTA HISTÓRICA

A definição original do Pascal (Wirth, 1971) não especifica quando a compatibilidade de tipo por estrutura ou por nome deve ser usada. Isso é altamente prejudicial à portabilidade, porque um programa correto em uma implementação pode ser ilegal em outra. O Pascal Padrão ISO – ISO Standard Pascal (ISO, 1982) informa as regras de compatibilidade de tipos da linguagem, as quais não são completamente por nome nem completamente pela estrutura. A estrutura é usada na maioria dos casos, enquanto o nome é usado para parâmetros formais e em algumas outras poucas situações.

procedimento de exemplo anterior, o `x` declarado em `Big` pode ser acessado em `Sub1` pela referência `Big.x`.

5.5.2 Blocos

Muitas linguagens permitem que novos escopos estáticos sejam definidos no meio do código executável. Esse poderoso conceito, introduzido no ALGOL 60, permite 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. Tais seções de código são chamadas de **blocos**; daí a origem da frase **linguagem estruturada em 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 blocos. Por exemplo, se `list` fosse um vetor de inteiros, alguém poderia escrever

```
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 seguinte 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, uma declaração de uma variável efetivamente esconde quaisquer

declarações de variáveis com o mesmo nome no escopo externo maior¹⁰. Note que esse código é legal em C e C++, mas ilegal 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.

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 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 é de suas declarações até o final dos blocos nos quais essas declarações aparecem. Entretanto, 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 verdade para os métodos. Note que C# ainda requer todas as variáveis declaradas antes de serem usadas. Logo, independentemente do 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.

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 envolvida 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` é a partir da sentença `for` até o final de seu corpo.

¹⁰ Conforme discutido na Seção 5.5.4, em C++ tais variáveis globais ocultas podem ser acessadas no escopo interno usando o operador de escopo `(::)`.

5.5.4 Escopo global

Algumas linguagens, incluindo C, C++, PHP 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 a 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 aquelas que incluem uma declaração de uma variável local com o mesmo nome. Uma variável global definida após uma função pode ser tornada visível na função declarando-a como externa, como:

```
extern int sum;
```

Em C99, as definições de variáveis globais sempre têm valores iniciais, mas as declarações de variáveis globais nunca têm. Se a declaração estiver fora das definições de função, ela pode incluir o qualificador `extern`.

Essa ideia de declarações e definições é usada também para funções em C e C++, onde 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 usando o 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`.

Programas em PHP são geralmente embutidos em documentos XHTML. Independentemente se estiverem embutidos em XHTML ou em arquivos próprios, os programas em PHP são puramente interpretados. Sentenças podem ser interpoladas com definições de funções. Quando encontradas, as sentenças são interpretadas; as definições de funções são armazenadas para referências futuras. As variáveis em PHP são implicitamente declaradas quando aparecem como alvos de sentenças de atribuição. 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 de suas declarações até o fim do programa, mas pulam sobre quaisquer definições de funções subsequentes. Logo, as variáveis

globais não são implicitamente visíveis em nenhuma função. As variáveis globais podem ser tornadas visíveis em funções em seu escopo de duas formas. Se a função inclui uma variável local com o mesmo nome da global, esta pode ser acessada por meio do vetor \$GLOBALS, usando o nome da variável como o índice do vetor. Se não existe uma variável local na função com o mesmo nome da global, esta pode se tornar visível com sua inclusão em uma sentença de declaração global. Considere o exemplo:

```
$day = "Monday";
$month = "January";

function calendar() {
    $day = "Tuesday";
    global $month;
    print "local day is $day <br />";
    $gday = $GLOBALS['day'];
    print "global day is $gday <br \>";
    print "global month is $month <br />";
}

calendar();
```

A interpretação desse código produz:

```
local day is Tuesday
global day is Monday
global month is January
```

As regras de visibilidade para variáveis globais em Python não são usuais. As variáveis não são normalmente declaradas, 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 uma variável global pode ter valores atribuídos a ela apenas se tiver sido declarada como global na função. Considere os exemplos:

```
day = "Monday"

def tester():
    print "The global day is:", day

tester()
```

A saída desse *script*, como as variáveis globais podem ser referenciadas diretamente nas funções, é:

```
The global day is: Monday
```

O *script* a seguir tenta atribuir um novo valor a 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 a referência a `day` na primeira linha do corpo da função se tornar uma referência ilegal para a variável local.

A atribuição a `day` pode ser para a variável global se `day` for declarada como global no início da função. Isso previne que a atribuição de valores a `day` crie uma variável local, como 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
```

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. Esses casos são discutidos no Capítulo 12, “Suporte à Programação Orientada a Objetos”.

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 fornece mais acesso tanto a variáveis quanto a subprogramas do que o necessário – uma ferramenta pouco refinada 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. 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 está-

tico pode levar a projetos de programas que mantêm pouca semelhança ao seu original, mesmo em áreas do programa nas quais mudanças não foram feitas. Os projetistas são encorajados a usar muito mais variáveis globais do que o necessário. Todos os subprogramas podem terminar aninhados no mesmo nível, no programa principal, usando variáveis globais em vez de níveis mais profundos de aninhamento¹¹. 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 o uso de construções de encapsulamento, incluídas em muitas das novas linguagens. Construções de encapsulamento são discutidas no Capítulo 11, “Tipos Abstratos de Dados e Construções de Encapsulamento”.

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 o procedimento Big da Seção 5.5.1:

```
procedure Big is
    X : Integer;
    procedure Sub1 is
        X : Integer;
        begin -- de Sub1
        ...
        end; -- de Sub1
    procedure Sub2 is
        begin -- de Sub2
        ...X...
        end; -- de Sub2
    begin -- de Big
    ...
    end; -- de Big
```

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 a qualquer uma das declarações de x, dependendo da sequência de chamadas.

Uma maneira pela qual o significado correto de x pode ser determinado em tempo de execução é iniciar a busca com as variáveis locais. Essa também

¹¹ Parece muito com a estrutura de um programa em C, não parece?

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 desse procedimento chamador, e assim por diante, até que uma declaração de `x` seja encontrada. 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 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 conforme discutimos nesta seção, apesar de a semântica ser geralmente ser 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. 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. 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 por causa do 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 estão todas visíveis para qualquer outro subprograma 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 previamente chamados 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 com o 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 faz os programas muito mais difíceis de serem lidos, porque a sequência de chamadas de subprogramas deve ser conhecida para 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 acesso a não locais quando o escopo estático é usado. A razão para isso é explicada no Capítulo 10.

Por outro lado, o escopo dinâmico tem seus méritos. Em alguns casos, os parâmetros passados de um subprograma para outro são variáveis definidas no chamador. Nada disso precisa ser passado em uma linguagem com escopo dinâmico, porque elas são implicitamente visíveis no subprograma chamado.

Não é difícil entender por que o escopo dinâmico não é tão usado como 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 é de sua declaração até o final do método. O tempo de vida dessa variável é do 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 usando o 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 ela é parte.

O escopo e o tempo de vida também não são relacionados quando chamadas a subprogramas estão envolvidas. 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 ao 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 com a 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 na sentença. 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 técnicas para implementar referências a variáveis não locais tanto em linguagens de escopo estático quanto de escopo dinâmico.

Em Ada, escopos podem ser criados por definição de procedimentos. O ambiente de referenciamento de uma sentença inclui as variáveis locais e todas as variáveis declaradas em procedimentos nos quais a sentença está aninhada (excluindo variáveis em escopos não locais que estejam ocultos por declarações em procedimentos mais próximos). Cada definição de procedimento cria um novo escopo e, dessa forma, um novo ambiente.

Considere o seguinte esqueleto de programa em Ada:

```
procedure Example is
    A, B : Integer;
    ...
    procedure Sub1 is
        X, Y : Integer;
        begin -- de Sub1
            ... <----- 1
        end; -- de Sub1
    procedure Sub2 is
        X : Integer;
        ...
    end;
```

```

procedure Sub3 is
    X : Integer;
begin -- de Sub3
    ... <----- 2
end; -- de Sub3
begin -- de Sub2
    ... <----- 3
end; -- de Sub2
begin -- de Example
    ... <----- 4
end. -- de Example

```

Os ambientes de referenciamento dos pontos de programa indicados são:

<i>Ponto</i>	<i>Ambiente de referenciamento</i>
1	X e Y de Sub1, A e B de Example
2	X de Sub3, (X de Sub2 está oculto), A e B de Example
3	X de Sub2, A e B de Example

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 Sub3 (ele está menos profundamente aninhado), o escopo de Sub1 não é um ancestral estático de Sub3, então 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 não terminado ainda. 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 prévias de subprogramas.

Considere o seguinte programa de exemplo. Assuma que as únicas chamadas a funções são: main chama sub2, que chama sub1.

```

void sub1() {
    int a, b;
    ... <----- 1
} /* Fim de sub1 */
void sub2() {
    int b, c;
    ... <----- 2
    sub1;
}

```

```
    } /* end of 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 de constante 3.14159.

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 grande programa, 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 em:

```
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 trocado, apenas uma linha deve ser modificada (a variável `len`), independentemente do número de vezes em que ela é usada no programa. Esse é outro exemplo dos benefícios 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.

O Fortran 95 permite apenas que expressões constantes sejam usadas como valores de suas constantes nomeadas. Essas expressões constantes podem conter constantes nomeadas previamente declaradas, valores constantes e operadores. A razão para a restrição a constantes e expressões constantes em Fortran 95 é ele usar vinculação estática de valores às constantes nomeadas. Constantes nomeadas em linguagens que usam vinculação estática de valores são algumas vezes chamadas de **constantes de manifesto**.

Ada e C++ permitem a vinculação dinâmica de valores a constantes nomeadas. Isso permite expressões contendo variáveis serem 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` é 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`, dinamicamente vinculadas a valores, podem ter valores atribuídos a elas na declaração ou com um construtor estático¹². 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.

Ada permite constantes nomeadas de enumeração e tipos estruturados, discutidos no Capítulo 6.

A discussão de valores vinculados a constantes nomeadas naturalmente leva ao tópico de inicialização, porque vincular um valor a uma constante nomeada é o mesmo processo, exceto que é permanente.

Em muitos casos, é conveniente para as variáveis ter valores antes de o código do programa ou subprograma nos quais 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 é também 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 sensibilidade à capitalização e o relacionamento de nomes com palavras especiais, que são palavras reservadas ou palavras-chave, são as questões de projeto para nomes.

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.

¹² Construtores estáticos em C# rodam em algum ponto indeterminado antes de a classe ser instanciada.

A vinculação é a associação de atributos com entidades de programa. O conhecimento dos tempos de vinculação de atributos a entidades é essencial para 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 uma maior flexibilidade, às custas da 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.

O escopo estático é um recurso central do 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 da legibilidade, confiabilidade e eficiência.

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 a capitalização?
3. De que forma as palavras reservadas são melhores do que as palavras-chave?
4. O que é um apelido?
5. Que categoria de variáveis de referência em C++ é sempre composta de apelidos?
6. O que é o lado esquerdo de uma variável? O que é o lado direito?
7. Defina *vinculação* e *tempo de vinculação*.
8. Após o projeto e implementação de uma linguagem, quais são os quatro tipos de vinculações que podem ocorrer em um programa?
9. Defina *vinculação estática* e *vinculação dinâmica*.
10. Quais são as vantagens e desvantagens de declarações implícitas?
11. Quais são as vantagens e desvantagens da vinculação de tipos dinâmica?
12. Defina *variáveis estáticas, dinâmicas da pilha, dinâmicas do monte explícitas e dinâmicas do monte implícitas*.
13. Defina *tempo de vida, escopo, escopo estático e escopo dinâmico*.
14. Como a referência a uma variável não local em um programa com escopo estático está conectada a sua definição?
15. Qual é o problema geral do escopo estático?
16. O que é o ambiente de referenciamento de uma sentença?
17. O que é um ancestral estático de um subprograma? O que é um ancestral dinâmico de um subprograma?
18. O que é um bloco?
19. Quais são as vantagens e as desvantagens do escopo dinâmico?
20. Quais são as vantagens das constantes nomeadas?

CONJUNTO DE PROBLEMAS

1. Decida qual das seguintes formas de identificadores é mais legível e justifique sua decisão.

SumOfSales
sum_of_sales
SUMOFSALES

2. Algumas linguagens de programação não têm tipos. Quais são as vantagens e desvantagens óbvias de não ter 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 fortemente 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 Ada:

```
procedure Main is
    X : Integer;
    procedure Sub3; -- Essa é uma declaração de Sub3
                    -- Ela permite que Sub3 a chame
    procedure Sub1 is
        X : Integer;
        procedure Sub2 is
            begin -- de Sub2
            ...
            end; -- de Sub2
        begin -- de Sub1
        ...
        end; -- de Sub1
    procedure Sub3 is
        begin -- de Sub3
        ...
        end; -- de Sub3
    begin -- de Main
    ...
    end; -- de Main
```

Assuma que a execução desse programa é na seguinte ordem de unidades:

Main chama Sub1
Sub1 chama Sub2
Sub2 chama Sub3

- a. Assumindo escopo estático, a seguir, qual declaração de X é a correta para uma referência a X?

- i. Sub1
 ii. Sub2
 iii. Sub3
 b. Repita a parte a, mas assuma escopo dinâmico.
7. Assuma que o seguinte programa Ada foi compilado e executado usando regras de escopo estático. Que valor de X é impresso no procedimento Sub1? Sob regras de escopo dinâmico, qual o valor de X impresso no procedimento Sub1?

```
procedure Main is
  X : Integer;
  procedure Sub1 is
    begin -- de Sub1
    Put(X);
    end; -- de Sub1
  procedure Sub2 is
    X : Integer;
    begin -- de Sub2
    X := 10;
    Sub1
    end; -- de Sub2
  begin -- de Main
  X := 5;
  Sub2
  end; -- de Main
```

8. Considere o programa:

```
procedure Main is
  X, Y, Z : Integer;
  procedure Sub1 is
    A, Y, Z : Integer;
    procedure Sub2 is
      A, B, Z : Integer;
      begin -- de Sub2
      ...
      end; -- de Sub2
    begin -- de Sub1
    ...
    end; -- de Sub1
  procedure Sub3 is
    A, X, W : Integer;
    begin -- de Sub3
    ...
    end; -- de Sub3
  begin -- de Main
  ...
  end; -- de Main
```

Liste todas as variáveis, com as unidades de programa onde elas estão declaradas, visíveis nos corpos de Sub1, Sub2 e Sub3, assumindo que o escopo estático esteja sendo usado.

9. Considere o programa:

```
procedure Main is
    X, Y, Z : Integer;
    procedure Sub1 is
        A, Y, Z : Integer;
        begin -- de Sub1
        ...
        end; -- de Sub1
    procedure Sub2 is
        A, X, W : Integer;
        procedure Sub3 is
            A, B, Z : Integer;
            begin -- de Sub3
            ...
            end; -- de Sub3
        begin -- de Sub2
        ...
        end; -- de Sub2
    begin -- de Main
    ...
end; -- de Main
```

Liste todas as variáveis, com as unidades de programa onde elas estão declaradas, visíveis nos corpos de Sub1, Sub2 e Sub3, assumindo que o escopo estático está sendo usado.

10. Considere o programa em C:

```
void fun(void) {
    int a, b, c; /* definição 1 */
    ...
    while (...) {
        int b, c, d; /*definition 2 */
        ... <----- 1
        while (...) {
            int c, d, e; /* definition 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;
    ...
}

```

Dada as seguintes sequências de chamadas e assumindo 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 ela foi definida.

- main chama fun1; fun1 chama fun2; fun2 chama fun3.
 - main chama fun1; fun1 chama fun3.
 - main chama fun2; fun2 chama fun3; fun3 chama fun1.
 - main chama fun3; fun3 chama fun1.
 - main chama fun1; fun1 chama fun3; fun3 chama fun2.
 - main chama fun3; fun3 chama fun2; fun2 chama fun1.
12. Considere o programa:

```

procedure Main is
    X, Y, Z : Integer;
    procedure Sub1 is
        A, Y, Z : Integer;
        begin -- de Sub1
        ...
    end; -- de Sub1
    procedure Sub2 is
        A, B, Z : Integer;
        begin -- de Sub2
        ...
    end; -- de Sub2
    procedure Sub3 is
        A, X, W : Integer;
        begin -- de Sub3
        ...
    end; -- de Sub3
    begin -- de Main
    ...

```

```
end; -- de Main
```

Dada as seguintes sequências de chamadas e assumindo 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 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 usa 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 referencia variáveis definidas em todos os seus subprogramas que o envolvem no aninhamento.
4. Escreva uma função em C que inclua a seguinte sequência de sentenças:

```
x = 21;  
int x;  
x = 42;
```

Rode o programa e explique os resultados. Reescreva o mesmo código em C++ e Java e compare os resultados.

5. 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**.
6. Escreva três funções em C ou C++: uma que declare um grande vetor estaticamente, outra que declare o mesmo grande vetor na pilha e outra que crie o mesmo grande vetor no monte. Chame cada um desses subprogramas um grande número de vezes (ao menos 100 mil vezes) e mostre na tela o resultado necessário para cada um. Explique os resultados.

Capítulo 6

Tipos de Dados

- 6.1** Introdução
- 6.2** Tipos de dados primitivos
- 6.3** Cadeias de caracteres
- 6.4** Tipos ordinais definidos pelo usuário
- 6.5** Tipos de matrizes
- 6.6** Matrizes associativas
- 6.7** Registros
- 6.8** Uniões
- 6.9** Ponteiros e referências
- 6.10** Verificação de tipos
- 6.11** Tipagem forte
- 6.12** Equivalência de tipos
- 6.13** Teoria e tipos de dados

Este capítulo primeiro introduz 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, os detalhes dos tipos de dados estruturados – especificamente matrizes, registros 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. Métodos de implementação para tipos de dados algumas vezes têm um impacto significativo no projeto desses tipos. Logo, a implementação dos diversos tipos de dados é outra parte importante deste capítulo, especialmente para a implementação de matrizes.

A seguir, é feita uma minuciosa investigação da verificação de tipos, tipagem forte e das regras de equivalência de tipos. A última seção do capítulo introduz o básico sobre a teoria de tipos de dados.

6.1 INTRODUÇÃO

Um **tipo de dado** define uma coleção de valores de dados e um conjunto de operações pré-definidas sobre eles. Programas de computador produzem resultados por meio da manipulação de dados. Um fator importante em determinar a facilidade com a qual eles realizam suas tarefas é o quanto bem os tipos de dados disponíveis na linguagem usada casam com os objetos no espaço do problema no mundo real. Logo, é crucial uma linguagem oferecer suporte para uma coleção apropriada de tipos e estruturas de dados.

Os conceitos contemporâneos de tipagem de dados têm evoluído nos últimos 55 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 do Fortran pré-90, as listas ligadas e as árvores binárias eram implementadas com vetores.

As estruturas de dados do COBOL deram o primeiro passo na direção oposta do modelo do 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. Isso, desde então, foi incorporado à Ada e ao Fortran. Os projetistas de PL/I incluíam muitos tipos de dados, com a intenção de suportar uma grande faixa de aplicações. Uma abordagem melhor, introduzida no ALGOL 68, é fornecer alguns tipos básicos e operadores de definição de estrutura flexíveis que permitem a um programador projetar uma estrutura de dados para cada necessidade. Essa era 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 um 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 do 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, direcionada pelo sistema de tipos da linguagem, são discutidos na Seção 6.10. Um segundo uso é a assistência que o sistema de tipos fornece 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 de um sistema de tipos é a documentação. As declarações de tipo em um programa documentam informação 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 com cada expressão na linguagem e inclui suas regras para equivalência e compatibilidade de tipos. Certamente uma das partes mais importantes para entender a semântica de uma linguagem de programação é entender seu sistema de tipos.

Os dois tipos de dados estruturados (não escalares) são as matrizes e os registros, apesar de a popularidade das matrizes associativas ter aumentado nos últimos anos. 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 uma 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 para construir o código para as operações de alocação e liberação.

A palavra *objeto* é normalmente associada com o valor de uma variável e com o espaço que ocupa. Neste livro, entretanto, reservamos a palavra *objeto* exclusivamente para instâncias de tipos de dados abstratos definidos pelo usuário, em vez de também usá-la para os valores de variáveis de tipos pré-de-

finidos. Em linguagens orientadas a objetos, todas as instâncias de todas as classes, sejam pré-definidas ou definidas pelo usuário, são chamadas de objetos. Objetos são discutidos em detalhes nos Capítulos 11 e 12.

Nas seções seguintes, todos os tipos de dados comuns são discutidos. Para a maioria, questões particulares ao tipo são levantadas. Para todos, um ou mais exemplos de projeto são descritos. Uma questão de projeto é fundamental para todos os tipos de dados: Que operações são fornecidas para variáveis do 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 de hardware – por exemplo, a maioria dos tipos inteiros. Outros requerem apenas um pouco de suporte externo ao hardware para sua implementação.

Os tipos de dados primitivos de uma linguagem são usados, com um ou mais construtores de tipo, para fornecer os tipos estruturados.

6.2.1 Tipos numéricos

Muitas das primeiras linguagens de programação têm 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**. Muitos computadores agora suportam diversos tamanhos de inteiros. Por exemplo, Java inclui quatro tamanhos inteiros com sinal: `byte`, `short`, `int` e `long`. Algumas linguagens, como C++ e C#, incluem tipos inteiros sem sinal, simplesmente 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 por hardware. Um exemplo de um tipo inteiro não suportado diretamente por hardware é o tipo inteiro longo de Python. Valores desse tipo podem ter um tamanho ilimitado. Valores inteiros longos podem ser especificados como literais, como no seguinte exemplo:

243725839182756281923L

Além disso, operações aritméticas inteiras que produzem valores muito grandes para serem representados com o tipo `int` são armazenadas 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 qualquer 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. É claro, nenhum desses números pode ser representado precisamente em qualquer espaço finito. 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 binários¹. Outro problema com tipos ponto flutuante é a perda de precisão por meio de operações aritméticas. Para mais informações sobre os problemas na notação de ponto flutuante, procure por qualquer livro sobre análise numérica.

Valores de ponto flutuante são representados como frações expoentes, uma forma emprestada da notação científica. Computadores mais antigos usavam uma variedade de diferentes representações para valores de ponto flutuante. Entretanto, a maioria das máquinas mais novas usam o formato padrão para ponto flutuante da IEEE chamado IEEE Floating-Point Standard 754. Os implementadores de linguagens usam quaisquer representações suportadas por hardware. A maioria das linguagens inclui dois tipos de ponto flutuante, chamados de `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 partes fracionárias maiores e/ou uma faixa de expoentes maior são necessárias. 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 ponto flutuante é definida em termos de precisão e faixa. **Precisão** é a exatidão da

¹ 0.1 em decimal é 0.0001100110011... em binário.

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 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 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 o COBOL. C# também tem um tipo de dados decimal.

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

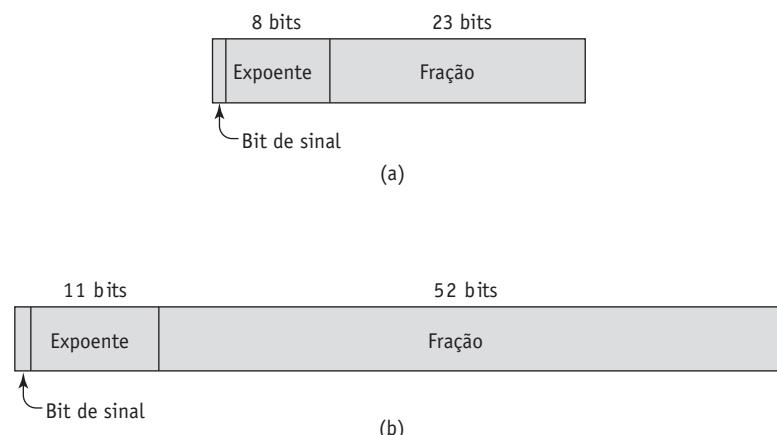


Figura 6.1 Formatos de ponto flutuante da IEEE: (a) precisão simples, (b) precisão dupla.

um tipo de ponto flutuante, como vimos na Seção 6.2.1.1. As desvantagens dos tipos decimais são que a faixa de valores é restrita porque nenhum expoente é permitido, e sua representação em memória é 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, 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. Gastam-se ao menos quatro bits para codificar um dígito decimal. Logo, armazenar um número decimal codificado de seis dígitos requer 24 bits de memória. Entretanto, gastam-se apenas 20 bits para armazenar o mesmo número em binário².

As operações em valores decimais são feitas em hardware em máquinas que têm tais capacidades; caso não tenham, elas são simuladas em software.

6.2.2 Tipos booleanos

Tipos **booleanos** são talvez 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 o C89, no qual 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 do C99 e do C++ terem um tipo booleano, também permitem que expressões numéricas sejam usadas como se fossem booleanas. Esse não é o caso com 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 ASCII (Padrão Americano de Codificação para Intercâmbio de Informação – 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

² É claro, a menos que um programa precise manter um grande número de grandes valores decimais, a diferença é insignificante.

é outra codificação de 8 bits para caracteres, mas ele permite 256 caracteres diferentes. Ada 95 usa o ISO 8859-1.

Por causa da 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 usar o conjunto de caracteres Unicode. Desde então, ele foi adotado em JavaScript, Python, Perl e C#.

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.

6.3 CADEIAS DE CARACTERES

Um **tipo cadeia de caracteres** é um tipo 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 saída de todos os tipos de dados é geralmente feita em termos de cadeias. É claro, cadeias de caracteres são também um tipo essencial para todos os programas que realizam manipulação de caracteres.

6.3.1 Questões de projeto

As duas questões de projeto específicas às cadeias de caracteres mais importantes são:

- As cadeias devem ser apenas 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 comuns em cadeias são: atribuição, concatenação, referência a subcadeias, comparação e casamento de padrões.

Referências a subcadeias são discutidas no contexto mais geral de vetores, onde as referências a subcadeias são chamadas de **fatias**.

Em geral, tanto a operação de atribuição quanto a de comparação de cadeias de caracteres são complicadas pela possibilidade de operandos de diferentes tamanho. 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 problemas em se lembrarem delas.

O casamento de padrões é outra operação fundamental das cadeias de caracteres. 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 abordagem usada por C e C++.

C e C++ usam vetores de caracteres (**char**) para armazenar cadeias de caracteres. Essas linguagens fornecem uma coleção de operações em cadeias por meio de uma biblioteca padrão. A maioria dos usos de cadeias e das funções das bibliotecas usam a convenção de que as cadeias de caracteres são terminadas com um caractere especial, nulo, representado com um zero. Essa é uma alternativa em relação a manter o tamanho das variáveis do tipo cadeia. As operações de bibliotecas simplesmente conduzem suas operações até que o caractere nulo apareça na cadeia sendo 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 seguinte declaração:

```
char str[] = "apples";
```

Nesse exemplo, **str** é um vetor de elementos do tipo **char**, especificamente **apples**₀, onde **0** é o caractere nulo.

Umas 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 lexicograficamente (pela ordem dos códigos de caracteres) duas cadeias; e **strlen**, que retorna o número de caracteres, não contando o caractere nulo, em uma cadeia. 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 de C, também disponíveis em C++, são inherentemente inseguras e têm levado a diversos erros de programação. O problema é que as funções nessa biblioteca que movem dados de cadeias não

NOTA HISTÓRICA

SNOBOL 4 foi a primeira linguagem bastante conhecida a suportar o casamento de padrões.

verificam transbordamentos (*overflows*) no destino. Por exemplo, considere a seguinte chamada a `strcpy`:

```
strcpy(dest, src);
```

Se o tamanho de `dest` é 20 e o de `src` é 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 com muitas das outras funções na biblioteca de cadeias de C. Além das cadeias no estilo de C, C++ também suporta cadeias por meio de sua biblioteca de classes padrão, similar àquela 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 a biblioteca de cadeias de C.

O Fortran 95 trata cadeias como um tipo primitivo e fornece atribuição, operadores relacionais, concatenação e operações de referências a subcadeias para elas. Em Java, cadeias são suportadas pelas classes `String`, cujos valores são cadeias constantes, e `StringBuffer`, cujos valores são modificáveis e mais parecidas com vetores de caracteres. Esses valores são especificados com métodos da classe `StringBuffer`. C# e Ruby incluem classes similares àquelas de Java para representar cadeias.

Python também 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 às matrizes de caracteres. Entretanto, as cadeias em Python são imutáveis, de modo similar aos objetos da classe `String` em Java.

Perl, JavaScript, Ruby e PHP incluem operações de casamento de padrões pré-definidas. 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 completo sobre esse tipo de expressões de casamento de padrões (Friedl, 2006). Nesta seção, fornecemos apenas uma breve visão no estilo dessas expressões por meio de dois exemplos relativamente simples.

Considere expressão de um 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 específica 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 literal³. O ponto de interrogação quantifica aquilo 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 são incluídos nas bibliotecas de classes de C++, Java, Python e C#.

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, sendo chamada de uma **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++, a classe pré-definida `String` em Ruby e para a biblioteca de classes do .NET disponível para o C#.

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 que as cadeias em C usam um caractere especial para indicar o final da cadeia, em vez de manter o tamanho da cadeia.

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 **cadeias de tamanho dinâmico**, opção que requer a sobrecarga da alocação e liberação de armazenamento dinâmico, mas fornece a flexibilidade máxima.

Ada 95 suporta as três opções de tamanho de cadeias.

³ Deve ser usado um “caractere de escape” para o ponto, usando uma barra invertida, porque o ponto tem um significado especial em uma expressão regular.

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 pré-definida que faz o que `strcpy` em C faz. Então, uma simples atribuição de uma cadeia para outra necessaria de um laço de repetição. A adição de cadeias como um tipo primitivo de uma linguagem não é custoso, 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, fornecer cadeias por meio de uma biblioteca padrão é quase tão conveniente quanto tê-las como um tipo primitivo.

Operações sobre cadeias como um 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 feitas 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 limitado dinâmicas requerem 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 requerem 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.

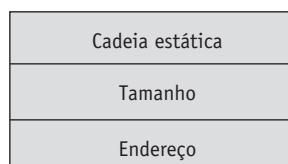


Figura 6.2 Descritor em tempo de compilação para cadeias estáticas.

Cadeia dinâmica de tamanho limitado
Tamanho máximo
Tamanho atual
Endereço

Figura 6.3 Descritor em tempo de execução para cadeias dinâmicas de tamanho limitado.

As cadeias de tamanho dinâmico limitado de C e C++ não requerem 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 a sua faixa nessas linguagens.

Cadeias de tamanho estático e de tamanho dinâmico limitado não requerem 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 apenas um processo de alocação simples é necessário.

Cadeias de tamanho dinâmico requerem um gerenciamento de armazenamento mais complexo. O tamanho de uma cadeia e, dessa forma, o armazenamento com a qual ela está vinculada, devem crescer e encolher dinamicamente.

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 com esse 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 liberação de segmentos de tamanhos variáveis é discutido na Seção 6.9.9.3.

Apesar de o método que usa listas encadeadas requerer 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 ORDINAIS DEFINIDOS PELO USUÁRIO

Um **tipo ordinal** é um no qual a faixa de valores possíveis pode ser facilmente associada com o conjunto dos inteiros positivos. Em Java, por exemplo, os tipos ordinais primitivos são `integer`, `char` e `boolean`. Existem dois tipos ordinais definidos pelo usuário suportados pelas linguagens de programação: enumerações e subfaixas.

6.4.1 Tipos enumeração

Um **tipo enumeração** é um 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.

As questões de projeto para tipos enumeração são:

- 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 para inteiros?
- Existem outros tipos que são convertidos para um tipo enumeração?

Todas essas questões de projeto são relacionadas à verificação de tipos. Se uma variável de enumeração é convertida para um tipo numérico, existe pouco controle sobre sua faixa de operações legais ou sobre sua faixa de valores. Se um valor do tipo `int` é convertido para um tipo enumeração, uma variável do tipo enumeração pode ter quaisquer valores inteiros atribuídos a ela, independentemente de ela representar uma constante de enumeração ou não.

6.4.1.1 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 em

```
int red = 0, blue = 1;
```

No programa, poderíamos usar `red` e `blue` como se fossem tipos de cores. O problema com essa abordagem é que não definimos um tipo para nossas cores, não existe uma verificação de tipos quando elas são usadas. Por exemplo, seria permitido adicionar as duas juntas, apesar de 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 prevêido tornando-as constantes nomeadas.

C e Pascal foram as primeiras linguagens amplamente usadas a incluir um tipo de dados de enumeração. C++ inclui os tipos 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 para `int` quando são inseridos em um contexto inteiro. Isso permite seu uso em qualquer expressão numérica. Por exemplo, se o valor atual de `myColor` é `blue`, a expressão

```
myColor++
```

atribuiria `green` para `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 para um tipo enumeração em C++. Por exemplo,

```
myColor = 4;
```

é ilegal em C++. Essa atribuição seria legal se o lado direito tivesse sido convertido para o tipo `colors`. Isso previne alguns erros em potencial.

Constantes de enumeração em C++ podem aparecer em apenas um tipo enumeração no mesmo ambiente de referenciamento.

Em Ada, é permitido que os literais de enumeração apareçam em mais de uma declaração no mesmo ambiente de referenciamento. Eles são chamados de **literais sobrecarregados**. A regra para resolver a sobrecarga – isso é, decidir o tipo de uma ocorrência de um desses literais é que ele deve ser determinável a partir do contexto de sua aparição. Por exemplo, se um literal sobrecarregado e uma variável de enumeração são comparados, o tipo do literal é resolvido como sendo do tipo da variável. Em alguns casos, o programador deve indicar alguma especificação de tipo para uma ocorrência de um literal sobrecarregado para evitar um erro de compilação.

Como nem os literais nem as variáveis de enumeração em Ada são convertidos para inteiros, tanto a faixa de operações quanto de valores dos tipos enumeração são restritas, permitindo muitos erros de programação serem detectados pelo compilador.

Um tipo enumeração foi adicionado à linguagem Java em sua versão 5.0, em 2004. Todos os tipos enumeração em Java são subclasses implícitas da classe pré-definida `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 possíveis valores de uma enumeração são as únicas possíveis instâncias da classe. 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 para qualquer outro tipo.

Tipos enumeração em C# são parecidos com os de C++, exceto pelo fato de que nunca são convertidos para 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.

É interessante que nenhum dos tipos de linguagens de *scripting* relativamente recentes inclui tipos enumeração, entre elas Perl, JavaScript, PHP, Python, Ruby e Lua. Mesmo Java já tinha 10 anos quando os tipos enumeração foram adicionados.

6.4.1.2 Avaliação

Tipos enumeração podem fornecer vantagens tanto em relação à legibilidade quanto à confiabilidade. A legibilidade é melhorada de uma maneira muito direta. Valores nomeados são facilmente reconhecidos, enquanto valores codificados não.

Na área da confiabilidade, os tipos enumeração de Ada, C# e Java 5.0 fornecem duas vantagens. Primeiro, nenhuma operação aritmética é permitida em tipos enumeração. Isso previne adicionar dias da semana, por exemplo. Segundo, nenhuma variável de enumeração pode ter um valor atribuído a ela fora de sua faixa definida. Se a enumeração `colors` tem 10 constantes de enu-

meração e usa 0..9 como seus valores internos, nenhum número maior do que 9 pode ser atribuído a variável do tipo `colors`.

Como C trata as variáveis de enumeração como variáveis inteiras, ele não fornece as duas vantagens.

C++ é um pouco melhor. Valores numéricos podem ser atribuídos a variáveis do tipo enumeração apenas se eles puderem ser convertidos para o tipo da variável que está recebendo a atribuição. Valores numéricos atribuídos a variáveis do tipo enumeração são verificados para determinar se eles 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 efetiva. Por exemplo,

```
enum colors {red = 1, blue = 1000, green = 100000}
```

Nesse exemplo, um valor atribuído a uma variável do tipo `colors` será verificada apenas para determinar se ela está na faixa de 1..100000.

Os tipos enumeração de Ada, C# e Java 5.0 são melhores do que os de C++, porque as variáveis do tipo enumeração nunca são convertidas para tipos inteiros.

6.4.2 Tipos subfaixa

Um **tipo subfaixa** é uma subsequência contígua de um tipo ordinal. Por exemplo, 12..14 é uma subfaixa do tipo inteiro. Tipos subfaixa foram introduzidos pelo Pascal e incluídos em Ada. Não existem questões de projeto específicas para os tipos subfaixa.

6.4.2.1 Projeto de Ada

Em Ada, subfaixas são incluídas na categoria chamada de subtipos. Como mencionado no Capítulo 5, os subtipos não são novos tipos; em vez disso, eles são novos nomes para versões possivelmente restritas de tipos existentes. Por exemplo, considere as declarações:

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
subtype Weekdays is Days range Mon..Fri;
subtype Index is Integer range 1..100;
```

Nesses exemplos, a restrição nos tipos existentes está na faixa de valores possíveis. Todas as operações definidas para o tipo ancestral também são definidas para o subtipo, exceto a atribuição de valores fora da faixa especificada. Por exemplo, no código

```
Day1 : Days;
Day2 : Weekdays;
...
Day2 := Day1;
```

a atribuição é legal a menos que o valor de `Day1` seja `Sat` ou `Sun`.

O compilador deve gerar código de verificação de faixa para cada atribuição a uma variável de subfaixa. Enquanto os tipos são verificados em relação à sua compatibilidade em tempo de compilação, as subfaixas requerem verificação de faixas em tempo de execução.

Um dos usos mais comuns dos tipos ordinais definidos pelo usuário é para o índice de vetores e matrizes, conforme será discutido na Seção 6.5. Eles também podem ser usados para variáveis de laços de repetição. De fato, subfaixas de tipos ordinais são a única maneira pela qual a faixa das variáveis de laços `for` em Ada pode ser especificada.

6.4.2.2 Avaliação

Tipos subfaixa melhoram a legibilidade ao tornar claro aos leitores que as variáveis de subtipos podem armazenar apenas certas faixas de valores. A confiabilidade é aumentada com tipos subfaixa, porque a atribuição de um valor a uma variável de subfaixa que está fora da faixa especificada é detectada como um erro, seja pelo compilador (no caso do valor atribuído ser um valor literal) ou pelo sistema em tempo de execução (no caso de uma variável ou expressão). É estranho que nenhuma linguagem contemporânea, exceto Ada 95, tenha tipos subfaixa.

6.4.3 Implementação de tipos ordinais definidos pelo usuário

Conforme discutido, tipos enumeração são normalmente implementados como inteiros. Sem restrições nas faixas de valores e de operações, eles não oferecem um aumento na confiabilidade.

Tipos subfaixa são implementados exatamente da mesma forma que seus tipos ancestrais, exceto que as verificações de faixas devem ser implicitamente incluídas pelo compilador em cada atribuição de uma variável ou de uma expressão a uma variável subfaixa. Esse passo aumenta o tamanho do código e o tempo de execução, mas normalmente se considera que o custo vale a pena. Além disso, um compilador com boa otimização pode otimizar parte da verificação.

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 de um mesmo tipo e as referências a eles são especificadas usando expressões de índices. Se qualquer uma das expressões de índices em uma referência incluir variáveis, a referência requererá um cálculo adicional em tempo de execução para determinar o endereço da posição de memória sendo referenciada.

Em muitas linguagens, como C, C++, Java, Ada e C#, todos os elementos de uma matriz precisam ser do mesmo tipo. Nas, ponteiros e referências são restritos, para que possam apontar ou referenciar um único tipo, de forma que os objetos ou valores de dados sendo 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.

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 elas têm seu armazenamento alocado?
- Que tipos de fatias são permitidas (caso sejam)?

Nas seções seguintes, exemplos de escolhas de projeto feitas para as matrizes das linguagens de programação mais comuns são discutidos.

6.5.2 Matrizes e índices

Elementos específicos de uma matriz são referenciados por meio de um mecanismo sintático de dois níveis, onde 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

`nome_matriz(lista_valores_índices) → elemento`

A sintaxe das referências a uma matriz é bastante universal: o nome da matriz é seguido por uma lista de índices, envoltos ou em parênteses ou em colchetes. Na maioria das linguagens que fornecem matrizes multidimensionais como matrizes de matrizes, cada índice aparece em seu próprio colchete. Um problema com o uso de parênteses para envolver 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

NOTA HISTÓRICA

Os projetistas do 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 representando colchetes.

exatamente como 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 os índices de matrizes em Ada, ambos os leitores de programas e 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, resultando em uma redução na legibilidade.

Os projetistas de Ada escolheram especificamente os parênteses para envolver índices de forma que haveria uma 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 um elemento em particular da matriz e as chamadas mapeiam os parâmetros reais para a definição da função e, por fim, a 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 matriz: o do elemento e o do índice. O tipo dos índices é normalmente uma subfaixa de inteiros, mas Ada permite que qualquer tipo ordinal seja usado como índice, como tipos booleanos, caracteres e enumerações. Por exemplo, em Ada, alguém poderia ter:

```
type Week_Day_Type is (Monday, Tuesday, Wednesday,
                        Thursday, Friday);
type Sales is array (Week_Day_Type) of Float;
```

Um laço de repetição `for` em Ada pode usar qualquer variável de tipo ordinal para seu contador, como veremos no Capítulo 8. Isso permite que matrizes com índices de tipos ordinais sejam convenientemente processadas.

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 obrigar a verificação de faixas é um fator importante na confiabilidade. A maioria das linguagens contemporâneas não especifica verificação de faixas de índices, mas Java, ML e C# o fazem. Por padrão, Ada verifica a faixa de todos os índices, mas esse recurso pode ser desabilitado pelo programador.

Índices em Perl são um pouco não usuais pois, apesar de os nomes de todas as matrizes começarem com uma 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 cífrões 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 é algumas vezes 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; no Fortran 95, ele é padronizado como 1, mas pode ser modificado para qualquer literal inteiro. Em outras, os limites inferiores podem ser especificados pelo programador.

Existem cinco categorias de matrizes, baseadas na vinculação das faixas de índices, na vinculação ao armazenamento e a partir de onde o armazenamento é alocado. Os nomes das categorias indicam as escolhas de projeto para essas três questões. Nas primeiras quatro 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. Mantenha em mente que, quando as faixas de índices são fixas, a matriz não pode modificar seu tamanho.

Uma **matriz estática** é uma 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 vinculado por todo o tempo de execução do programa.

Uma **matriz dinâmica da pilha fixa** é uma 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 da pilha** é uma na qual tanto as faixas de índices quanto a alocação de armazenamento são vinculadas dinamicamente em tempo de elaboração. Uma vez que as faixas de índices são vinculadas e o armazenamento é alocado, entretanto, ambas permanecem fixas durante o tempo de vida da variável. A vantagem das matrizes dinâmicas da pilha em relação às estáticas e às dinâmicas da pilha fixas é a flexibilidade. O tamanho de uma matriz não precisa ser conhecido até ela ser usada.

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 tanto as faixas de índices

quanto as vinculações de armazenamento são feitas quando o programa de usuário requisita-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 do que o tempo de alocação a partir da pilha.

Uma **matriz dinâmica do monte** é uma 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 liberação levam mais tempo e podem acontecer muitas vezes durante a execução do programa. Exemplos das cinco 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. As matrizes em Ada podem ser dinâmicas da pilha, como a seguir:

```
Get(List_Len);
declare
    List : array (1..List_Len) of Integer;
begin
    ...
end;
```

Nesse exemplo, o usuário informa o número de elementos desejados para a matriz `List`, que são então alocados dinamicamente quando a execução alcança o bloco `declare`. Quando a execução alcança o fim do bloco, a matriz `List` é liberada.

C e C++ também fornecem matrizes dinâmicas do monte fixas. As 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 o operador `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 6.9.5.

O Fortran 95 também oferece suporte para matrizes dinâmicas do monte fixas. Em Java, todas as matrizes 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.

NOTA HISTÓRICA

O 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 projetistas do Fortran I desenvolveram um método muito rápido para acessar os elementos de matrizes de até três dimensões, usando os três registradores de índices do IBM 704. O Fortran IV foi implementado pela primeira vez em um IBM 7094, que tinha sete registradores de índices. Isso permitia aos projetistas do Fortran IV permitirem matrizes com até sete índices. A maioria das outras linguagens contemporâneas não determina esses limites.

C# inclui uma segunda classe de matrizes, `ArrayList`, que fornece matrizes dinâmicas da pilha. Objetos dessa classe são criados sem elementos, como em

```
ArrayList intList = new ArrayList();
```

Elementos são adicionados a esse objeto por meio do método `Add`, como em

```
intList.Add(nextOne);
```

Java inclui uma estrutura similar ao `ArrayList` de C#, exceto que o uso de índices não é suportado – métodos de acesso (`get` e `set`) devem ser usados para acessar os elementos.

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 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 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 contíguos. Por exemplo, suponha que tenhamos uma matriz chamada `list` que tem elementos com os índices 0..9⁴. Considere a 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 requerem 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 podem ser apagados. Uma referência a um elemento não existente em Python resulta em um erro em tempo de execução, enquanto uma referência similar em Ruby e Lua leva a `nil` e nenhum erro é informado.

6.5.4 Inicialização de matrizes

Algumas linguagens fornecem os meios para inicializar matrizes no momento em que seu armazenamento é alocado. Em Fortran 95, uma matriz pode ser inicializada atribuindo a ela uma matriz agregada em sua declaração.

⁴ A faixa de índices poderia facilmente ter sido entre 1000..1009.

Uma matriz agregada no Fortran 95 para uma matriz de uma dimensão é uma lista de literais delimitados por parênteses e barras. Por exemplo, poderíamos ter

```
Integer, Dimension (3) :: List = (/0, 5, 5/)
```

C, C++, Java e C# também permite inicialização de suas matrizes, mas com uma nova virada: na declaração C

```
int list [] = {4, 5, 7, 83};
```

o compilador define o tamanho da matriz. Isso supostamente seria uma conveniência, mas não é livre de custos. Ela efetivamente remove 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. Nesse caso, a matriz é composta de ponteiros para caracteres. Por exemplo

```
char *names [] = {"Bob", "Jake", "Darcie"};
```

Esse exemplo ilustra a natureza dos literais de caracteres em C e C++. No exemplo anterior, de um literal do tipo cadeia sendo 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"};
```

Ada fornece dois mecanismos para inicializar matrizes na sentença de declaração: listando os valores na ordem na qual serão armazenados, ou diretamente atribuindo-os a uma posição de índice usando o operador `=>`, que em Ada é chamado de flecha (**arrow**). Por exemplo, considere o seguinte:

```
List : array (1..5) of Integer := (1, 3, 5, 7, 9);
Bunch : array (1..5) of Integer := (1 => 17, 3 => 34,
                                     others => 0);
```

Na primeira sentença, todos os elementos da matriz `List` têm valores de inicialização, atribuídos nas posições dos elementos da matriz na ordem na qual eles aparecem. Na segunda, o primeiro e o terceiro elementos do vetor são inicializados usando atribuição direta, e a cláusula `others` é usada para inicializar os elementos restantes. Como no Fortran, essas listas de valores entre parênteses são chamadas de **valores agregados**.

Python inclui um mecanismo poderoso para criar matrizes chamado de **compreensões de lista**, uma ideia da notação de conjuntos. Elas apareceram pela primeira vez na linguagem de programação funcional Haskell (veja o Capítulo 15). A mecânica de uma compreensão de lista é de uma função aplicada a cada um dos elementos de uma matriz e de uma nova matriz construída a partir dos resultados. A sintaxe de uma compreensão de lista em Python é:

[expressão `for` variável_iteração `in` matriz `if` condição]

Considere o exemplo:

```
[x * x for x in range(12) if x % 3 == 0]
```

A compreensão de lista retorna a seguinte matriz:

```
[0, 9, 36, 81]
```

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.

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, 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.

Ada permite atribuições de matrizes, incluindo aquelas nas quais o lado direito é um valor agregado em vez de um nome de matriz. Ada também fornece concatenação, especificada pelo e comercial (`&`). A concatenação é definida entre duas matrizes de dimensão única e entre uma de dimensão única e um escalar. Praticamente todos os tipos em Ada têm operadores relacionais próprios para igualdade e diferença.

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

determina se duas variáveis referenciam o mesmo objeto (`is`) e outro que compara todos os objetos correspondentes nos objetos referenciados, independentemente de quão profundamente eles estão aninhados, em relação à igualdade (`==`).

Python inclui um tipo de matriz imutável chamado de **tuplas**. Sintaticamente, as tuplas se diferem das matrizes ao usar parênteses para envolver seus literais, em vez de colchetes. Semanticamente, a diferença é que as tuplas são imutáveis. Elementos individuais de uma tupla não podem ser modificados. Se duas tuplas são concatenadas, uma nova é criada para o resultado. Se uma tupla precisa ser modificada, ela pode ser convertida para uma matriz usando a função `list`. Após a mudança, ela pode ser convertida novamente para uma tupla usando a função `tuple`. Um uso de tuplas ocorre quando uma matriz precisa ser protegida em relação à escrita, como quando ela é enviada como um parâmetro para uma função externa e o usuário não quer que a função seja capaz de modificar o parâmetro.

Como em Python, os elementos das matrizes de Ruby são referências a objetos e, quando um operador Ruby `==` é usado entre duas matrizes, o resultado é verdadeiro somente se duas matrizes têm o mesmo tamanho e os elementos correspondentes são iguais. As matrizes de Ruby podem ser concatenadas com um método `Array`.

O Fortran 95 inclui algumas operações chamadas de **elementais** porque ocorrem entre pares de elementos de matrizes. Por exemplo, o operador de adição (+) entre duas matrizes resulta em uma matriz das somas dos pares de elementos das duas. Os operadores de atribuição, de aritmética e os lógicos são sobrecarregados para matrizes de quaisquer tamanhos ou formas. O Fortran 95 também inclui funções intrínsecas, ou bibliotecas, para multiplicação de matrizes, transposição de matrizes e o produto escalar de vetores.

Matrizes e suas operações estão no coração de APL; ela é a linguagem de processamento de matrizes mais poderosa já desenvolvida. Devido à sua relativa obscuridade e à sua falta de efeito 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 operandos escalares. Por exemplo,

`A + B`

é uma expressão válida, independentemente de `A` ou `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`

ϕM transpõe M (suas linhas viram suas colunas e vice-versa)
 $\div M$ inverte M

APL também inclui diversos operadores especiais que recebem outros operadores como operandos. Um desses é o operador de produto interno, especificado com um ponto (.). Ele recebe dois operandos, os quais são operadores binários. Por exemplo,

$+ . \times$

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,

$A \times B$

é o produto interno matemático de A e B (um vetor dos produtos dos elementos correspondentes de A e B). A sentença

$A + . \times B$

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 multidimensional na qual todas as linhas têm o mesmo número de elementos, todas as colunas têm o mesmo número de elementos e assim por diante. Matrizes retangulares modelam tabelas retangulares exatamente.

Uma **matriz irregular** é uma na qual o tamanho das linhas não precisa ser o mesmo. Por exemplo, uma matriz irregular pode ser composta de três linhas, uma com 5 elementos, uma com 7 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]`

Fortran, Ada e C# oferecem suporte para matrizes retangulares (C# também oferece suporte para matrizes irregulares). Nesses casos, 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

Uma **fatia** de uma matriz é alguma subestrutura dela. Por exemplo, se `A` é uma matriz, a primeira linha de `A` é uma possível fatia, assim como a última linha e a 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` (aqueles 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 segundo elemento 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]`.

O Fortran 95 suporta fatias ainda mais complicadas. Por exemplo, se `Mat` é uma matriz, colunas específicas podem ser especificadas, como em `Mat(:, 2)` que se refere à coluna com o índice 2.

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`, que 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 é 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]`.

Em Ada, apenas fatias altamente restritas são permitidas: aquelas que consistem em elementos consecutivos de uma matriz de uma dimensão. Por exemplo, se `List` é uma matriz com faixa de índices `(1..100)`, `List(5..10)` é uma fatia de `List` que consiste nos seis elementos indexados de 5 a 10. Conforme discutido na Seção 6.3.2, uma fatia de um tipo `String` é uma referência a uma subcadeia.

6.5.8 Avaliação

Matrizes são incluídas em praticamente todas as linguagens de programação. As vantagens primárias desde sua introdução no Fortran I são a inclusão de todos os tipos ordinais como possíveis tipos de índices, fatias e, é claro, 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

```
list [k]
```

Uma matriz de uma dimensão é implementada como uma lista de células de memória adjacentes. Suponha que a matriz `list` seja definida como tendo um limite inferior para a faixa de índices de valor 0. A função de acesso para `list` é geralmente construída da seguinte forma

$$\text{endereço}(\text{list}[k]) = \text{endereço}(\text{list}[0]) + k * \text{tamanho_do_elemento}$$

onde o primeiro operando da adição é a parte constante da função de acesso, e a segunda 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 feitas 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 única 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 podem precisar ser armazenadas em um descritor em tempo de execução. Se as faixas de índices de um tipo de matriz em particular são estáticas, as faixas 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 for vinculada dinamicamente, essas partes do descritor devem ser mantidas em tempo de execução.

Verdadeiras matrizes multidimensionais, 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 – normalmente, 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 comumente usadas pelas quais as matrizes multidimensionais podem ser mapeadas para uma dimensão: ordem principal de linha e ordem principal de coluna. 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 tiver os valores

```
3 4 7  
6 2 5  
1 3 8
```

Matriz
Tipo do elemento
Tipo do índice
Limite inferior do índice
Limite superior do índice
Endereço

Figura 6.4 Descritor em tempo de compilação para matrizes de uma dimensão.

ela seria armazenada usando a ordem principal de linha como

3, 4, 7, 6, 2, 5, 1, 3, 8

Na **ordem principal de coluna**, os elementos de uma matriz que tem como último índice o valor de limite inferior desse índice são armazenadas primeiro, seguidos pelos elementos do segundo valor do último índice e assim por diante. Nesse caso, a matriz é armazenada pelas colunas. Se a matriz de exemplo fosse armazenada na ordem principal de coluna, teria a seguinte ordem em memória:

3, 6, 1, 4, 2, 3, 7, 5, 8

A ordem principal de coluna é usada no Fortran, mas outras linguagens que têm matrizes multidimensionais verdadeiras usam a ordem principal de linhas.

Às vezes, é essencial conhecer a ordem de armazenamento das matrizes multidimensionais – por exemplo, quando tais matrizes são entrada e saída como unidades em Fortran. Em todos os casos, o acesso sequencial aos elementos da matriz será mais rápido se eles forem acessados na ordem em que foram armazenados, porque isso resultará em uma melhor localidade de memória⁵.

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 é o número de linhas acima dele multiplicado pelo tamanho de uma linha, somado ao número de elementos à esquerda dele. Isso é ilustrado na Figura 6.5, na qual fizemos a simplificação de que todos os limites inferiores dos índices são igual a um (normalmente, o caso no Fortran).

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

$$\begin{aligned} \text{localização}(a[i, j]) = & \text{endereço de } a[1, 1] \\ & + (((\text{número de linhas acima da } i\text{-ésima linha}) * \\ & (\text{tamanho de uma linha})) \\ & + (\text{número de elementos à esquerda da } j\text{-ésima coluna})) \\ & * \text{tamanho do elemento)} \end{aligned}$$

Como o número de linhas acima da i -ésima linha é $(i - 1)$ e o número de elementos à esquerda da j -ésima coluna $(j - 1)$, temos que

⁵ Uma melhor localidade de memória significa que menos recargas de *cache* serão necessárias.

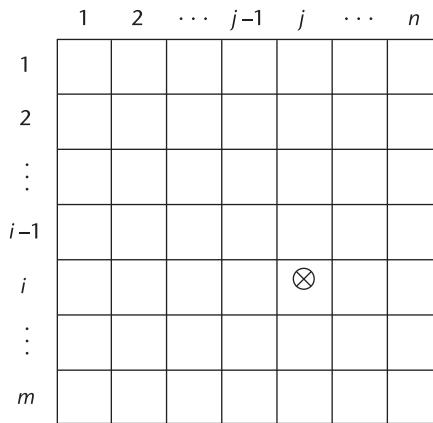


Figura 6.5 A localização do elemento $[i, j]$ em uma matriz.

$$\text{localização}(a[i, j]) = \text{endereço de } a[1, 1] + (((i - 1) * n) + (j - 1)) * \text{tamanho_do_elemento}$$

onde n é o número de elementos por linha. Isso pode ser reorganizado para a forma

$$\text{localização}(a[i, j]) = \text{endereço de } a[1, 1] - ((n + 1) * \text{tamanho_do_elemento}) + ((i * n + j) * \text{tamanho_do_elemento})$$

onde os dois primeiros termos são a parte constante e o último é a parte variável.

A generalização para limites inferiores arbitrários resulta na seguinte função de acesso:

$$\text{localização}(a[i, j]) = \text{endereço de } a[li_linha, li_col] + (((i - li_linha) * n) + (j - li_col)) * \text{tamanho_do_elemento}$$

onde li_linha é o limite inferior das linhas e li_col é o limite inferior das colunas. Isso pode ser reorganizado para a seguinte forma

$$\begin{aligned} \text{localização}(a[i, j]) &= \text{endereço de } a[li_linha, li_col] \\ &- (((li_linha * n) + li_col) * \text{tamanho_do_elemento}) \\ &+ (((i * n) + j) * \text{tamanho_do_elemento}) \end{aligned}$$

onde os primeiros dois termos são a parte constante e o último é a parte variável. Isso pode ser generalizado de forma relativamente fácil 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 caros. O descritor em tempo de compilação para uma matriz multidimensional é mostrado na Figura 6.6.

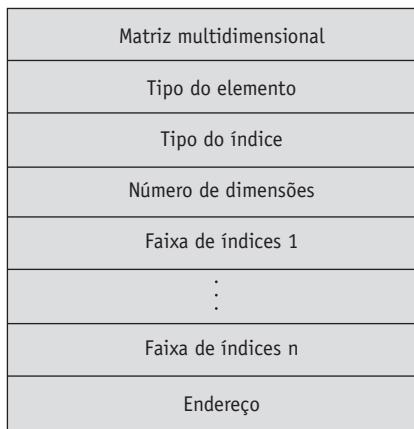


Figura 6.6 Um descritor em tempo de compilação para uma matriz multidimensional.

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 associativas, os índices nunca precisavam 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 são também suportadas diretamente por Python, Ruby e Lua, assim como pelas bibliotecas de classe padrão de Java, C++ e C#.

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 distinto: 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 usando uma notação similar à para 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 (\$). Por exemplo,

```
$salaries{"Perry"} = 58850;
```

Um novo elemento é adicionado usando a mesma forma da sentença de atribuição. Um elemento pode ser removido da dispersão com o operador **delete**, como em

```
delete $salaries{"Gary"};
```

A dispersão inteira pode ser esvaziada por meio da atribuição do literal vazio a ela, tal como em

```
@salaries = ();
```

O tamanho de uma dispersão em Perl é dinâmico: aumenta quando um novo elemento é adicionado e encolhe quando um é apagado (e também quando ela é esvaziada por meio da atribuição do literal vazio). O operador **exists** retorna verdadeiro ou falso, dependendo se sua operanda é 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, e **each** itera sobre os pares de elemento de uma dispersão.

As matrizes associativas de Python, chamadas de **dicionários**, são similares àquelas de Perl, exceto que os valores são todos referências a objetos. As matrizes associativas suportadas por Ruby são similares àquelas de Python, exceto que as chaves podem ser quaisquer objetos⁶, em vez de apenas cadeias. Então, existe uma progressão a partir das dispersões em Perl, nas quais as chaves devem ser cadeias, para as matrizes em PHP, nas quais as chaves podem ser inteiras ou cadeias; e para as dispersões em Ruby, nas quais qualquer objeto pode ser uma chave.

As matrizes em PHP são tanto normais quanto associativas, podendo ser tratadas como ambas. A linguagem fornece funções permitindo 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

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.

de quaisquer tipos. Uma tabela pode ser usada como uma matriz tradicional, como uma matriz associativa ou como um registro. Quando usada como uma matriz tradicional ou como uma matriz associativa, colchetes são colocados 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.

Uma matriz associativa é muito melhor do que uma matriz caso as buscas a elementos sejam necessárias, 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, por exemplo, com os nomes de empregados e seus salários. Por outro lado, se cada elemento de uma lista deve ser processado, é mais eficiente usar uma matriz.

6.6.2 Implementando 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 requerer isso. Um valor de dispersão de 32-bits é computado para cada entrada e armazenado com a entrada, apesar de uma matriz associativa inicialmente usar 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, mais bits do valor de dispersão são usados. Apenas metade das entradas precisa ser movida quando isso acontece. Assim, apesar de a expansão das matrizes associativas não ser gratuita, ela não é tão cara 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 pela qual eles 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, existe uma necessidade nos programas de modelar coleções de dados que não são do mesmo tipo ou tamanho. Por exemplo, informações sobre um estudante universitário podem incluir seu nome, seu número de estudante, 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 estudante, um ponto flutuante para a média de notas no histórico e assim por diante. Registros são projetados para esse tipo de necessidade.



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 na Pontifícia Universidade Católica do Rio de Janeiro, no Brasil. (para mais informações sobre Lua, visite [www.lua.org.](http://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 101 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 para 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 traduzímos tais programas manualmente para hexadecimal de forma a colocá-los em memória para executá-los.

Desde então, fui fisigado.

Existem poucas linguagens de programação bem-sucedidas projetadas em ambientes acadêmicos nos últimos 25 anos. Apesar de você ser um acadêmico, Lua foi projetada para aplicações bastante práticas. Você considera Lua 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 aceitaria-

mos as demandas dos usuários. Isso nos deu alguma latitude que outras linguagens não desfrutaram.

Mais recentemente, temos feito 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 Lua. 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 abordagem? Quisemos dizer que as partes mais importantes de 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 abordagem, 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 – isso 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 abordagem apresenta uma grande desvantagem também: você pode 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 em Lua. Por que você não adicionou um? Isso pode soar engraçado, mas o que realmente sentimos falta foi do valor “falso”; não tínhamos uso para um valor “verdadeiro”. Como no 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 um valor falso com 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 em que 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 grandes. Removemos diversos recursos de Lua, mas na maioria dos casos porque eles foram superados por um novo recurso, normalmente “melhor” em algum sentido. Isso aconteceu com métodos de etiquetagem – *tag methods* (substituídos pelos metatípicos), 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 quebraria a compatibilidade com as versões anteriores da linguagem é considerado, como é tomada essa decisão? Essas sempre são decisões 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 forma a 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. 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 pré-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 a 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). Também, 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, até mais úteis agora que as funções apresentam escopo léxico apropriado. Em meu livro sobre Lua, terminei 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 usar Lua para estender uma aplicação, para transformá-la Lua 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 quanto significativas são as tabelas de Lua na evolução das estruturas de dados nas linguagens de programação? Não penso que as tabelas de Lua tenham tido qualquer relevância na evolução de outras linguagens. Talvez isso mude no futuro, mas não estou seguro disso. Na minha opinião, a principal vantagem oferecida pelas tabelas de Lua é sua simplicidade, uma solução “tudo-em-um”. Mas essa simplicidade tem seus custos: por exemplo, a análise estática de programas Lua é muito difícil porque as tabelas são usadas de forma muito genérica e onipresente. Cada linguagem tem suas prioridades.

Pode parecer que registros e matrizes heterogêneas são a mesma coisa, mas esse não é o caso. Os elementos de matrizes heterogêneas são todos 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 sido parte de todas as linguagens de programação mais populares, exceto pelas versões anteriores ao Fortran 90, desde os anos 1960, quando foram introduzidos pelo COBOL. Em algumas linguagens que têm suporte para a programação orientada a objetos, os registros são simulados com objetos.

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 com classes, mas são um tanto diferentes. As estruturas em C# são tipos de valores alocados na pilha, de maneira oposta aos 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 estruturas de dados. Elas são discutidas com mais detalhes em relação a essa capacidade no Capítulo 11.

Em Python e Ruby, registros podem ser implementados como dispersões, as quais elas próprias podem ser elementos de matrizes.

As seguintes seções descrevem como os registros são declarados ou definidos, como referências a campos dentro dos registros são feitas, e discute as operações de registros comuns.

As questões de projeto específicas dos registros são:

- Qual é a forma sintática das referências a campos?
- Referências elípticas são permitidas?

6.7.1 Definição de registros

A diferença fundamental entre um registro e uma matriz é que elementos de registro, ou **campos**, não são referenciados por índices. Em vez disso, os campos são nomeados com identificadores, e referências para os campos são feitas usando esses identificadores. Outra diferença entre matrizes e registros é que os registros em algumas linguagens podem incluir uniões, as quais são discutidas na Seção 6.8.

O formato do COBOL de uma declaração de registro, que é parte da divisão de dados de um programa COBOL, é ilustrado no exemplo:

```
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, que indicam por seus valores relativos à estrutura hierárquica do registro. Qualquer linha seguida por uma 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.

Ada usa uma sintaxe diferente para registros; em vez de usar os números de níveis do COBOL, estruturas de registro são indicadas de uma maneira ortogonal ao simplesmente aninhar declarações de registros dentro de declarações de registro. Em Ada, registros não podem ser anônimos – devem ser tipos nomeados. Considere a declaração em Ada:

```
type Employee_Name_Type is record
    First : String (1..20);
    Middle : String (1..10);
    Last : String (1..20);
end record;
type Employee_Record_Type is record
    Employee_Name: Employee_Name_Type;
    Hourly_Rate: Float;
end record;
Employee_Record: Employee_Record_Type;
```

As declarações de registro do Fortran 95 requerem que quaisquer registros aninhados sejam previamente definidos como tipos. Então, para o registro de empregado de exemplo, o registro do nome do empregado precisaria ser definido primeiro, e então o registro de empregado simplesmente o nomearia como o tipo de seu primeiro campo.

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 os campos do registro.

Conforme mencionado, as matrizes associativas de Lua podem ser convenientemente usadas como registros. Por exemplo, considere a seguinte 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 envolve. As referências a campos em COBOL têm a forma

`nome_campo OF nome_registro_1 OF ... OF nome_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: usam o nome do maior registro que envolve os outros primeiro e o nome do campo por último. Por exemplo, a seguir temos uma referência ao campo Middle no exemplo anterior de registro em Ada:

Employee_Record.Employee_Name.Middle

C e C++ usam essa mesma sintaxe para referenciar os membros de suas estruturas. As referências a campos em Fortran 95 também têm essa forma, com sinais de percentual (%) em vez de pontos.

Referências a elementos em uma tabela Lua podem aparecer na sintaxe de referências a campos de registros, conforme visto nas sentenças de atribuição na 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. Tanto as referências de campos nos exemplos de COBOL quanto de Ada acima são completamente qualificadas. Como uma alternativa para as referências qualificadas, o 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 do empregado no registro COBOL declarado acima. Apesar de as referências elípticas serem uma conveniência para o programador, elas requerem que o compilador tenha estruturas de dados e procedimentos elaborados de forma a identificar corretamente campo referenciado. Elas também são, de certa forma, prejudiciais para a legibilidade.

6.7.3 Operações em registros

A atribuição é uma operação comum em registros. Na maioria dos casos, os tipos dos dois lados devem ser idênticos. Ada permite comparações entre registros para igualdade e diferença. Além disso, os registros em Ada podem ser inicializados com literais agregados.

COBOL fornece a sentença MOVE CORRESPONDING para mover registros. Ela copia um campo do registro de origem especificado para o registro de destino apenas se este tiver um campo com o mesmo nome. Essa é uma operação útil em aplicações de processamento de dados, nas quais os registros de entrada são movidos para atributos de saída após algumas modificações. Como os registros de entrada geralmente têm muitos campos com os mesmos nomes e propósitos, como campos em registros de saída, mas não necessariamente na mesma ordem, a operação MOVE CORRESPONDING pode salvar muitas sentenças. Por exemplo, considere as seguintes estruturas em COBOL:

```

01 INPUT-RECORD .
  02 NAME .
    05 LAST          PICTURE IS X(20) .
    05 MIDDLE        PICTURE IS X(15) .
    05 FIRST         PICTURE IS X(20) .
  02 EMPLOYEE-NUMBER PICTURE IS 9(10) .
  02 HOURS-WORKED   PICTURE IS 99 .

01 OUTPUT-RECORD .
  02 NAME .
    05 FIRST         PICTURE IS X(20) .
    05 MIDDLE        PICTURE IS X(15) .
    05 LAST          PICTURE IS X(20) .
  02 EMPLOYEE-NUMBER PICTURE IS 9(10) .
  02 GROSS-PAY      PICTURE IS 999V99 .
  02 NET-PAY        PICTURE IS 999V99 .

```

A sentença

```
MOVE CORRESPONDING INPUT-RECORD TO OUTPUT-RECORD .
```

copia os campos FIRST, MIDDLE, LAST, e EMPLOYEE-NUMBER do registro de entrada para o registro de saída.

6.7.4 Avaliação

Os registros são tipos de dados valiosos em linguagens de programação. O projeto de tipos registro é direto e seu uso é seguro. O único aspecto dos registros que não é totalmente legível são as referências elípticas permitidas pelo COBOL.

Registros e matrizes são fortemente relacionados com 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 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 particular. 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.

6.7.5 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 com cada campo. Acessos a campos são todos manipulados usando 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 UNIÕES

Uma **união** é um tipo cujas variáveis podem armazenar diferentes valores de tipos em vários momentos durante a execução de um programa. Como um 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 para cada entrada na tabela

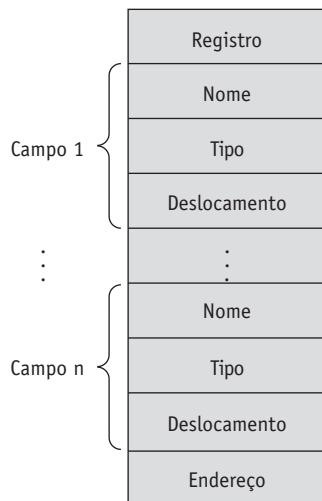


Figura 6.7 Um descritor em tempo de compilação para um registro.

é para o valor da constante. Suponha que para uma linguagem em particular 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 de tabela, pudesse armazenar um valor de qualquer um desses três tipos. Então, todos os valores constantes podem 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.

6.8.1 Questões de projeto

O problema da verificação de tipos para as uniões, discutido na Seção 6.10, leva a uma importante questão de projeto. Outra questão fundamental é como representar sintaticamente uma união. Em alguns projetos, as uniões estão confinadas a serem partes de estruturas do tipo registro, mas em outras elas não estão. Então, as questões de projeto primárias particulares aos tipos união são:

- A verificação de tipos deve ser obrigatória? Note que qualquer desses tipos de verificação deve ser dinâmico.
- As uniões devem ser embutidas em registros?

6.8.2 Uniões discriminadas *versus* uniões livres

Fortran, C e C++ fornecem construções para representar uniões nas quais não existe um suporte da linguagem para a verificação de tipos. Em Fortran, a sentença `Equivalence` é usada para especificar uniões; em C e C++, é a construção `union`. As uniões nessas linguagens são chamadas de **uniões livres**, porque é permitido que os programadores tenham total liberdade em relação à verificação de tipos sobre o uso dessas uniões. 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 seu 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 o ALGOL 68. Elas são agora suportadas por Ada.

6.8.3 Uniões em Ada

O projeto de Ada para uniões discriminadas, baseado no projeto de sua linguagem antecessora, o Pascal, permite ao usuário especificar variáveis de um tipo de registro variável que armazenará apenas um dos valores de tipo possíveis na variação. Dessa maneira, o usuário pode dizer ao sistema quando a verificação de tipos pode ser estática. Tal variável restrita é chamada de **variável variante restrita**.

A etiqueta de uma variável variante restrita é tratada como uma constante nomeada. Registros variantes sem restrições em Ada permitem que os valores de suas variantes troquem de tipo durante a execução. Entretanto, o tipo da variante pode ser modificado apenas pela atribuição do registro inteiro, incluindo o discriminante. Isso proíbe registros inconsistentes, visto que se o novo registro atribuído for um agregado de dados constante, o valor da etiqueta e o tipo da variante podem ser estaticamente verificados em relação à consistência⁷. Se o valor atribuído for uma variável, sua consistência foi garantida, e o novo valor dessa variável é, com certeza, consistente.

O exemplo a seguir mostra um registro variante em Ada:

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form : Shape) is
  record
    Filled : Boolean;
    Color : Colors;
    case Form is
      when Circle =>
        Diameter : Float;
      when Triangle =>
        Left_Side : Integer;
        Right_Side : Integer;
        Angle : Float;
      when Rectangle =>
        Side_1 : Integer;
        Side_2 : Integer;
    end case;
  end record;
```

A estrutura desse registro variante é mostrada na Figura 6.8. As duas sentenças a seguir declaram variáveis do tipo Figure:

```
Figure_1 : Figure;
Figure_2 : Figure (Form => Triangle);
```

⁷ A consistência aqui significa que se a etiqueta indicar que o tipo atual da união é Integer, o valor atual da união é de fato um Integer.

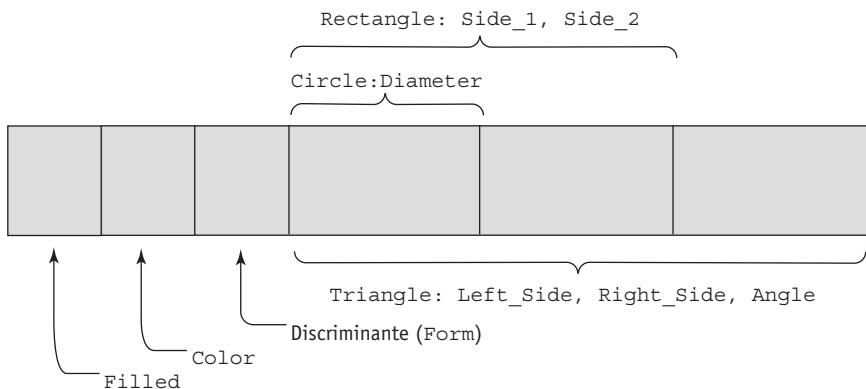


Figura 6.8 Uma união discriminada de três variáveis do tipo Shape (assuma que todas as variáveis são do mesmo tamanho).

Figure_1 é declarada como um registro variante sem restrições que não tem um valor inicial. Seu tipo pode ser modificado pela atribuição de um registro completo, incluindo o discriminante, conforme a seguir:

```
Figure_1 := (Filled => True,
              Color => Blue,
              Form => Rectangle,
              Side_1 => 12,
              Side_2 => 3);
```

O lado direito dessa atribuição é um agregado de dados.

A variável declarada Figure_2 é restrita a ser um triângulo e não pode ser modificada para outra variante.

Essa forma de união discriminada é segura, porque sempre permite a verificação de tipos, apesar de as referências aos campos em variantes sem restrição precisarem ser verificadas dinamicamente. Por exemplo, suponha que tivéssemos a seguinte sentença:

```
if (Figure_1.Diameter > 3.0) ...
```

O sistema em tempo de execução precisaria verificar Figure_1 para determinar se sua etiqueta Form é Circle. Se não for, seria um erro de tipo referenciar seu diâmetro (Diameter).

6.8.4 Avaliação

Uniões são construções potencialmente inseguras em algumas linguagens e uma das razões pelas quais Fortran, 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 outro lado, as uniões podem ser usadas seguramente, como em seu projeto na linguagem Ada. Em C e C++, as uniões devem ser usadas com cuidado.

Nem Java nem C# incluem uniões, o que pode ser um reflexo da crescente preocupação com a segurança em linguagens de programação.

6.8.5 Implementação de uniões

Uniões são implementadas simplesmente por meio do uso do mesmo endereço para cada uma das variantes possíveis. Um espaço de armazenamento suficiente para a maior variante é alocado. No caso das variantes restritas na linguagem Ada, o espaço exato de armazenamento pode ser usado porque não existe variação. A etiqueta de uma união discriminada é armazenada com a variante em uma estrutura similar a um registro.

Em tempo de compilação, a descrição completa de cada variante precisa ser armazenada, o que pode ser feito por meio da associação de uma tabela de escolha com a entrada da etiqueta no descritor. A tabela de escolha tem uma entrada para cada variante, a qual aponta a um descritor para a variante particular. Para ilustrar essa organização, considere o seguinte exemplo em Ada:

```
type Node (Tag : Boolean) is
  record
    case Tag is
      when True => Count : Integer;
      when False => Sum : Float;
    end case;
  end record;
```

O descritor para esse tipo poderia ter a forma mostrada na Figura 6.9.

6.9 PONTEIROS E REFERÊNCIAS

Um tipo **ponteiro** é um no qual as variáveis têm uma faixa de valores que consistem em endereços de memória e um valor especial, **nil**. O valor **nil**

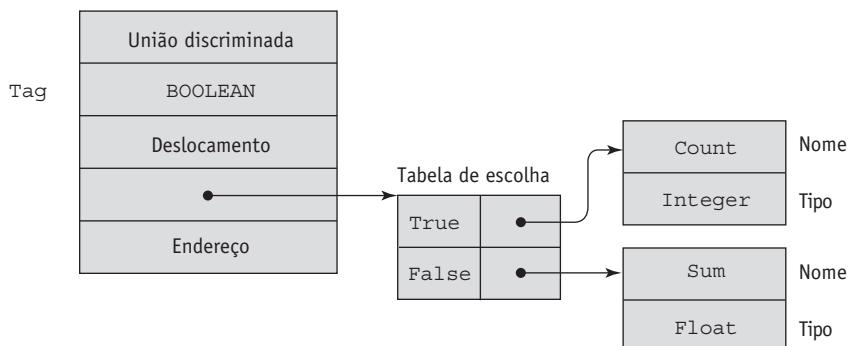


Figura 6.9 Um descritor em tempo de compilação para uma união discriminada.

não é um endereço válido e é usado para indicar que um ponteiro não pode ser usado atualmente para referenciar uma célula de memória.

Ponteiros são projetados para dois tipos de uso. Primeiro, os ponteiros fornecem alguns dos poderes 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 na área onde o armazenamento é dinamicamente alocado, chamado de **monte** (*heap*).

As variáveis dinamicamente alocadas a partir do monte são chamadas de **variáveis dinâmicas do monte**. Em geral, elas não têm identificadores associadas a elas e logo 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.

Ponteiros, diferentemente de matrizes e registros, não são tipos estruturados, apesar de serem definidos usando um operador de tipo (`*` em C e C++ e `access` em Ada). Além disso, são diferentes de variáveis escalares porque são mais usados para referenciar alguma outra variável, em vez de serem usados para armazenar dados de algum tipo. Essas duas categorias de variáveis são chamadas de tipos de referência e tipos de valor, respectivamente.

Ambas as formas de 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 como o Fortran 77, que não tem ponteiros. Isso requereria que o programador fornecesse e mantivesse uma coleção de nós de árvore disponíveis, os quais provavelmente seriam implementados em matrizes paralelas. Além disso, pela falta de armazenamento dinâmico em Fortran 77, seria necessário que o programador adivinhasse 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 6.9.7, são fortemente relacionadas aos ponteiros.

6.9.1 Questões de projeto

As questões de projeto primárias particulares a 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?
- Os ponteiros são restritos em relação ao tipo de valores aos quais eles 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?

6.9.2 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, o mecanismo de alocação, seja por operador ou por subprograma pré-definido, serve para inicializar a variável de ponteiro. Se os ponteiros são usados para endereçamento indireto às variáveis que não são dinâmicas do monte, então deve existir um operador explícito ou um subprograma pré-definido 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 a qual está vinculada, no caso de um ponteiro ser um endereço. Isso é exatamente como uma variável que não é um ponteiro em uma expressão seria interpretada, apesar de que, nesse caso, seu valor provavelmente não seria um endereço. Entretanto, o ponteiro também poderia ser interpretado como uma referência ao valor dentro da célula de memória apontado pela célula a qual a variável de ponteiro está vinculada. Nesse caso, o ponteiro é 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, que leva uma referência por meio de um nível de indireção, é a segunda operação fundamental dos ponteiros.

O desreferenciamento de ponteiros pode ser tanto explícito quanto implícito. No Fortran 95, é implícito, mas em muitas outras linguagens contemporâneas, ocorre apenas quando explicitamente especificado. Em C++, ele é explicitamente especificado com o asterisco (*) como um operador unário pré-fixado. 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
```

modifica `j` para o valor 206. Esse processo é mostrado na Figura 6.10.

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 para 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 nos referenciarmos a esse campo. O operador `->`, quando usado entre um ponteiro para um registro e um campo desse registro, combina o desreferenciamento e a referência ao campo. Por exemplo, a expressão `p->age` é equivalente a `(*p).age`. Em Ada, `p.age` pode ser usada, porque tais usos de ponteiros são implicitamente desreferenciados.

Linguagens que fornecem ponteiros para o gerenciamento de um monte devem incluir uma operação explícita de alocação. A alocação é al-

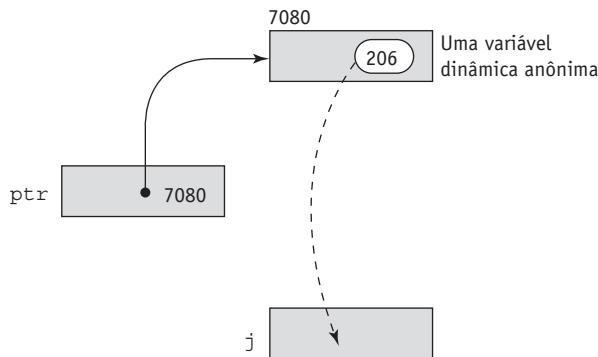


Figura 6.10 A operação de atribuição $j = *ptr$.

gumas vezes especificada com um subprograma, como `malloc` em C. Em linguagens que suportam programação orientada a aspectos, a alocação de objetos no monte é normalmente especificada com o operador `new`. C++, que não permite liberação implícita, usa `delete` como seu operador de liberação.

6.9.3 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 de 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. Os tipos de referência são discutidos na Seção 6.9.7.

6.9.3.1 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 sendo apontada pode ter sido realocada para alguma variável dinâmica do monte nova. Se a nova variável não for do mesmo tipo da antiga, as verificações de tipos dos usos do ponteiro solto são 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 é usado para

* N. de T.: Tais ponteiros também são conhecidos como ponteiros pendurados, ponteiros pendentes ou ponteiros selvagens. Do original em inglês *dangling pointers*.

modificar a variável dinâmica do monte, o valor da nova variável será destruído. Por fim, é possível que a posição agora esteja sendo temporariamente usada pelo sistema de gerenciamento de armazenamento, possivelmente como um ponteiro em uma cadeia de blocos de armazenamento disponíveis, permitindo que uma mudança na posição acarrete a 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 para null), mas p2 não é modificado pela operação. p2 é agora um ponteiro solto. Se a operação de liberação não modificar p1, tanto p1 quanto p2 seriam soltos.

Por exemplo, em C++ teríamos:

```
int * arrayPtr1;
int * arrayPtr2 = new int[100];
arrayPtr1 = arrayPtr2;
delete [] arrayPtr2;
// Agora, arrayPtr1 é solto, porque o armazenamento no monte
// para o qual ele estava apontando 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 tenha sido liberado.

Note que a liberação explícita de variáveis dinâmicas é a causa dos ponteiros soltos.

6.9.3.2 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.

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 prevenir os ponteiros soltos, também proibia o reuso de armazenamento no monte que o programa não precisava mais. Lembre-se de que a linguagem Pascal foi projetada para ser uma linguagem de ensino em vez de uma ferramenta industrial.

A primeira variável dinâmica do monte é agora inacessível, ou perdida. Isso às vezes é chamado de **vazamento de memória** e, independentemente de a linguagem usar liberação implícita ou explícita, é um problema. Nas seguintes seções, investigamos como os projetistas de linguagens lidaram com os problemas de ponteiros soltos e variáveis dinâmicas do monte perdidas.

6.9.4 Ponteiros em Ada

Os ponteiros de Ada são chamados de tipos de **acesso**. O problema dos ponteiros soltos é parcialmente tratado pelo projeto de Ada, ao menos na teoria. Uma variável dinâmica do monte pode ser (como uma opção do implementador) implicitamente liberada no fim do escopo de seu tipo de ponteiro, diminuindo drasticamente a necessidade de liberação explícita.

Entretanto, poucos compiladores Ada (se é que algum o fez) implementam essa forma de coleta de lixo, então a vantagem só existe em teoria. Como as variáveis dinâmicas do monte podem ser acessadas por variáveis de apenas um tipo, quando o final do escopo dessa declaração de tipo é alcançado, nenhum ponteiro pode ser deixado apontando para a variável dinâmica. Esse fato diminui o problema, porque a alocação explícita implementada de modo inadequado é a maior fonte de ponteiros soltos. Infelizmente, a linguagem Ada também tem um liberador explícito, `Unchecked_Deallocation`. O nome foi dado para desencorajar seu uso, ou ao menos avisar o usuário de seus potenciais problemas. `Unchecked_Deallocation` pode causar ponteiros soltos.

O problema de variáveis dinâmicas do monte perdidas não é eliminado pelo projeto de ponteiros de Ada.

6.9.5 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, mostrando 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.

Diferentemente dos de Ada, que podem apontar apenas para o monte, os ponteiros 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 & 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 para 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` para `count`. Note que a declaração de um ponteiro especifica seu tipo de domínio.

As duas sentenças de atribuição acima são equivalentes em seu efeito em `count` à única atribuição

```
count = init;
```

Aos ponteiros, podem ser atribuídos o valor de endereço de qualquer variável do tipo de domínio correto, ou a constante zero, usada para `nil`.

A aritmética de ponteiros é também possível de algumas formas restritas. Por exemplo, se `ptr` é uma variável de ponteiro declarada para apontar a alguma variável de algum tipo de dados, então

```
ptr + index
```

é uma expressão legal. A semântica de tal expressão é a seguinte. Em vez de simplesmente adicionar o valor de `index` a `ptr`, o valor de `index` 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 para 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 seguinte discussão é relacionada a matrizes multidimensionais apenas.

Em C e C++, todas as matrizes usam zero como o 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;
```

Considere a atribuição

```
ptr = list;
```

a qual atribui o endereço de `list[0]` a `ptr`, porque um nome de matriz sem um índice é interpretado como o endereço base de uma matriz. 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 *`, que podem apontar para valores de quaisquer tipos. Eles são, para todos os efeitos, ponteiros genéricos. Entretanto, a verificação de tipos não é um problema com ponteiros `void *`, porque essas linguagens não permitem desreferenciá-los. Um uso comum de ponteiros `void *` é 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 na memória para outro. Ela seria mais geral se pudesse ser passados ponteiros de qualquer tipo. Isso seria permitido se os parâmetros formais correspondentes na função fossem do tipo `void *`. A função então os converteria para o tipo `char *` e realizaria a operação, independentemente de quais tipos de ponteiros foram passados como parâmetros reais.

6.9.6 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. Como resultado, 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 previne a atribuição ao valor de endereço de uma variável de referência.

Variáveis de tipo de referência são especificadas em definições ao prece-
der 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 tí-
pos de referência em C++ fornecem uma comunicação de duas direções 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 comunica-
ção de duas direções, mas parâmetros formais como ponteiros requerem desre-
ferenciamento explícito, tornando o código menos legível e 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 Java, variáveis de referência são estendidas da forma de C++ para
uma que as permitem substituírem os ponteiros inteiramente. Em sua bus-
ca por uma segurança aumentada em relação ao C++, os projetistas de Java
removeram os ponteiros no estilo de C++ de uma vez só. Diferentemente
das variáveis de referência de C++, em Java elas podem ter atribuídas a elas
diferentes instâncias de classes; ou seja, não são constantes. Todas as instâncias
de classes em Java são referenciadas por variáveis de referência. Na verdade, 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 ob-
jeto) 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 implicitamente liberadas (não
existe um operador explícito de liberação), não podem existir referências sol-
tas em Java.

C# inclui tanto as referências de Java quanto os ponteiros de C++. Entretanto, o uso de ponteiros é fortemente desencorajado. Na verdade, quaisquer programas que usem ponteiros devem incluir o modificador `unsafe`. Note que, apesar dos objetos apontados por referências serem implicitamente liberados, isso não é verdade para objetos apontados por ponteiros. Os ponteiros foram incluídos em C# principalmente para permitir que os progra-
mas em C# interoperassem com código C e C++.

Todas as variáveis nas linguagens orientadas a objetos Smalltalk, Python, Ruby e Lua são referências. Elas são sempre implicitamente desreferenciadas. Além disso, os valores diretos dessas variáveis não podem ser acessados.

6.9.7 Avaliação

Os problemas de ponteiros soltos e lixo já foram discutidos em profundidade. Os de gerenciamento do monte são discutidos na Seção 6.9.9.3.

Ponteiros têm sido comparados à instrução `goto`, que aumenta a faixa de sentenças que podem ser executadas a seguir. 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 tinhá sido feita por Hoare (1973): “Sua introdução nas linguagens de alto nível têm sido 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 o poder completo dos ponteiros em C e C++ pela maior segurança das referências. A extensão do uso de ponteiros nos programas C# será uma medida disso.

6.9.8 Implementação de ponteiros e de tipos de referência

Na maioria das linguagens, os ponteiros são usados no gerenciamento do monte. O mesmo é verdadeiro para referências em Java e em C#, assim como para as variáveis em Smalltalk, Python e Ruby, então não podemos tratar ponteiros e referências separadamente. Primeiro, descrevemos brevemente como ponteiros e referências são representados internamente. Então, discutimos duas soluções possíveis para o problema dos ponteiros soltos. Por fim, descrevemos os principais problemas com as técnicas de gerenciamento do monte.

6.9.8.1 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 implementadas nesses sistemas como pares de células de 16-bits, um para cada uma das duas partes de um endereço.

6.9.8.2 Solução para o problema dos ponteiros soltos

Existem diversas soluções propostas para o problema dos ponteiros soltos. Dentre 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 `null`, indicando que a variável dinâmica do monte não existe mais. Essa abordagem previne 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.

Lápidas são caras 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 indireção, o que requer um ciclo adicional de máquina na maioria dos computadores. Aparentemente, nenhum dos projetistas das linguagens mais populares achou que a segurança adicional valia o custo, já que nenhuma linguagem amplamente usada emprega lápidas.

Uma alternativa às lápidas é a **abordagem fechaduras e chaves** (*locks-and-keys*) 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 legal; 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 ilegal. 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 mais casará com a fechadura, então o acesso não será permitido.

É claro, a melhor solução para o problema dos ponteiros soltos é tirar a responsabilidade pela liberação de variáveis dinâmicas do monte das mãos dos programadores. 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 fizeram isso. Tanto Java quanto C# usam essa abordagem para suas variáveis de referência. Lembre-se de que os ponteiros em C# não incluem liberação implícita.

6.9.8.3 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 um no qual segmentos de tamanho variável são alocados e liberados. Note que, para a liberação, discutiremos apenas as abordagens implícitas. Nossa discussão será breve e longe de ser completa, 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 quando toda a alocação e a liberação é de 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 pela primeira vez em larga escala. Todos os programas LISP e a maioria dos dados na linguagem consistem em células em listas encadeadas.

Em um monte de alocação de tamanho único, todas as células disponíveis estão encadeadas juntas usando os ponteiros nas células e formando uma lista de espaços disponíveis. A alocação é simplesmente uma questão de pegar o número de células necessárias dessa lista quando elas forem necessárias. A liberação é um processo muito mais complexo. Uma variável dinâmica do monte pode ser apontada por mais de um ponteiro, tornando difícil de determinar quando a variável não é mais útil para o programa. Simplesmente porque um ponteiro está desconectado de uma célula obviamente não a torna lixo; podem existir diversos outros ponteiros ainda apontando para a célula.

Em LISP, diversas das operações mais frequentes em programas criam coleções de células que não estão mais acessíveis ao programa e logo 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**, na qual 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 têm sido 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 estão atualmente apontando 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 chegar a zero, significa que nenhum ponteiro no programa está apontando para a célula, e então ela se tornou lixo e pode ser retornada para a lista de espaço disponível.

Existem três problemas com o 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 o 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 ser uma porção significativa do tempo de execução total de um programa. É claro, se as mudanças nos ponteiros não forem muito frequentes, isso não é 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 complicações 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 previne de ser coletada e colocada de volta para a 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 aquelas da aplicação, então ela nunca causa demoras significativas na execução da aplicação.

O processo original de coleta de lixo marcar e varrer opera 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 ele tenha alocado todas as células disponíveis. Nesse ponto, um processo de marcar e varrer é iniciado para recolher todo o lixo que foi deixado flutuando 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 marcadas como ainda sendo usadas são retornadas para a lista de espaço disponível.

Para ilustrar a cara dos algoritmos usados para marcar as células que estão atualmente sendo usadas, 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 de `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.

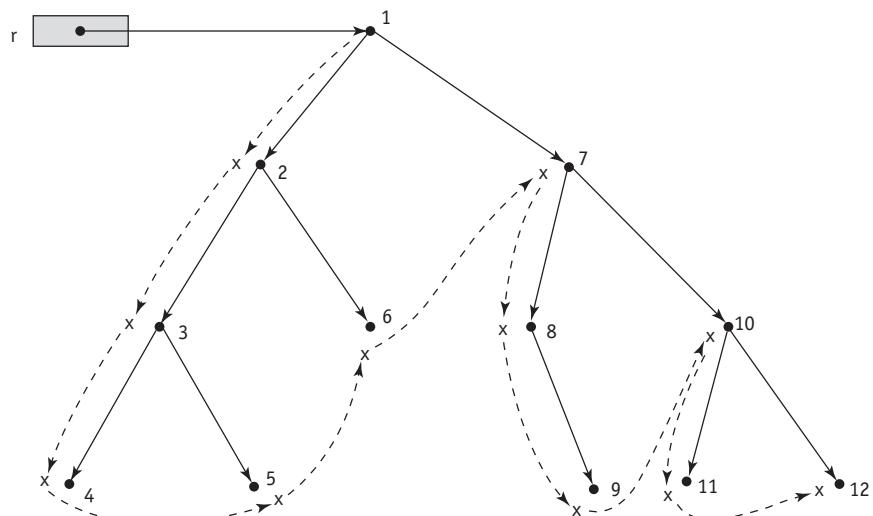
```

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 6.11 mostra um exemplo das ações desse procedimento em um dado grafo. 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



Linhos tracejadas mostram a ordem de marcação dos nós

Figura 6.11 Um exemplo das ações do algoritmo de marcação.

Schorr e Waite (1967). Seu método inverte os 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 com a versão original da abordagem marcar e varrer era ser realizada raramente – apenas quando um programa havia usado 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 sendo atualmente usada. Isso causa uma espera significativa no progresso 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 tem sido 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 efetivo 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 alternativa é realizar o processo marcar e varrer em partes, em vez de em toda a memória associada com a aplicação, em diferentes momentos. Isso fornece os mesmos tipos de melhorias que o marcar e varrer incremental.

Ambos os algoritmos de marcação para o método marcar e varrer e os processos necessários para o método de contagem de referências podem ser tornados mais eficientes pelo uso de rotação de ponteiros e operações de deslocamento descritas por Suzuki (1982).

Células de tamanho variável. Gerenciar um monte a partir do qual células de tamanho variável⁸ são alocadas tem todas as dificuldades de gerenciar um monte para células de tamanho único, mas também tem problemas adicionais. Infelizmente, células de tamanho variável são requeridas pela maioria das linguagens de programação. Os problemas adicionais advindos do gerenciamento de células de tamanho variável dependem do método usado. Se o marcar e varrer for usado, 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 requerer que cada célula tenha seu tamanho especificado como seu primeiro campo. Então, o percurso pode ser feito, apesar de requerer 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 uma corrente pode ser percorrida a partir de um ponteiro se não existe uma posição pré-definida para o ponteiro nas células apontadas? As células que não contêm ponteiros também são problemáticas. Adicionar um ponteiro interno

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.

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 rodar 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 para 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 de tamanhos variados, ou blocos, deixando 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 podem precisar ser unidos em blocos maiores. Alternativas ao uso do primeiro bloco grande o suficiente na lista podem diminuir a busca, mas podem requerer que a lista seja ordenada pelo tamanho do bloco. Em ambos os casos, manter a lista é uma sobrecarga adicional.

Se contadores de referências forem usados, 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, veja Wilson (2005).

6.10 VERIFICAÇÃO DE TIPOS

Para a discussão de verificação de tipos, o conceito de operandos e operadores é generalizado para incluir subprogramas e sentenças de atribuição. Subprogramas serão pensados como operadores cujos operandos são seus parâmetros. O símbolo de atribuição será considerado como um operador binário, com sua variável-alvo e sua expressão sendo os operandos.

A **verificação de tipos** é a atividade de garantir que os operandos de um operador são de tipos compatíveis. Um tipo **compatível** ou é legal para o operador, ou é permitido a ele, dentro das regras da linguagem, ser implicitamente convertido pelo código gerado pelo compilador (ou pelo interpretador) para um tipo legal. 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 inteira sofre uma coerção para `float` e uma adição de ponto flutuante é realizada.

Um **erro de tipo** é a aplicação de um operador a um operando de um tipo não apropriado. Por exemplo, na versão original do C, se um valor `int` fosse passado para uma função que esperava um valor `float`, um erro de tipo ocorria (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, por causa de 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 custosa. A penalidade para a verificação estática é uma flexibilidade reduzida para o programador. Menos atalhos e truques são permitidos. Tais técnicas, entretanto, são normalmente consideradas não apropriadas.

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 registros variantes em Ada, Equivalence em Fortran, e uniões de C e C++. Nesses casos, a verificação de tipos, se 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.

6.11 TIPAGEM FORTE

Uma das ideias em projeto de linguagem que se tornou proeminente na chamada revolução da programação estruturada da década de 1970 foi a **tipagem forte**, muito reconhecida como uma característica de linguagem altamente valiosa. Infelizmente, ela 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.

O Fortran 95 não é fortemente tipado porque o uso de Equivalence entre variáveis de tipos diferentes permite que uma variável de um tipo se refira a um valor de outro tipo, sem o sistema ser capaz de verificar o tipo do valor quando uma das variáveis equivalentes é referenciada ou tem valores atribuídos. Na verdade, a verificação de tipos em variáveis com o uso de Equivalence eliminaria grande parte de sua utilidade.

Ada é quase fortemente tipada, pois permite aos programadores usarem brechas nas regras de verificação de tipos ao requererem especificamente que a verificação de tipos seja suspensa para uma conversão de tipos em particular.

Essa suspensão temporária de verificação de tipos pode ser feita apenas quando uma instanciação da função genérica `Unchecked_Conversion` é chamada. Tais funções podem ser instanciadas para qualquer par de subtipos⁹. Alguém obtém um valor de seu tipo de parâmetro e retorna a cadeia de bits que é o valor atual do parâmetro. Nenhuma conversão real é realizada; é meramente uma forma de extrair o valor de uma variável de um tipo e usá-la como se fosse de um tipo diferente. Esse tipo de conversão é algumas vezes chamado de **conversão implícita sem conversão (nonconverting cast)**. Conversões não verificadas podem ser úteis para as operações de alocação e de liberação de armazenamento definidas pelo usuário, na qual endereços são manipulados como inteiros, mas devem ser usados como ponteiros. Como nenhuma verificação é feita em `Unchecked_Conversion`, é responsabilidade do programador garantir que o uso de um valor obtido a partir dela tenha algum significado.

C e C++ não são linguagens fortemente tipadas porque ambas incluem tipos união não 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.

Java e C#, apesar de serem baseadas em C++, são fortemente tipadas no mesmo sentido de Ada. 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 legal. O valor do operando inteiro sofre uma coerção para ponto-flutuante, e uma operação de ponto-flutuante é realizada. Isso é o que normalmente o programador 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 inteiros (`int`) `a` e `b` e a variável de ponto-flutuante (`float`) chamada `d`. Agora, se um programador queria digitar `a + b`, mas equivocadamente digitou `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 Fortran, C e C++, são menos confiáveis do que linguagens com poucas coerções, como Ada. Java e C# têm cerca de metade das coerções de tipo em atribuições que C++, então sua detecção de erros é melhor do que a de C++, mas ainda assim não está perto de ser tão efetiva quanto a de Ada. A questão da coerção é examinada em detalhes no Capítulo 7.

⁹ Normalmente, os dois subtipos devem ter o mesmo tamanho. Entretanto, uma implementação pode fornecer uma conversão não verificada (`Unchecked_Conversion`) para subtipos de tamanhos diferentes e fabricar regras para dizer como as diferenças são implementadas.

6.12 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 linguagem¹⁰. 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 torná-lo aceitável ao operador.

As regras de compatibilidade de tipos são simples e rígidas para os escalares pré-definidos. 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 se um operando de um em uma expressão 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 pré-definidas. Talvez o resultado mais importante de duas variáveis sendo de tipos equivalentes é que uma pode ter seu valor atribuído para a outra.

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.

¹⁰ 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.

Outro problema com a 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 com a 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 do 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 refere ao seu próprio tipo, como uma lista encadeada). Outras questões também podem aparecer. 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 um programa Fortran ou Ada são equivalentes se têm o mesmo tipo de elemento, mas faixas de índices de $0..10$ e $1..11$? Dois tipos de enumeração são equivalentes se têm o mesmo número de componentes, também literais com nomes diferentes?

Outra dificuldade com a 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 um 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 com a equivalência por nome. Um **tipo derivado** é um novo tipo baseado em algum previamente definido com o 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 ambos os tipos não são equivalentes aos de 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 nos tipos ancestrais, enquanto ainda assim herda todas as operações do ancestral.

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 `Integer`.

Note que os tipos derivados de Ada são muito diferentes dos de subfaixa de Ada. 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 legais 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, a equivalência de tipos por estrutura é usada. 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 da 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 do que a equivalência por nome definida no início desta seção. Se tivéssemos escrito em vez disso

```
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 do 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 é equivalente ou 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 é se duas estruturas, enumerações ou uniões são definidas em diferentes arquivos, o que faz com que a equivalência de tipos por estrutura seja usada. Essa é uma brecha na regra de equivalência de tipos por nomes que permite a equivalência de estruturas, enumerações e uniões que são definidas em diferentes arquivos.

C++ é como C, exceto que 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.

6.13 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 se tornou mais tarde uma ferramenta padrão na lógica. Qualquer discussão geral de 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 do escopo deste livro.

Em ciência da computação, existem dois ramos de teoria de tipos, um prático e um abstrato. O prático se preocupa com tipos de dados em linguagens de programação comerciais; o abstrato foca principalmente em cálculo lambda tipado, uma área de pesquisa intensa por parte dos cientistas da computação teóricos nos últimos 50 anos. Esta seção se restringe a uma breve discussão de 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 introduzido no Capítulo 3.

Um modelo alternativo às gramáticas de atributos usa um mapa de tipos e uma coleção de funções, não associadas com regras gramaticais, que especificam as regras de tipos. Um mapa de tipos é similar ao estado de um programa usado em semântica denotacional, consistindo em um conjunto de pares ordenados, com o primeiro elemento de cada par sendo um nome de variável e o segundo elemento sendo seu tipo. Um mapa de tipos é construído usando as 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 qualquer 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 ordinais 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 introduzidas 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 caracteres¹¹, para endereços dos elementos da matriz.

Um produto cartesiano de n conjuntos, S_1, S_2, \dots, S_n , é um conjunto denotado por $S_1 \times S_2 \times \dots \times S_n$. Cada elemento do conjunto do produto cartesiano tem um elemento de cada um dos conjuntos constituintes. Então, $S_1 \times S_2 = \{(x, y) \mid x \text{ está em } S_1 \text{ e } y \text{ está em } S_2\}$. 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 como um tipo de dados (veja a Seção 6.5). Os produtos cartesianos também modelam registros, ou estruturas, apesar de não exatamente. Produtos cartesianos não têm nomes de elementos, mas os registros os requerem. 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 6.8. Subconjuntos matemáticos são definidos por meio do fornecimento de uma regra que os elementos devem seguir.

¹¹ Em Ruby e Lua, as chaves das matrizes associativas não precisam ser cadeias de caracteres – elas podem ser de qualquer tipo.

Os conjuntos modelam os subtipos de Ada, apesar de não exatamente, dado que os subtipos devem consistir em elementos contíguos de 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 * em C, não são definidos em termos de uma operação de conjunto.

Isso conclui nossa discussão de formalismos em tipos de dados, assim como nossa discussão completa de tipos de dados.

RESUMO

Os tipos de dados de uma linguagem são uma grande parte do que determina seu estilo e sua utilidade. Com as estruturas de controle, eles formam o coração 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 por hardware.

Os tipos de enumeração e de subfaixa definidos pelo usuário são convenientes e melhoraram 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, uma implementação de um mapeamento. As matrizes podem ser estáticas, como as em C++ cuja definição inclui o especificador **static**; dinâmicas da pilha fixas, como em funções C (sem o especificador **static**); dinâmicas da pilha, como nos blocos Ada; dinâmicas do monte fixas, como em objetos Java; ou dinâmicas do monte, como nas matrizes em Perl. A maioria das linguagens permite poucas operações nas matrizes completas.

Os registros são agora incluídos na maioria das linguagens. Campos de registros são especificados de diversas maneiras. No caso do COBOL, podem ser referenciados sem nomear todos os registros que os envolvem, apesar disso ser confuso de implementar e de prejudicar a legibilidade. Em diversas linguagens que suportam a programação orientada a objetos, os registros são suportados com objetos.

Uniões são estruturas que podem armazenar valores de tipo 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 da linguagem Ada.

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 de ponteiros.

O nível de dificuldade para implementar um tipo de dados tem uma forte influência na inclusão ou não do tipo em uma linguagem. Tipos de enumeração, de subfaixas e de registro são relativamente fáceis de implementar. Matrizes também são diretas, apesar de o acesso a elementos das matrizes ser um processo caro quando ela tem diversos índices. A função de acesso requer uma adição e uma multiplicação adicional para cada índice.

Os ponteiros são relativamente fáceis de serem implementados, se o gerenciamento do monte não for considerado. Esse gerenciamento é relativamente fácil se todas as células são do mesmo tamanho, mas é complicado para a alocação e a liberação de células de tamanho variável.

A tipagem forte é o conceito de requerer 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 legais dentre 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 suas regras 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 dá uma das primeiras definições sistemáticas de tipos estruturados em Dahl et al. (1972). Uma discussão geral de uma ampla variedade de tipos de dados é feita em Cleaveland (1986).

A implementação de verificações em tempo de execução nas possíveis insecuranças 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 outros textos sobre linguagens de programação, como Pratt e Zelkowitz (2001) e Scott (2000). Uma discussão detalhada dos 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 do que os de C++?
8. Quais são as questões de projeto para matrizes?
9. Defina matrizes *estáticas*, *dinâmicas da pilha fixas*, *dinâmicas da pilha*, *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. Que recurso de inicialização de matrizes está disponível em Ada e não está disponível em outras linguagens imperativas comuns?
15. O que é uma constante agregada?

16. Que operações de matrizes são fornecidas especificamente para matrizes unidimensionais em Ada?
17. Quais são as diferenças entre as fatias do Fortran 95 e as de Ada?
18. Defina *ordem principal de linha* e *ordem principal de coluna*.
19. O que é uma função de acesso para uma matriz?
20. Quais são as entradas requeridas em um descritor de matriz em Java e quando elas devem ser armazenadas (em tempo de compilação ou em tempo de execução)?
21. Qual é o propósito dos números de níveis em registros no COBOL?
22. Defina *referências completamente qualificadas* e *elípticas* para campos em registros.
23. Defina *união*, *união livre* e *união discriminada*.
24. Quais são as questões de projeto para uniões?
25. As uniões de Ada são sempre verificadas em relação ao tipo?
26. Quais são as questões de projeto para os tipos ponteiro?
27. Quais são os dois problemas comuns com ponteiros?
28. Por que os ponteiros da maioria das linguagens são restritos de forma que apontem uma única variável de tipo?
29. O que é um tipo de referência em C++ e para que ele é comumente usado?
30. Por que as variáveis de referência em C++ são melhores do que os ponteiros para parâmetros formais?
31. Quais vantagens as variáveis de tipo de referência em Java e C# têm em relação aos ponteiros em outras linguagens?
32. Descreva a abordagem preguiçosa e a ansiosa para recuperar lixo.
33. Por que a aritmética de referências em Java e C# não faz sentido?
34. O que é um tipo compatível?
35. Defina erro de tipo.
36. Defina fortemente tipada.
37. Por que Java não é fortemente tipada?
38. O que é uma conversão implícita sem conversão?
39. Por que C e C++ não são fortemente tipadas?
40. O que é a equivalência de tipos por nome?
41. O que é a equivalência de tipos por estrutura?
42. Qual é a vantagem principal da equivalência de tipos por nome?
43. Qual é a desvantagem principal da equivalência de tipos por estrutura?
44. Para que tipos o C usa a equivalência de tipos por estrutura?
45. Que operações de conjunto modelam o tipo de dados **struct** de C?

CONJUNTO DE 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 que o 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, fechaduras e chaves para evitar ponteiros soltos, a partir 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? Por exemplo, considere a desreferência implícita de um ponteiro para um registro em Ada quando ele é usado para referenciar um campo de um registro.
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. As uniões em C e C++ são separadas dos registros dessas linguagens, em vez de combinadas como em Ada. Quais são as vantagens e desvantagens dessas duas escolhas?
10. Matrizes multidimensionais podem ser armazenadas em ordem principal de linha, como em C++, ou em ordem principal de coluna, como no Fortran. Desenvolva a função de acesso para ambas as disposições para matrizes tridimensionais.
11. Na linguagem Burroughs Extended ALGOL, as matrizes são armazenadas como uma matriz de uma dimensão de ponteiros para as linhas da matriz, as quais são tratadas como matrizes de uma dimensão de valores. Quais são as vantagens e desvantagens de tal esquema?
12. Analise e escreva uma comparação das funções `malloc` e `free` do C com os operadores `new` e `delete` de C++. Use a segurança como a consideração primária na comparação.
13. Analise e escreva uma comparação do uso de ponteiros C++ e de variáveis de referência em Java para referenciar variáveis dinâmicas do monte fixas. Use a segurança como a consideração primária na comparação.
14. Escreva uma pequena discussão sobre o que foi perdido e o que foi ganho na decisão dos projetistas de Java de não incluírem os ponteiros de C++.
15. 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 requerida em C++? Considere sistemas de tempo real.
16. Quais são os argumentos para a inclusão dos tipos de enumeração em C#, apesar de eles não estarem disponíveis nas primeiras versões de Java?
17. Qual é sua expectativa em relação ao nível de uso de ponteiros em C#? Com que frequência eles serão usados quando não são absolutamente necessários?
18. Crie duas listas de aplicações de matrizes, uma para aquelas que requerem matrizes irregulares e outra para aquelas que requerem matrizes retangulares. Agora, argumente se apenas matrizes irregulares, matrizes regulares ou ambas devem ser incluídas em uma linguagem de programação.
19. Compare as capacidades de manipulação de cadeias das bibliotecas de classes de C++, Java e C#.
20. Busque a definição de *fortemente tipada* conforme dada por Gehani (1983) e compare-a com a definição deste capítulo. De que forma elas são diferentes?
21. De que forma a verificação de tipos estática é melhor do que a verificação de tipos dinâmica?
22. 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 para o qual você tenha acesso. Escreva um relatório sobre suas descobertas.
2. Determine se algum compilador C para o qual você tenha acesso implementa a função `free`.
3. Escreva um programa que faça 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 de 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 coisas incorretas ou bobas são legais. Agora, escreva um programa em C# que faça as mesmas coisas e execute-o para determinar quantas dessas coisas incorretas ou bobas são legais. 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 usando apenas variáveis inteiros. 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 faça as mesmas operações, mas que use ponteiros e aritmética de ponteiros para a função de mapeamento de armazenamento para 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 legal em C++, mas não é em Java?
11. Para que tipos de A e B a sentença de atribuição simples `A = B` é legal em Java, mas não é em Ada?

Capítulo 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 com a ordem de avaliação dos operandos quando as funções têm efeitos colaterais. Operadores sobre carregados, tanto pré-definidos quanto definidos pelo usuário, são então discutidos, com seus efeitos nas expressões em programas. A seguir, expressões de modo misto são descritas e avaliadas, levando à definição e à avaliação de conversões de tipo de alargamento e de estreitamento, tanto implícitas quanto explícitas. Expressões relacionais e booleanas são discutidas, incluindo o processo de avaliação em curto-circuito. Por fim, são cobertas as sentenças de atribuição, desde sua forma mais simples até todas suas variações, incluindo atribuições como expressões e atribuições de modo misto.

O capítulo foca nas sentenças de expressão e de atribuição das linguagens imperativas. Questões de especificação e avaliação de expressões em linguagens funcionais e lógicas são discutidas nos Capítulos 15 e 16, respectivamente.

Expressões de casamento de padrões de cadeias de caracteres foram cobertas como parte do material sobre cadeias de caracteres no Capítulo 6, então não são mais 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 sendo usada. Um mecanismo formal (BNF) para descrever a sintaxe de expressões foi introduzido no Capítulo 3. Neste capítulo, as semânticas de expressões são discutidas – isso é, o que elas significam –, o que é governado pela maneira como elas são avaliadas.

Para entender a avaliação de expressões, é necessário estar familiarizado com as ordens de avaliação de operadores e operandos, ditadas 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 as diferenças de tipos, coerções e avaliação em curto-circuito.

A essência das linguagens de programação imperativas é o papel dominante das sentenças de atribuição. O propósito de uma sentença de atribuição é modificar o valor de uma variável. Então, uma parte integral de todas as linguagens imperativas é o conceito de variáveis cujos valores mudam durante a execução de um programa. Linguagens não imperativas algumas vezes

incluem variáveis de um tipo diferente, como os parâmetros de funções em linguagens funcionais.

7.2 EXPRESSÕES ARITMÉTICAS

A avaliação automática de expressões aritméticas similar 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**, dois operandos, ou **ternário**, três operandos. Na maioria das linguagens de programação imperativas, os operadores binários usam a **notação convencional (infix)**, ou seja, eles aparecem entre seus operandos. Uma exceção é Perl, possuindo alguns operadores **pré-fixados*** (*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 em linguagens imperativas.

A seguir, são listadas as principais questões de projeto para 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 de operadores de uma linguagem ditam 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:

$$a + b * c$$

* N. de T.: Também conhecida como notação polonesa.

Suponha que as variáveis `a`, `b` e `c` tenham os valores 3, 4 e 5, respectivamente. Se avaliada da esquerda para a direita (`a` adição primeiro e depois a multiplicação), o resultado é 35. Se 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 conceito de colocar os operadores em uma hierarquia de prioridades de avaliação e basear a avaliação da ordem das expressões parcialmente nessa hierarquia. Por exemplo, na matemática, a multiplicação tem prioridade maior do que a adição, talvez por seu nível mais alto de complexidade. Se essa convenção fosse aplicada na expressão de exemplo anterior, como seria o caso na maioria das linguagens de programação, a multiplicação seria feita primeiro.

As regras de precedência de operadores para avaliação de expressões definem 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 visto pelo projetista da linguagem. As regras de precedência de operadores das linguagens imperativas comuns são praticamente todas iguais, porque são baseadas naquelas da matemática. Nessas linguagens, a exponenciação tem a mais alta precedência (quando fornecida pela linguagem), seguida pela multiplicação e divisão no mesmo nível, depois pela adição e 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, falando sobre C++, chamou tal operador de um acidente histórico e acertou ao classificá-lo inútil (1990, p. 56). A subtração unária, é claro, normalmente modifica o sinal de seu operando. Em Java e C#, também causa a conversão implícita de operandos dos tipos `short` e `byte` para o tipo `int`.

Em todas as linguagens imperativas comuns, o operador unário de subtração pode aparecer em uma expressão ou no início dessa ou ainda em qualquer lugar dentro da expressão, desde que seja entre parênteses para prevenir que ele fique ao lado de outro operador. Por exemplo,

`a + (- b) * c`

é legal, mas

`a + - b * c`

normalmente não é.

A seguir, considere as expressões:

- `a / b`
- `a * b`
- `a ** b`

Nos primeiros dois casos, a precedência relativa do operador unário de subtração e o 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. Das 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 do que a subtração unária, então

- A $\star\star$ B

é equivalente a

- (A $\star\star$ B)

As precedências dos operadores aritméticos de algumas linguagens de programação bastante usadas são:

	<i>Ruby</i>	<i>Linguagens Baseadas em C</i>	<i>Ada</i>
<i>Mais alta</i>	$\star\star$	$\star\star$ e $\star\star$ pós-fixados	$\star\star$, abs
	$+ -$ unários	$\star\star$ e $\star\star$ pré-fixados, $+ -$ unários	$\star\star$, mod , rem
	$*, /, \%$	$*, /, \%$	$+ -$ unários
<i>Mais baixa</i>	$+ -$ binários	$+ -$ binários	$+ -$ binários

O operador $\star\star$ é a exponenciação. O $\%$ das linguagens baseadas em C e de Ruby é exatamente como o operador **rem** de Ada: recebe dois operandos inteiros e calcula o resto da divisão pelo segundo¹. O operador **mod** em Ada é idêntico ao **rem** quando ambos os operadores são positivos, mas pode ser diferente quando um ou ambos são negativos. Os operadores $\star\star$ e $\star\star$ das linguagens baseadas em C são descritos na Seção 7.7.4. O operador **abs** de Ada é um unário que calcula o valor absoluto de seu operando.

APL é estranha em relação às outras linguagens porque tem um só 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.

¹ Em versões de C antes do C99, o operador $\%$ era dependente da implementação em algumas situações, porque a divisão também era dependente dela.

Quando uma expressão contém duas ocorrências adjacentes² 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, ou seja, que a ocorrência mais à esquerda é avaliada primeiro ou a ocorrência mais à direita é avaliada primeiro, respectivamente.

A associatividade nas linguagens imperativas mais usadas é da esquerda para a direita, exceto que 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. Mas a exponenciação em Fortran e Ruby é associativa à direita, então na expressão

`A ** B ** C`

o operador direito é avaliado primeiro.

Em Ada, a exponenciação não é associativa. Logo, a expressão

`A ** B ** C`

é ilegal. Tal expressão deve ser entre parênteses para mostrar a ordem desejada, como em

`(A ** B) ** C`

ou

`A ** (B ** C)`

Em Visual Basic, o operador de exponenciação, `^`, é associativo à esquerda. As regras de associatividade para algumas linguagens imperativas bastante utilizadas são:

<i>Linguagem</i>	<i>Regra de Associatividade</i>
Ruby	Esquerda: <code>*</code> , <code>/</code> , <code>+</code> , <code>-</code> Direita: <code>**</code>
Linguagens Baseadas em C	Esquerda: <code>*</code> , <code>/</code> , <code>%</code> , <code>+ binário</code> , <code>- binário</code> Direita: <code>++</code> , <code>--</code> , <code>- unário</code> , <code>+ unário</code>
Ada	Esquerda: Todos, exceto <code>**</code> Não associativo: <code>**</code>

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.

² Dizemos que dois operadores são “adjacentes” se eles são separados por um único operando.

Por exemplo, na expressão

$$A \times B + C$$

o operador de adição é avaliado primeiro, seguido pelo operador de multiplicação (\cdot 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 imperativas 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 contendo 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 fossem também 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 ser capaz de produzir um código levemente 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 (por causa de limitações de tamanho). O fato de um operador matemático ser associativo não necessariamente implica 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 Ada precise avaliar a expressão

$$A + B + C + D$$

e que A e C são números positivos muito grandes, e B e D são números negativos com valores absolutos muito grandes. Nessa situação, adicionar B a A não causa uma exceção de transbordamento (*overflow*), mas adicionar C a A o faz. De maneira similar, adicionar C a B não causa transbordamento, mas adicionar D a B o faz. Por causa das limitações da aritmética computacional, a adição é catastroficamente não associativa nesse caso. Logo, se o compilador reordenar essas operações de adição, ele afeta o valor da expressão. Esse problema, é claro, pode ser evitado pelo programador, assumindo que os valores aproximados das variáveis são 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 têm 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 está disponível até que a adição da subexpressão que contém parênteses seja avaliada. A seguir, 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 que Ruby é uma linguagem orientada a objetos pura, ou seja, todos os valores de dados, incluindo literais, são objetos. Ruby oferece suporte para a coleção de operações aritméticas e lógicas incluída nas linguagens baseadas em C. O que separa Ruby das linguagens baseadas em C na área de expressões é que todos os operadores aritméticos, relacionais e de atribuição, assim como 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 um parâmetro.

Um resultado interessante da implementação de operadores como métodos é que eles podem ser sobreescritos por programas de aplicação e podem, também, ser redefinidos. Embora não seja sempre útil redefinir operadores para tipos pré-definidos, é útil, conforme veremos na Seção 7.3, definir operadores pré-definidos para tipos definidos pelo usuário, o que pode ser feito com a sobrecarga de operadores em algumas linguagens.

7.2.1.5 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 usando 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 é 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, usando 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) nos quais qualquer outra expressão possa ser usada. Além das linguagens baseadas em C, expressões condicionais são fornecidas em Perl, JavaScript e Ruby.

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 ser parte da instrução de linguagem de máquina e não requerer uma busca em memória. Se um operando é uma expressão entre parênteses, todos os operadores que ela contém devem ser avaliados antes de seu valor poder ser usado como um 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 um 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 é 10 e o valor da expressão é 20. Mas se o segundo operando for avaliado primeiro, o valor do primeiro operando é 20 e o valor da expressão é 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 a função for avaliada primeiro).

Note que funções na matemática não têm efeitos colaterais, porque não existe a noção de variáveis na matemática. O mesmo é verdade para linguagens de programação funcionais puras. 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 proibir que a avaliação afetasse o valor das expressões simplesmente pela proibição de efeitos colaterais funcionais. O segundo método de evitar o problema

é dizer na definição da linguagem que os operandos em expressões devem ser avaliados em uma ordem específica e exigir que os implementadores garantam essa ordem.

Proibir efeitos colaterais funcionais é difícil e elimina alguma flexibilidade para o programador. Considere o caso de C e C++, com apenas funções; isso significa que todos os subprogramas retornam um valor. Para eliminar os efeitos colaterais de parâmetros de duas direções 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 de aumentar a velocidade de execução. Em compiladores, por exemplo, 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 no programa 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 parâmetros³. 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 tem efeitos colaterais, `result1` e `result2` serão iguais, porque as expressões atribuídas a elas são equivalentes. Entretanto, suponha que `fun` tem o efeito colateral de adicionar 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.

³ Além disso, o valor da função não pode depender da ordem em que os seus parâmetros são avaliados.

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 transparentes referencialmente. Ser transparente referencialmente torna uma função equivalente a uma função matemática, em termos de facilidade de entendimento.

Como eles não têm variáveis, os programas escritos em linguagens funcionais puras são transparentes referencialmente. Funções em uma linguagem funcional pura não têm estado, que seriam armazenados em variáveis locais. Se tal função usa um valor de fora da função, esse valor deve ser uma constante, visto que não existem variáveis. Logo, o valor da função depende dos valores de seus parâmetros e possivelmente de uma ou de mais constantes globais.

A transparência referencial será discutida em mais detalhes no Capítulo 15, “Linguagens de Programação Funcionais”.

7.3 OPERADORES SOBRECARREGADOS

Operadores aritméticos são usados para mais de um propósito. Por exemplo, nas linguagens de programação imperativas, 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 bit a bit E (*AND*). Como um operador unário, entretanto, seu significado é totalmente diferente: 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 com esse 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 pode dizer se o operador é binário ou unário. Então, mais uma vez, uma falha em incluir o primeiro operando

quando o operador deveria ser binário passa despercebida como um erro pelo compilador. Entretanto, o significado das duas operações, unária e binária, são fortemente relacionados, não afetando a legibilidade de maneira adversa.

Algumas linguagens capazes de suportar tipos abstratos de dados (veja o Capítulo 11), como C++ e C#, permitem ao programador sobrepor 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 um subprograma chamado * responsável por essa nova operação. O compilador escolherá o significado correto quando um operador sobrepor é especificado, baseado nos tipos dos operandos, assim como ocorre com os operadores sobrepor definidos pela linguagem. Por exemplo, se essa nova definição para * é feita em um programa C#, um compilador C# usará a nova definição de * sempre que o operador aparecer com um inteiro simples como o operando da esquerda e uma matriz de inteiros como o operando da direita.

Quando usada com bom-senso, a sobrepor de operadores definida pelo usuário pode melhorar a legibilidade. Por exemplo, se + e * fossem sobrepor para um tipo de dados abstrato matriz e A, B, C e D são 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 sobrepor definida pelo usuário pode ser prejudicial à legibilidade. Por exemplo, nada previne que um usuário defina + como sendo 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 construir um sistema de software a partir de módulos criados por grupos diferentes. Se grupos diferentes sobreporaram o mesmo operador de maneiras diferentes, essas diferenças obviamente precisarão ser eliminadas antes de juntar as peças do sistema como um todo.

C++ tem alguns operadores que não podem ser sobreporados. Dentre eles, estão o operador de membro de classe ou de estrutura (.) e o operador de resolução de escopo (::). Interessantemente, a sobrepor de operadores foi um dos recursos de C++ não copiado em Java. Entretanto, ela reapareceu em C#.

A implementação de sobrepor 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 para um tipo que não pode armazenar aproximações equivalentes a todos os valores do tipo original. Por exemplo, converter um `double` para um `float` em Java é uma conversão de estreitamento, porque a faixa do tipo `double` é muito maior do que a de `float`. Uma **conversão de alargamento** converte um valor para um tipo que pode incluir ao menos aproximações de todos os valores do tipo original. Por exemplo, converter um `int` para um `float` em Java é uma conversão de alargamento. Conversões de alargamento são quase sempre seguras, mantendo a magnitude 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` é convertido para um inteiro em um programa Java, o resultado será apenas distamente relacionado com o 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 para ponto flutuante serem conversões 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 são também armazenados em 32 bits, com apenas cerca de sete dígitos de precisão (por causa do 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 tipos de retorno permitem a sobreescrita de um método de uma superclasse – apenas quando os tipos forem os mesmos, ou apenas em algumas situações. Essa questão, assim como o conceito de subclasses como subtipos, conforme discutido no Capítulo 12.

Conversões de tipo podem ser ou explícitas ou implícitas. As duas subseções seguintes discutem esses dois tipos 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 para operandos implícitos porque os computadores não têm operações binárias que recebam operandos de tipos di-

ferentes. Lembre-se de que, no Capítulo 5, a coerção foi definida como uma conversão de tipo implícita iniciada pelo compilador. Conversões de tipo explícitas solicitadas pelo programador são referidas como conversões explícitas (*casts*), não como coerções.

Apesar de alguns símbolos de operações poderem ser sobre carregados, 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 linguagem⁴. Para operadores sobre carregados 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 é legal na linguagem, o compilador deve escolher um deles para sofrer coerção e fornecer o código para essa coerção. Na discussão a seguir, as escolhas de projeto a respeito da coerção de diversas linguagens comuns são examinadas.

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 ser preocupar com essa categoria de erros ou se o compilador deve detectá-los.

Como uma simples ilustração do problema, considere o código Java:

```
int a;
float b, c, d;
...
d = b * a;
```

Assuma que o segundo operando de um operador de multiplicação deveria ser *c*, mas por um erro, foi digitado como *a*. Como expressões de modo misto são legais 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 legais em Java, esse erro de digitação seria detectado pelo compilador como um erro de tipo.

Como a detecção de erros é reduzida quando expressões de modo misto são permitidas, Ada permite poucos operandos de tipos mistos em expressões. Ela não permite misturar operandos inteiros e de ponto flutuante em uma expressão, com uma exceção: o operador de exponenciação, ******, pode receber ou

⁴ Essa premissa não é verdadeira para muitas linguagens. Um exemplo é dado posteriormente nesta seção.

um tipo de ponto flutuante ou um tipo inteiro como seu primeiro operando e um tipo inteiro como o segundo operando. Ada permite algumas outras misturas de tipos de operandos, normalmente relacionados aos tipos de subfaixa. Se o código de exemplo em Java fosse escrito em Ada, como em

```
A : Integer;
B, C, D : Float;
...
C := B * A;
```

o compilador Ada diria que a expressão é errônea, porque operandos dos tipos `Float` e `Integer` não podem ser misturados para o operador `*`.

Na maioria das outras linguagens comuns, não existem restrições em expressões aritméticas de modo misto.

As linguagens baseadas em C têm tipos inteiros menores do que `int`. Em Java, eles são `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 para um tipo maior. Por exemplo, considere o seguinte código Java:

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 combinada com um inteiro em uma expressão. 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, o outro operando sofre uma coerção para ponto flutuante, e a operação resultante é de ponto flutuante. Essa política de coerção é muito cara, 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.

```
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 para o tipo `byte` e colocada em `a`. Dado o maior tamanho das memórias dos computadores atuais, existe pouco incentivo em 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 para o valor do objeto que está sendo 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`.

Em Ada, os *casts* têm a sintaxe de chamadas a funções. Por exemplo,

`Float (Sum)`

Lembre-se (do Capítulo 5) de que Ada também tem uma função de conversão de tipos genérica, chamada `Unchecked_Conversion`, que não muda a representação do valor – ela meramente modifica seu tipo.

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, erros de tipo dos operandos não podem ocorrer. Os possíveis erros por causa de 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 o qual ele foi alocado. Isso é chamado de **transbordamento** (*overflow*) ou de **transbordamento negativo** (*underflow*), dependendo se o resultado era muito grande ou muito pequeno. Uma limitação da aritmética é que a divisão por zero não é permitida. É claro, o fato de que não é matematicamente permitido não previne um programa de tentar fazer 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

NOTA HISTÓRICA

Os projetistas do Fortran I usaram abreviações em inglês para os operadores relacionais porque os símbolos `>` e `<` não estavam nas perfuradoras de cartões na época do projeto do Fortran I (meio dos anos 50).

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 numéricicos, cadeias e tipos ordinais.

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 `!=`, Ada usa `/=`, Lua usa `~`, e Fortran 95 usa `.NE.` ou `<>`. JavaScript e PHP têm dois operadores relacionais adicionais, `==` e `!==`, similares aos seus operadores relativos, `==` e `!=`, mas que previnem seus operandos de sofrerem coerção. 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. Ruby usa `==` para o operador de igualdade relacional que usa coerções, e `eq?` 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 do 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 o 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 uma precedência igual. De acordo com isso, os operadores E e OU de Ada tem a mesma precedência. Entretanto, as linguagens baseadas em C atribuem uma precedência maior para o E do que para o OU. Talvez isso tenha resultado da correlação sem base 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 que 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

<i>Mais alta</i>	++ e --pós-fixados + e - unários, ++ e -- pré-fixados, ! *, /, % + e - binários <, >, <=, >= =, != &&
<i>Mais baixa</i>	

Versões do C anteriores ao C99 são estranhas dentre as linguagens imperativas mais populares, visto que elas não têm um tipo booleano e, dessa forma, não têm valores booleanos. Assim, valores numéricos são usados para representar valores booleanos. Em vez de operandos booleanos, variáveis escalares (numéricas ou caracteres) e constantes são usadas, com o zero sendo 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 estranho do projeto de C para expressões relacionais é que a seguinte expressão é legal:

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 dos 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 do 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 riqueza das coleções de operadores e da complexidade de expressões possíveis nessas linguagens.

A legibilidade dita 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 legal para um operador booleano. Nas outras linguagens imperativas, qualquer expressão não booleana usada como um 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 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, como $0 * x = 0$ para qualquer x . Então, quando a é 0, não há a necessidade de avaliar $(b / 13 - 1)$ ou de realizar a segunda multiplicação. Entretanto, em expressões aritméticas, esse atalho não é facilmente detectado durante a execução, então ele nunca é tomado.

O valor da expressão booleana

```
(a >= 0) && (b < 10)
```

é independente da segunda expressão relacional se $a < 0$, porque a expressão $(\text{FALSE} \&\& (b < 10))$ é falsa para todos os valores de b . Então, quando $a < 0$, 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 com a 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 usando a sentença **while**. Uma versão simples de código Java para tal busca, assumindo 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** são 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 o 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 parte da expressão que contém um efeito colateral não é 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-círcuito 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* \leq *b*. Se o programador assumiu que *b* seria modificado toda vez em que essa expressão fosse avaliada durante a execução (e a corretude do programa dependesse disso), o programa falharia.

Ada permite que o programador especifique avaliação em curto-círcuito dos operadores booleanos E e OU por meio dos operadores de duas palavras **and then** e **or else**. Por exemplo, mais uma vez assumindo que *List* é declarada como tendo uma faixa de *1..Listlen*, o código Ada

```
Index := 1;
while (Index <= Listlen) and then (List (Index) /= Key)
  loop
  Index := Index + 1;
end loop;
```

não causaria um erro quando *Key* não estivesse em *List* e *Index* se tornasse maior que *Listlen*.

Nas linguagens baseadas em C, os operadores E e OU usuais, **&&** e **||**, respectivamente, são avaliados em curto-círcuito. 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-círcuito. É claro, os operadores bit a bit apenas são equivalentes aos operadores booleanos usuais se todos os operandos forem restritos a serem ou 0 (para falso) ou 1 (para verdadeiro).

Todos os operadores lógicos de Ruby, Perl e Python são avaliados em curto-círcuito.

A inclusão tanto de operadores de curto-círcuito quanto operadores comuns em Ada é claramente o melhor projeto, porque ele fornece ao programador a flexibilidade de escolher avaliação em curto-círcuito para qualquer expressão booleana para a qual ela for apropriada.

7.7 SENTENÇAS DE ATRIBUIÇÃO

Como falamos previamente, a sentença de atribuição é uma das construções centrais em linguagens imperativas. Ela fornece o mecanismo pelo qual o usuário pode mudar dinamicamente as vinculações de valores a variáveis. Na seção seguinte, a forma mais simples de atribuição é discutida. As seções subsequentes descrevem uma variedade de alternativas.

7.7.1 Atribuições simples

Praticamente todas as linguagens de programação usadas atualmente usam 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.

O ALGOL 60 foi pioneiro no uso de `:=` como o operador de atribuição, que evita a confusão da atribuição com a igualdade. Ada usa esse operador de atribuição.

As escolhas de projeto de 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 autocontida, e o destino é restringido como 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 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 pelo ALGOL 68, que foram posteriormente adotados de uma forma levemente diferente pelo C, e são 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 autocontidas de um único operador. Eles podem aparecer como operadores pré-fixados, que precedem os operandos, ou como operadores pós-fixados, que seguem 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 descrita 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` ocorre primeiro; então `count` é 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 é certamente uma. 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` é primeiro incrementado e então negado. Então, ele é equivalente a

```
- (count ++)
```

em vez de

```
(- count) ++
```

7.7.5 Atribuição como uma expressão

Nas linguagens baseadas em C, Perl e JavaScript, a sentença de atribuição produz um resultado, que 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 que ele tem 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 `{ ... }` é 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 do que dos operadores relacionais. Sem os parênteses, o novo caractere seria comparado com `EOF` primeiro. 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 fornece ainda outro tipo de efeito colateral em expressões. Esse tipo de efeito colateral pode levar a expressões que são difíceis de serem lidas e entendidas. Uma expressão com qualquer tipo de efeito colateral tem essa desvantagem. Tal expressão 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 estranha de execução. 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 é também legal em Python.

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 eles são 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. Essa suposição seria errada, entretanto, porque os operadores de C foram herdados da linguagem B, projetada antes do primeiro PDP-11.

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, ele não é detectado como um erro pelo compilador. Em vez de testar uma expressão relacional, o valor que é atribuído a *x* é testado (nesse caso, é o valor de *y* que chega nessa sentença). Isso é um resultado de duas decisões de projeto: permitir que atribuições se comportem como um operador binário comum e usar dois operadores bastante similares, = 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**, não permitindo que esse problema ocorra*.

7.7.6 Atribuições de listas

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);
```

A semântica é 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);
```

Isso 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 de listas em Ruby é similar a de Perl, exceto pelo fato de que o lado esquerdo e o direito não ficam entre parênteses. Além disso, Ruby inclui algumas outras versões mais elaboradas de atribuições de listas, que não são discutidas aqui.

* N. de R. T.: Na verdade, esse problema pode ocorrer em Java caso as variáveis *x* e *y* do exemplo fossem do tipo primitivo `boolean` ou do tipo `wrapper Boolean`.

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 está recebendo a atribuição, ou a coerção pode ser usada em alguns casos em que os tipos não casam? Fortran, 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 legais, com coerções livremente aplicadas⁵. Ada não permite atribuições de modo misto.

Num claro distanciamento de C++, Java e C# permitem a atribuição de modo misto apenas se a coerção requerida é de alargamento⁶. Então, um valor `int` pode ser atribuído a 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 a C e C++.

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 de 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 Note que, em Python e Ruby, os tipos são associados com objetos, não com variáveis, então não existe algo como atribuições de modo misto nessas linguagens.

⁶ Não é exatamente verdade: se um literal inteiro, para o qual o compilador atribui por padrão o tipo `int`, 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, que não pode resultar em um erro.

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 sobre carregado?
12. Defina *conversões de alargamento e de estreitamento*.
13. Qual é a diferença entre == e === em JavaScript?
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. Diga uma linguagem que sempre faz avaliação em curto-circuito de expressões booleanas. Diga uma que nunca faz isso. Diga uma na qual o programador pode escolher.
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 de listas?
25. Que atribuições de modo misto são permitidas em Ada?
26. Que atribuições de modo misto são permitidas em Java?
27. O que é um *cast*?

CONJUNTO DE PROBLEMAS

1. Quando você poderia querer 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 sobre carregados em sua linguagem favorita seria benéfica? Por quê?
4. Seria uma boa ideia eliminar todas as regras de precedência de operadores e requerer parênteses para mostrar a precedência desejada em expressões? Por quê?
5. As operações de atribuição de C (por exemplo, +=) deveriam ser incluídas em outras linguagens (que ainda não as têm)? Por quê?
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)? Por quê?
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. Assuma as seguintes regras de associatividade e de precedência para expressões:

<i>Precedência</i>	<i>Mais alta</i>	<i>*, /, not</i>
		<i>+, -, &, mod</i>
		<i>- (unário)</i>
		<i>=, /=, <, <=, >, >=, ></i>
		<i>and</i>
	<i>Mais baixa</i>	<i>or, xor</i>
<i>Associatividade</i>	<i>Esquerda para a direita</i>	

Mostre a ordem de avaliação das seguintes expressões por meio do uso de parênteses em todas as subexpressões e colocando 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, assumindo que não existem regras de precedência e 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. Assuma que os únicos operandos são os nomes a, b, c, d, e.
12. Usando a gramática do Problema 11, desenhe árvores de análise sintática para as expressões do Problema 9.
13. Considere que a função fun é 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`
- se os operandos na expressão forem avaliados da esquerda para a direita?
 - se os operandos na expressão forem avaliados da direita para a esquerda?
- Qual seu principal argumento contra as (ou a favor das) regras de precedência de operadores de APL?
 - Explique por que é difícil eliminar efeitos colaterais funcionais em C.
 - Para alguma linguagem de sua escolha, crie uma lista de símbolos de operadores que poderiam ser usados para eliminar todas as sobrecargas de operadores.
 - 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.
 - Deveria ser permitido a um compilador com otimização para C e C++ modificar a ordem das subexpressões em uma expressão booleana? Por quê?
 - Responda a questão do Problema 17 para Ada.
 - 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`, assumindo que

- os operandos são avaliados da esquerda para a direita.
 - os operandos são avaliados da direita para a esquerda.
- Por que Java especifica que operandos em expressões são todos avaliados na ordem esquerda para a direita?
 - Explique como as regras de coerção de uma linguagem afetam sua detecção de erros.

EXERCÍCIOS DE PROGRAMAÇÃO

- Rode 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.
- Reescreva o programa do Exercício 1 em C++, Java e C#, execute-os e compare os resultados.
- Escreva um programa de teste em sua linguagem favorita que determine e mostre a precedência e a associatividade de seus operadores aritméticos e booleanos.
- Escreva um programa em Ada que ilustre a diferença entre `mod` e `rem` usando uma variedade de operandos inteiros positivos e negativos.
- Escreva um programa em Java que exponha a regra de Java para a ordem de avaliação de operandos quando um deles é uma chamada a método.
- Repita o Exercício 5 com C++.

7. Repita o Exercício 6 com C#.
8. Escreva um programa em C++, Java ou C# que ilustre a ordem de avaliação de expressões usadas como parâmetros reais de um método.
9. 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);
```

10. Escreva um programa em Java, C++ ou C# que realize um grande número de operações de ponto flutuante e um número igual de operações sobre inteiros e compare o tempo necessário.

Capítulo 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 de 9 a 13. Entre esses dois extremos, está a importante questão do fluxo de controle entre sentenças, assunto deste capítulo.

Começamos com uma visão geral da evolução das sentenças de controle nas linguagens de programação imperativas. Esse tópico é seguido de um exame cuidadoso das construções de seleção, tanto para as de dois caminhos quanto para as de seleção múltipla. Depois, discutimos a variedade de construções para repetição que vêm sendo desenvolvidas e usadas em linguagens de programação. A seguir, verificamos os problemas associados com sentenças de desvios incondicionais. Por fim, descrevemos as construções 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 poderosas: alguma forma de selecionar entre caminhos alternativos de fluxo de controle (de execução de sentenças) e uma de causar a execução repetida de sentenças ou de sequências de sentenças. Sentenças que fornecem tais tipos de capacidades são chamadas de **sentenças de controle**.

As sentenças de controle da primeira linguagem de programação bem-sucedida, o Fortran, eram, 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 do que do projeto de linguagem. Na época, pouco se sabia acerca da dificuldade da programação, e se pensava que as sentenças de controle do Fortran em meados dos anos 1950 eram completamente aceitáveis. Pelos padrões de hoje, entretanto, elas seriam consideradas inadequadas.

Uma boa parte da pesquisa e da discussão foi devotada às sentenças de controle nos 10 anos entre meados dos anos 1960 e meados dos anos 1970. Uma das conclusões primárias desses esforços era de 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 dentre 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 – po-

tencialmente útil, mas não essencial. Esse fato, combinado com os problemas práticos do uso de desvios incondicionais, ou *gosos*, levou a um grande debate acerca do *goto*, conforme discutido na Seção 8.4.1.

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 da legibilidade. Todas as linguagens que se tornaram bastante usadas incluem mais sentenças de controle do que as duas minimamente requeridas, porque a facilidade de escrita é melhorada com um número maior e uma variedade mais ampla de sentenças. Por exemplo, em vez de obrigar 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 usado 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 demanda aos leitores dos programas conhecerem uma linguagem maior. Lembre que poucas pessoas aprendem todas as construções de uma linguagem relativamente grande; em vez disso, elas aprendem o subconjunto que escolhem usar, em geral um diferente daquele usado pelo programador responsável pelo programa que essas pessoas estão tentando ler. Por outro lado, poucas sentenças de controle podem requerer o uso de sentenças de mais baixo nível, como o *goto*, que também torna 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 às custas de sua simplicidade, tamanho e 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 adicionam pouco à flexibilidade de uma estrutura de controle, em relação ao decréscimo na legibilidade causada pela complexidade aumentada. Note que múltiplas entradas são possíveis apenas em linguagens que incluem *gosos* 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. A razão é que todas as linguagens de programação permitem alguma forma de múltiplas saídas de estruturas de controle, pela seguinte razão: se todas as saídas de uma estrutura de controle são restritas a transferir o controle para a primeira sentença seguinte à 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 caem em duas categorias gerais: dois caminhos e n caminhos, ou seleção múltipla. Seletores de dois caminhos são discutidos na Seção 8.2.1; construções 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 em:

- 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?

NOTA HISTÓRICA

O Fortran inclui um seletor de três caminhos chamado de “se aritmético” que usa uma expressão aritmética de controle. Ele faz com que o controle vá para uma das três sentenças rotuladas, dependendo se o valor de sua expressão de controle for negativo, zero ou maior que zero. Essa sentença está na lista de recursos em obsolescência do Fortran 95 e 2003.

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 para parênteses, então eles são normalmente omitidos, como em Ruby.

No C89, que não tinha um tipo de dados booleano, as 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, como Ada, Java, Ruby e C#, apenas expressões booleanas podem ser usadas para expressões de controle.

8.2.1.3 Forma da cláusula

Em muitas linguagens contemporâneas, as cláusulas então e senão aparecem ou como sentenças simples ou como sentenças compostas. Uma variação disso é Perl, na qual todas as cláusulas então e senão devem ser sentenças compostas, mesmo se contiverem sentenças únicas. As linguagens baseadas em C, como Perl, JavaScript e PHP, usam chaves para formar sentenças compostas, que servem como os corpos das cláusulas então e senão. Em Fortran 95, Ada, Python e Ruby, as cláusulas então e senão são sequências de sentenças. A construção de seleção completa é terminada em Fortran 95, Ada e Ruby com uma palavra reservada¹.

Python usa indentação para especificar sentenças compostas. Por exemplo,

```
if x > y :  
    x = y  
    print "case 1"
```

Todas as sentenças indentadas igualmente são incluídas na sentença composta². Note que em vez do `then`, um sinal de dois-pontos é usado 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 Aninhando seletores

Lembre que o problema da ambiguidade sintática de uma gramática direta para uma construção de seleção de dois caminhos foi discutido no Capítulo 3. Tal gramática era:

```
<if_stmt> → if <logic_expr> then <stmt>  
          | if <logic_expr> then <stmt> else <stmt>
```

¹ Na verdade, em Ada e em Fortran, são duas palavras reservadas, `end if` (Ada) ou `End If` (Fortran).

² A sentença seguinte à sentença composta deve ter a mesma indentação que o `if`.

NOTA HISTÓRICA

Os projetistas do ALGOL 60 escolheram usar sintaxe, em vez de uma regra, para conectar cláusulas *senão* com cláusulas *então*. Especificamente, não é permitido que uma sentença *if* seja diretamente aninhada em uma cláusula *então*. Se um *if* precisar ser aninhado em uma cláusula *então*, ele deve ser colocado em uma sentença composta, como no exemplo anterior em Java.

A diferença entre os dois projetos é que a versão em Java permite a alguém escrever o seletor aninhado que aparentemente está casando com a cláusula *senão* com a primeira cláusula *então*, mas que na verdade não casa, enquanto essa mesma forma é sintaticamente ilegal no ALGOL 60, não permitindo esse problema sutil que acontece em Java ocorrer no ALGOL 60.

A questão é que, quando uma construção de seleção é aninhada na cláusula então de uma construção de seleção, não é 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 construção pode ser interpretada de duas maneiras, dependendo se a cláusula senão casa com a primeira cláusula então ou com a segunda. Note que a indentação parece indicar que a cláusula senão casa com a primeira cláusula então. Entretanto, com a exceção de Python, a indentação 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 um casamento 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. Então, no exemplo, a cláusula senão seria a alternativa à segunda cláusula então. A desvantagem de usar uma regra em vez de alguma entidade sintática é que, apesar de o programador poder querer dizer que a cláusula senão seja a alternativa à primeira cláusula então e compilador achar que a estrutura está sintaticamente 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, a linguagem não sofre desse problema. Em Perl, o código anterior seria escrito como

```
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 formar 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, marca o final da cláusula então. Quando a cláusula então é uma sentença composta, ela é terminada com um fecha chaves. 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 construção 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 construção. Esse é o projeto escolhido para a construção de seleção em Fortran 95, Ada, Ruby e Lua. Por exemplo, considere a seguinte construção em Ruby, cuja forma é idêntica à de Lua:

```
if a > b then
    sum = sum + a
    acount = acount + 1
else
    sum = sum + b
    bcount = bcount + 1
end
```

O projeto dessa construção é mais regular do que aquele das construções de seleção das linguagens baseadas em C, porque a forma é a mesma independen-

temente do número de sentenças nas cláusulas então e senão. (Isso também é verdade para Perl). Lembre que em Ruby (e Lua), as cláusulas então e senão consistem em sequências de sentenças em vez de sentenças compostas. A primeira interpretação do exemplo de seleção no início da Seção 8.2.1.4, 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 construção de seleção da Seção 8.2.1.4, na qual a cláusula senão é casada com o `if` mais externo, pode ser escrita em Ruby como:

```
if sum == 0 then
  if count == 0 then
    result = 0
  end
else
  result = 1
end
```

A seguinte construção, escrita em Python, é semanticamente equivalente à última construção em Ruby acima:

```
if sum == 0 :
  if count == 0 :
    result = 0
  else:
    result = 1
```

Se a linha `else` fosse indentada para começar na mesma coluna que o `if` aninhado, a cláusula senão casaria com o `if` interno.

8.2.2 Construções de seleção múltipla

A construção de **seleção múltipla** permite a seleção de uma dentre 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 dentre mais de dois caminhos de controle em um programa é comum. Apesar de um seletor múltiplo poder ser construído

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 para 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 a do tipo de expressão na qual o seletor é baseado. Nesse caso, a faixa de possibilidades é grande, em parte porque o número de seleções possíveis é grande. 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. A seguir, existe a questão de se apenas um segmento selecionável pode ser executado quando a construção é executada. Essa não é uma questão para seletores de dois caminhos porque eles permitem apenas que uma das cláusulas esteja em um caminho de controle durante uma execução. Conforme veremos, a resolução dessa questão é um troca de confiabilidade por flexibilidade. Outra questão é a 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 dentre os segmentos selecionáveis). A escolha aqui é entre não permitir que essa situação aconteça e fazer com que a construção não faça nada quando tal situação ocorre.

A seguir, temos um resumo dessas questões de projeto:

- Qual é 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

O construtor de seleção múltipla da linguagem C, o **switch**, que também é parte de C++, Java e JavaScript, é um projeto relativamente primitivo. Sua forma geral é

```
switch (expressão) {  
    case expressão_constante_1 : sentença_1;  
    ...  
    case constante_n: sentença_n;  
    [default: sentença_n+1]  
}
```

onde a expressão de controle e as expressões constantes são de algum tipo discreto. Isso inclui tipos inteiros, como 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, a construção não faz nada.

A construção `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 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 em cada execução. Dessa forma, o código para as constantes 2 e 4 é executado a cada vez que o código nas constantes 1 e 3 é executado. Para separar logicamente esses segmentos, um desvio explícito deve ser incluído. A sentença `break`, um goto restrito, é normalmente usada para sair de construções `switch`.

A seguinte construção `switch` usa `break` para restringir cada execução para 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 a partir de um segmento de código para outro. No exemplo acima, os segmentos para os valores de caso 1 e 2 estão vazios, permitindo que o controle flua para os segmentos para 3 e 4, respectivamente. Essa é a razão pela qual não existem desvios implícitos na construção `switch`. O problema de confiabilidade com esse projeto surge quando a ausência errônea de uma sentença `break` em um segmento que permita o controle flua incorretamente

para o próximo segmento. Os projetistas do **switch** de C trocaram um decréscimo na confiabilidade por um aumento de flexibilidade. Estudos têm mostrado, entretanto, que a habilidade de fazer com que o controle flua de um segmento selecionável para outro é raramente usada. O **switch** de C é modelado de acordo com a sentença de seleção múltipla presente no 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 se fossem rótulos de sentenças normais. Essa permissividade pode resultar em estruturas altamente complexas dentro do corpo da sentença **switch**. O seguinte exemplo é 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);
```

Esse código pode parecer ter uma forma diabolicamente complexa, mas foi projetado para um problema real e funciona correta e eficientemente para resolver esse problema³.

O **switch** de Java previne 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 daquelas 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 construção **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++;
        break;
    case 0:
        Zeros++;
```

³ O problema é chamar `process_prime` quando `x` é um primo e `process_composite` quando `x` não é um primo. O projeto do corpo do **switch** resultou de uma tentativa de otimização baseada no conhecimento de que `x` era mais comumente encontrado na faixa de 1 a 10.

```
    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# se difere do de seus predecessores é que a expressão de controle e as construções `case` podem ser cadeias em C#.

A sentença `case` de Ada é um descendente da sentença de seleção múltipla que apareceu no ALGOL W em 1966. A forma geral dessa construção é:

```
case expressão is
    when lista de escolhas => sequência_de_sentencias;
    ...
    when lista de escolhas => sequência_de_sentencias;
    [when others => sequência_de_sentencias;]
end case;
```

onde a expressão é de um tipo ordinal (inteiro, booleano, caractere ou tipo enumeração) e a cláusula `when others` é opcional.

As listas de escolhas das sentenças `case` de Ada são geralmente literais únicos, mas também podem ser subfaixas, como 10..15. Elas também podem usar operadores OU, especificados pelo símbolo |, para criar uma lista de literais. Por exemplo, a seguinte lista poderia aparecer como uma lista de escolha: 10|15|20. A cláusula `when others` é usada para valores não representados. Ada requer que a lista de escolhas seja exaustiva, o que fornece um pouco mais de confiabilidade porque proíbe o erro de inadvertidamente omitir um ou mais valores de escolha. A maioria das sentenças `case` de Ada inclui uma cláusula `when others` para garantir que a lista de escolhas seja exaustiva. Como as listas de escolha devem ser exaustivas, nunca existe a questão sobre o que fazer quando a expressão de controle tem um valor não representado.

Os valores nas listas de escolha devem ser mutuamente exclusivos; ou seja, uma constante não pode aparecer em mais de uma lista de escolhas. Além disso, os literais (eles também podem ser constantes nomeadas) nas listas de escolha devem ser do mesmo tipo da expressão.

A semântica do `case` em Ada é: a expressão é avaliada, e o valor é comparado com os literais nas listas de escolha. Se um casamento é encontrado, o controle é transferido para a sentença anexada à constante casada. Quando a execução da sentença estiver completa, o controle é transferido para a primeira sentença seguinte à construção `case` como um todo. Então, a sentença `case` de Ada é mais confiável do que as sentenças `switch` das linguagens baseadas em C, que não têm saídas implícitas após os segmentos selecionáveis.

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, 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 usam o valor da última expressão avaliada. Uma das expressões **case** de Ruby 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 construção, e a cláusula **else** é opcional. Por exemplo⁴,

```
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**, mais parecida com um **switch**, tem a seguinte forma:

```
case expressão
when valor then
- sequência de sentenças
when valor then
- sequência de sentenças
[else
- sequência de sentenças]
end
```

Os valores **case** são comparados com as expressões **case**, um de cada vez, de cima para baixo, até que um casamento seja encontrado. A comparação é feita usando o operador relacional **==**, definido para todas as classes pré-definidas. Se o valor **case** for uma faixa, como **(1..100)**, **==** é definido como um teste inclusivo, retornando verdadeiro se o valor da expressão **case** está na faixa dada. Se o valor **case** for um nome de classe, **==** é definido para retornar

⁴ Este exemplo é de Thomas et al. (2005).

verdadeiro se o valor **case** for um objeto da classe de expressão **case** ou uma de suas superclasses. Se o valor **case** for uma expressão regular, **==** é definido como um simples casamento de padrão.

Considere o seguinte exemplo:

```
case int_val
when -1 then neg_count++
when 0 then zero_count++
when 1 then pos_count++
else puts "Error--int_val is out of range"
end
```

Perl, Python e Lua não têm construções de seleção múltipla.

8.2.2.3 Implementando estruturas de seleção múltipla

Uma construção de seleção múltipla é essencialmente um desvio de n -caminhos para segmentos de código, onde n é o número de segmentos selecionáveis. A implementação de tal construção deve ser feita com múltiplas instruções de desvios condicionais. Considere novamente a forma geral da construção **switch** do C, com **breaks**:

```
switch (expressão) {
    case expressão_constante_1: sentença_1;
        break;
    ...
    case constante_n: sentença_n;
        break;
    [default: sentença_n+1]
}
```

Uma tradução simples dessa construção é:

```
Código para avaliar a expressão em t
goto branches
rótulo_1: código para sentença_1
    goto out
...
rótulo_n: código para sentença_n
    goto out
default: código para sentença_n+1
    goto out
branches: if t = expressão_constante_1 goto rótulo_1
    ...
    if t = expressão_constante_n goto rótulo_n
    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 abordagem simples, mas ineficiente, aceitável quando o número de casos é pequeno, digamos menos de 10. Ela tem de fazer um número de testes equivalente, em média, a cerca de metade de número de casos para encontrar o correto. Para o caso padrão ser escolhido, todos os outros precisam ser testados. Em construções 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, que resultaria em aproximadamente tempos iguais (e pequenos) para escolher qualquer um dos segmentos selecionáveis. Se a linguagem permite faixas de valores para expressões **case**, como em Ada ou Ruby, o método de dispersão não é adequado. Para essas situações, uma tabela de busca binária de valores **case** e endereços de segmentos é melhor.

Se a faixa dos valores **case** é relativamente pequena e mais da metade da faixa completa de valores é representada, uma matriz cujos índices são os valores **case** e cujos valores são os rótulos de segmentos pode ser construída. 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 rápido.

É claro, escolher dentre 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 custa mais tempo de compilação.

8.2.2.4 Seleção múltipla usando **if**

Em muitas situações, uma construção **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 aliviar a legibilidade pobre de seletores de dois caminhos profundamente aninhados, algumas linguagens, como Perl e Python, estenderam-no especificamente para esse uso. A extensão permite que algumas das palavras especiais sejam deixadas de fora. Em particular, sequências **elseif** são substituídas com uma única palavra especial, e a palavra especial de fechamento no **if** aninhado não é usada. O seletor aninhado é então chama-

do de uma **cláusula elif**. Considere a seguinte construção 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 das duas. 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 construção *else-if* não é uma forma redundante de **switch**. Na verdade, nenhum dos seletores múltiplos em linguagens contemporâneas é tão geral quanto a construção *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 ver a diferença entre estruturas de seleção múltipla e construções *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 construção *else-if* podem usar a mesma estrutura de controle, exceto com um pouco mais de digitação.

NOTA HISTÓRICA

O Fortran I incluía uma sentença de controle de iteração de contagem chamado **DO**, que permaneceu o mesmo no Fortran II, IV e 66. O recurso distinto dessa sentença era que ela tinha um pós-teste, tornando-a diferente das sentenças de iteração de contagem de todas as outras linguagens de programação (Na verdade, a especificação do Fortran 66 não dita que sua sentença **DO** deve ser um laço com pós-teste. Entretanto, a maioria das implementações do Fortran 66 implementavam-no como tal).

O exemplo da construção *if-then-else-if* de Python acima pode ser escrito como a sentença **case** de Ruby:

```
case
when count < 10 then bag1 = True
when count < 100 then bag2 = True
when count < 1000 then bag3 = True
end
```

Construções *else-if* são baseadas em uma construção matemática comum, a expressão condicional. Linguagens de programação funcionais, discutidas no Capítulo 15, normalmente usam expressões condicionais como uma de suas construções básicas de controle.

8.3 SENTENÇAS DE ITERAÇÃO

Uma **sentença iterativa** é uma que faz com que uma sentença ou uma coleção de sentenças seja executada zero, uma ou mais vezes. Uma construção de iteração é também chamada de **um laço**. Todas as linguagens de programação desde Plankalkül incluem algum método de repetir a execução de segmentos de código. A iteração é a essência do poder da computação. Se a iteração não fosse possível, os programadores precisariam informar cada ação em sequência; os programas úteis seriam imensos e inflexíveis e levariam tempo inaceitavelmente longo para serem escritos e muita memória para serem armazenados.

A execução repetida de uma sentença é geralmente realizada em uma linguagem funcional por meio da recursão, em vez do uso de construções iterativas. A recursão em linguagens funcionais será discutida no Capítulo 15.

As primeiras construções iterativas em linguagens de programação eram diretamente relacionadas às matrizes. Isso resultava que, nos primeiros anos da era 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 têm sido 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 construção 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. Início e fim aqui 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 afeta o controle antes ou após a execução do corpo da construção. Uma terceira opção, que permite ao usuário decidir onde colocar o controle, é discutida na Seção 8.3.3.

O **corpo** de uma construção de iteração é a coleção de sentenças cuja execução é controlada pela sentença de iteração. Usamos o termo **pré-teste** para dizer que o teste para completar o laço ocorre antes que o corpo do laço seja executado e **pós-teste** para dizer que ele ocorre após a execução. A sentença de iteração e o corpo do laço associado formam uma **construção de iteração**.

Além das sentenças de iteração primárias, discutimos uma forma alternativa que está em uma classe própria: controle de iteração definido pelo usuário.

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 os laços controlados logicamente serem mais gerais do 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 do que as abordagens mais comuns de programação 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 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).

É também verdade, claro, que as construções de linguagem vivem mais tempo do que as arquiteturas de máquina. Por exemplo, o autor desconhece máquinas contemporâneas com uma instrução de desvio de três caminhos para implementar a sentença `IF` aritmética do Fortran, apesar de essa sentença inicialmente ser incluída nessa linguagem porque o IBM 704 tinha tal instrução meio século atrás.

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 aparente é o tipo inteiro, mas e as enumerações, caracteres e pontos-flutuantes? Outra questão é se a variável de laço é uma variável normal, em termos de escopo, ou se ela 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 difícil de ser entendido, então outra questão é se a flexibilidade adicional que pode ser ganha por permitir tais mudanças vale a pena em termos da complexidade adicional. Uma questão similar surge acerca do número de vezes e do momento específico em que os parâmetros de laço são avaliados: se for apenas uma vez, isso resulta em laços mais simples, porém mais inflexíveis.

A seguir, temos um resumo dessas questões de projeto:

- Qual é o tipo e o escopo da variável de laço?
- Deve ser legal para a variável ou para os parâmetros de laço serem 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 As sentenças Do no Fortran 95

O Fortran 95 tem duas sentenças de contagem diferentes e ambas usam a palavra chave `Do`. A forma geral de uma das sentenças `Do` é

`Do rótulo variável = inicial, final [, tamanho do passo]`

onde o rótulo é aquele da última sentença no corpo do laço, e o tamanho do passo, quando ausente, tem valor padrão 1. A variável de laço deve ser do tipo `Integer`; os parâmetros do laço podem ser expressões e ter valores negativos ou positivos. O tamanho do passo, quando presente, não pode ter o valor zero. Os parâmetros do laço são avaliados no início da execução da sentença `Do`, e os valores são usados para calcular um **contador de iterações**, que então tem o número de vezes que o laço deve ser executado. O laço é controlado pelo contador de iterações, não pelos parâmetros do laço, logo mesmo se os parâmetros forem modificados, o que é permitido, essas mudanças não podem afetar o controle do laço. O contador de iterações é uma variável interna inacessível para o código de usuário.

Construções `Do` podem ser iniciadas apenas por meio da sentença `Do`, tornando a sentença uma estrutura de única entrada. Considere a construção.

```
Do 10 Index = 1, 10
      ...
10 Continue
```

Uma descrição em semântica operacional da sentença `Do` do Fortran 95 é mostrada a seguir⁵:

```
init_value = init_expression
terminal_value = terminal_expression
step_value = step_expression
do_var = init_value
iteration_count =
    max(int((terminal_value - init_value + step_value) / step_value), 0)
loop:
    if iteration_count ≤ 0 goto out
    [loop body]
    do_var = do_var + step_value
    iteration_count = iteration_count - 1
    goto loop
out: ...
```

O Fortran 95 também tem uma segunda forma de `Do`:

```
[name:]Do variable = initial, terminal [, stepsize]
      ...
End Do [name]
```

Esse `Do` usa uma palavra especial (ou frase) de fechamento, `End Do`, em vez de uma sentença rotulada. A seguir, temos um exemplo do esqueleto de uma construção `Do` que usa essa segunda forma:

```
Do Count = 1, 10
      ...
End Do
```

8.3.1.3 A sentença `for` de Ada

A sentença `for` de Ada tem a seguinte forma:

```
for variável in [reverse] faixa_discreta loop
      ...
end loop;
```

⁵ Note que essa descrição falha se `init_value` for 0 e o `terminal_value` é o maior valor inteiro legal para a implementação, porque esses valores causam um transbordamento do tipo inteiro na computação de `iteration_count`.

Uma faixa discreta é uma subfaixa de um tipo inteiro ou de enumeração, como 1..10 ou Segunda..Sexta. A palavra reservada **reverse**, quando presente, indica que os valores da faixa discreta são atribuídos para a variável do laço em ordem inversa. Note que o **for** de Ada é mais simples do que o Do do Fortran, porque o tamanho do passo é sempre 1 (ou o próximo elemento da faixa discreta). O recurso mais interessante da sentença **for** de Ada é o escopo da variável do laço, que é a faixa do laço. A variável é implicitamente declarada na sentença **for** e sua declaração é implicitamente removida após o término do laço. Por exemplo, em

```
Count : Float := 1.35;
for Count in 1..10 loop
    Sum := Sum + Count;
end loop;
```

a variável **Float** chamada **Count** não é afetada pelo laço **for**. Após o término do laço, a variável **Count** ainda é do tipo **Float** com o valor de 1.35. Além disso, a variável do tipo **Float** **Count** é oculta do código no corpo do laço, sendo mascarada pelo contador do laço **Count**, o qual é implicitamente declarado como da faixa discreta **Integer**.

A variável de laço em Ada não pode ter valores atribuídos a ela no corpo do laço. Variáveis usadas para especificar a faixa discreta podem ser modificadas no laço, mas como a faixa é avaliada apenas uma única vez, essas mudanças não afetam o controle do laço. É ilegal desviar para o corpo do laço **for** em Ada. A seguir, temos uma descrição em semântica operacional do laço **for** em Ada:

```
[defina for_var (seu tipo é o tipo da faixa discreta)]
[avalie faixa discreta]
loop:
    if [não existem elementos mais elementos na faixa discreta] goto out
    for_var = [próximo elemento da faixa discreta]
    [corpo do laço]
    goto loop
out:
    [remova a definição de for_var]
```

Como o escopo da variável do laço é o corpo do laço, as variáveis são indefinidas após o término do laço, portanto seus valores lá são irrelevantes.

8.3.1.4 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.



Parte 1: Linguística e o nascimento de Perl

LARRY WALL

Larry Wall desempenha vários papéis – está envolvido na publicação de livros, de linguagens, de software e na educação de crianças (ele tem quatro filhos). Passou algum tempo estudando na Seattle Pacific University, em Berkeley, e na UCLA. Larry também trabalhou na Unisys, na Jet Propulsion Laboratories e na Seagate. A publicação de linguagens trouxe a ele a maior parte da sua fama (“e a menor parte do dinheiro”, diz): Larry é o autor da linguagem de *scripting* Perl.

PRIMEIRO, UM POCO SOBRE SUA EXPERIÊNCIA PROFISSIONAL

Qual foi o seu melhor emprego? Trabalhar na JPL (Jet Propulsion Laboratories) era muito divertido. Eu fazia tanto a administração do sistema quanto o desenvolvimento de software. O trabalho de administrador de sistema era interessante porque envolvia 90% de tédio e 10% de pânico, assim, sobrava muito tempo para trabalhar na linguagem Perl.

Qual foi o seu emprego mais estranho? Essa é uma pergunta difícil. A maioria dos meus empregos foi estranha, de uma forma ou de outra; é difícil de escolher. Vejamos... Fui conselheiro de um acampamento chamado “Crazy Horse”. Eu escrevia software de negócios em BASIC em um sistema Wang que podia ser manipulado apenas por meio de menus. Fui o primeiro violino da MusiComedy Northwest por diversos anos. Talvez minha experiência musical mais estranha tenha sido uma sessão de gravação na qual eu era o único que conseguia tocar a parte do violino. Eles me gravaram oito ou 10 vezes, para que eu soasse como se fosse toda a seção de violinos.

Qual é o seu emprego atual? Meu emprego atual é projetar o Perl 6. Infelizmente ninguém está me pagando para isso no momento. É claro, também não estava recebendo oficialmente pelas cinco versões anteriores. Mas seria legal se estivessem me pagando, ao menos não oficialmente, dessa vez. Talvez quando você ler isso terão me contratado para ser um garçom/ator em Hollywood.

LINGUÍSTICA E LINGUAGENS COMPUTACIONAIS

Você estudou linguística. Na época em que estava engajado nesses estudos, que carreiras considerava? Minha esposa e eu seríamos linguistas de campo na África e fomos por um tempo associados à Wycliffe Bible Translators, até que minhas (recentemente de-

senvolvidas) alergias a comidas específicas puseram um fim nisso. De qualquer forma, não sei se teria sido um linguista muito bom – provavelmente faria mais bem à classe ficando de fora da profissão e escrevendo Perl. Independentemente disso, ainda amo a linguística e tenho estudado japonês por conta própria nos últimos anos apenas para impedir as sinapses de coagularem, coalharem ou o que quer que aconteça com elas.

Como o seu interesse em línguas faladas se traduziu para linguagens de programação? [Antes] de eu escrever Perl, admito que minha linguística e minha ciência da computação foram um tanto quanto separadas. No lado da computação, escrevi diversos compiladores sem me perguntar por que a maioria das linguagens de computação não parece tão natural. Do lado linguístico, escrevi alguns programas de linguagem natural em LISP, mas em geral não considerei as linguagens naturais compatíveis à análise por computador. (Quem já usou o Babblefish sabe que esse ainda é o caso). Somente quando eu comecei a trabalhar na linguagem Perl me dei conta da existência de muitos princípios que fazem uma língua soar natural, e alguns desses princípios podem ser ensinados para os computadores sem os enlouquecer.

O NASCIMENTO DE PERL

Em que você estava trabalhando na Unisys nos dias que antecederam a criação de Perl? Eu era um administrador e programador de sistemas que fornecia suporte para um projeto secreto da NSA (National Security Agency). Se eu lhe desse detalhes, você teria de me matar. Enfim, passei muito tempo trancado em “salas de cobre”, como eles as chamavam, brincando com computadores que oficialmente não existiam.

Como surgiu a ideia de uma nova linguagem? Estávamos tentando fazer gerência de configuração pelo país usando um link encriptado de bai-

xa velocidade. Isso estava produzindo pilhas de dados textuais, mas pouca informação útil. Perl foi inicialmente projetada para navegar por meio de arquivos espalhados contendo texto, encontrar as porções importantes e gerar relatórios. Inclusive, as explicações oficiais para Perl refletem isso: Practical Extraction and Report Language (Linguagem Prática para Extração e Geração de Relatórios) e Pathologically Eclectic Rubbish Lister (Listador de Lixo Patologicamente Eclético).

Que necessidade de negócios não estava sendo satisfeita? Essencialmente, a necessidade de flexibilidade. Estávamos usando UNIX por causa de sua flexibilidade, mas o conjunto de ferramentas que o UNIX fornecia para a criação de *scripts* no interpretador de comandos era muito difícil de ser usado para navegarmos em volta e dentro de arquivos de texto da forma que precisávamos. A linguagem de programação awk era um passo na direção certa, mas não era o suficiente para mim. Descobri como poderia fazer melhor. As ferramentas do UNIX eram muito boas em alguns aspectos e terríveis em outros. O problema era que a abordagem básica de caixa de ferramentas estava tentando agir como uma linguagem extensível sem sucesso. Pensei que se pudesse destilar as coisas boas do UNIX em uma linguagem real com uma granularidade fina o suficiente para chegar aos bits específicos quando fosse preciso, as pessoas achariam isso útil. Esse é o motivo do nascimento do Perl 1.

Qual é o seu primeiro amor: projetar uma linguagem ou projetar um programa? Uma coisa leva a

“Somente quando eu comecei a trabalhar na linguagem Perl me dei conta da existência de muitos princípios que fazem uma língua soar natural, e alguns desses princípios podem ser ensinados para os computadores sem os enlouquecer.”

outra? Respondendo a segunda pergunta, eu não acho que alguém possa projetar uma linguagem sem antes usar uma linguagem. Aprendemos as linguagens naturalmente de forma ascendente, primeiro imitando suas várias partes, e apenas mais tarde abstraindo os princípios que mantêm a linguagem coesa. Os cientistas da computação gostam de pensar que podem projetar coisas de maneira descendente, mas isso funciona apenas quando você já sabe que tipo de respostas está buscando. Em geral, as linguagens projetadas para resolver tipos de problemas específicos não são interessantes – ao menos não para mim.

Então, tive de aprender a programar primeiro. Assim, não acho que isso possa ser chamado de “primeiro amor”. É mais como um “primeiro amor/ódio”. As pessoas que realmente amam programar não têm interesse em projetar novas linguagens de computadores. Você apenas projeta uma nova linguagem quando acha que a atual é frustrante de alguma maneira. Logo, se você quiser seguir a carreira de projetista de linguagens de computadores, deve se preparar para uma vida de frustrações. Se alguma vez se sentir satisfeito com sua nova linguagem, é um idiota arrogante. Isso não significa que você seja um bom projetista. É claro, você pode ser um idiota arrogante como eu e também um frustrado. Ou seja, com certeza você não é um caso perdido.

Como as sentenças de atribuição em C produzem resultados e podem ser consideradas expressões, as expressões em uma sentença **for** são geralmente 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. No 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** do 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 construção **for**:

```
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, a primeira expressão estar 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: não existem variáveis de laço ou parâmetros de laço explícitos. Todas as variáveis envolvidas podem ser modificadas no corpo do laço. As expressões são avaliadas na ordem informada previamente. Apesar de poder criar problemas, é legal desviar para o corpo de um laço **for**.

O **for** de C é mais flexível do que as sentenças de laço de contagem do Fortran e de Ada, porque cada uma das expressões pode ser composta de múltiplas 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 descrição em semântica operacional desse código 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 (e não tem) um corpo de laço. Todas as ações desejadas são parte da sentença **for** propriamente dita, em vez de em 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** do C99 e do C++ diferem daquela das versões antigas do C de duas maneiras. Primeiro, em adição a uma expressão aritmética, ela pode usar uma expressão booleana para o 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** é a partir de 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 com 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 muito mais complexos e menos confiáveis do que os laços de contagem de Fortran e Ada.

8.3.1.5 A sentença **for** de Python

A forma geral do **for** de Python é

```
for variável_do_laço in objeto:
  - corpo do laço
  [else:
    - cláusula senão]
```

À variável do laço é atribuído o valor do objeto, uma vez para cada execução do corpo do laço. A cláusula senão, quando presente, é executada se o laço terminar normalmente.

Considere o exemplo a seguir:

```
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.

Lua tem duas sentenças `for` diferentes, uma das quais é chamada de `for numérico`. Essa sentença é similar ao `DO` do Fortran. Uma diferença importante é que em Lua o escopo da variável de laço é restrito ao corpo do laço. O outro `for` de Lua é discutido na Seção 8.3.4.

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 do 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 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 do que os laços controlados por contador, os laços controlados logicamente têm poucas questões de projeto.

- O controle deve ser de pré 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, converte-o para o tipo `int` e o retorna.

Na versão com pré-teste (**while**), a 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. A única diferença real entre o **do** e o **while** é que o **do** sempre faz o corpo do laço ser executado ao menos uma vez. Em ambos os casos, a sentença pode ser composta.

As descrições em semântica operacional dessas duas sentenças são:

`while`

`loop:`

`if` expressão_de_controle é falsa `goto` out
[corpo do laço]

```
    goto loop
out: ...

do-while

loop:
[corpo do laço]
if expressão_de_controle é verdadeira goto loop
```

É legal tanto em C quanto em C++ 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 **boolean**. E, como Java não possui um **goto**, os corpos dos laços não podem ser acessados diretamente de qualquer lugar, apenas a partir de seu início.

O Fortran 95 não tem um laço lógico, nem com pré-teste, nem com pós-teste. Ada tem um laço lógico com pré-teste, mas nenhuma versão pós-teste do laço lógico.

Perl e Ruby têm dois laços lógicos com pré-teste: **while** e **until**. O **until** é similar ao **while**, mas usa o inverso do valor da expressão de controle. Perl também tem dois laços com pós-teste, que usam **while** e **until** como modificadores de sentenças em blocos **do**.

Laços 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.

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 do início ou o final deste. Como resultado disso, 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 saídas do laço posicionadas pelos usuários. Talvez a questão mais interessante é se um único laço ou diversos laços aninhados podem ser abandonados. As questões de projeto para tal mecanismo são:

- O mecanismo condicional deve ser uma parte integral 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 estrutura 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 o controle é transferido em vez disso para o 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.

Java e Perl têm sentenças similares ao **continue**, exceto que elas podem incluir rótulos que especificam qual laço deve ser continuado.

Tanto **last** quanto **break** fornecem múltiplas saídas de laços, o que pode parecer prejudicial para a legibilidade. Entretanto, condições não usuais que requerem o término do laço são tão comuns que tal construção 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 ao uso de múltiplos **breaks** para deixar mais de um nível de laços é muito pior para a legibilidade.

A motivação para 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

NOTA HISTÓRICA

Apesar de diversos estudiosos tem sugerido, foi Edsger Dijkstra que deu ao mundo da computação a primeira publicação amplamente lida sobre os perigos do goto. Em sua carta, ele escreveu “A sentença goto nos moldes de hoje é primitiva demais; é um convite muito grande para esculhambar o programa de alguém (Dijkstra, 1968a). Durante os primeiros anos após a publicação da visão de Dijkstra sobre o goto, diversas pessoas pediram publicamente pela banição ou ao menos restrições ao uso do goto. Dentre aqueles que não eram favoráveis pela eliminação completa estava Donald Knuth (1974), que argumentou que existem ocasiões nas quais a eficiência do goto supera seu prejuízo para a legibilidade.”

dito. Entretanto, os alvos de saídas de laços posicionadas pelo usuário devem ser abaixo da saída e podem apenas seguir imediatamente após o final de uma sentença composta.

8.3.4 Iteração baseada em estruturas de dados

Apenas um tipo de estrutura de iteração adicional resta ser considerado aqui: a iteração controlada por uma estrutura de dados. Em vez de ter um contador ou uma expressão booleana para controlar as iterações, esses laços são controlados pelo número de elementos em uma estrutura de dados. Perl, JavaScript, PHP, Java e C# têm tais sentenças.

Uma sentença de iteração geral baseada em dados usa uma estrutura de dados e uma função definida pelo usuário para navegar nos elementos da estrutura. Tal função é um iterador, chamado no início de cada iteração. 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 os

dados em cada nó devem ser processados em alguma ordem em particular. Uma sentença de iteração definida pelo usuário para a árvore sucessivamente configuraria 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 para obter o primeiro elemento da lista.

O iterador deve sempre lembrar qual o nó que ele apresentou pela última vez de forma que visite 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 falha em encontrar mais elementos.

A construção `for` das linguagens baseadas em C, por causa de 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 estão para ser processados. Se o nó raiz é apontado por uma variável chamada `root`, e se `traverse` é uma função que modifica 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 pré-definidos 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) + "<br />") ;
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 agora constroem 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.

Em C++, iteradores para tipos definidos pelo usuário, ou classes, são implementados como funções amigas (*friend functions*) à classe ou como classes iteradoras separadas.

Em Java, os elementos de uma coleção definida pelo usuário que implementa a interface `Collection` pode ser visitada iterativamente com uma implementação da interface `Iterator`. A interface `Iterator` tem três métodos fundamentais: `next`, `hasNext` e `remove`. O método `next` é o iterador, na verdade.

Ele lança uma exceção do tipo `NoSuchElementException` quando não existirem mais elementos na coleção que está sendo iterada. O método `hasNext`, normalmente chamado antes da chamada a `next`, retorna `true` se existe ao menos mais um elemento.

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 em objetos em uma coleção que implementa a interface `Iterable`. Por exemplo, se tivéssemos uma coleção do tipo `ArrayList`⁶ chamada `myList` de cadeias, a seguinte sentença iteraria por meio de todos os seus elementos, atribuindo cada um a `myElement`:

```
for (String myElement : myList) { ... }
```

⁶ Um `ArrayList` é uma coleção pré-definida, na verdade, uma matriz dinâmica de objetos que podem ser de qualquer tipo – ou seja, é uma coleção de referências a objetos de qualquer classe. Tal classe implementa a interface `Iterable`.

Essa nova sentença é chamada de “foreach”, apesar de sua palavra reservada ser **for**.

A sentença foreach do C# itera nos elementos de matrizes e de outras coleções.

Por exemplo

```
String[] strList = {"Bob", "Carol", "Ted", "Beelzebub"};
...
foreach (String name in strList)
    Console.WriteLine("Name: {0}", name);
```

A notação `{0}` no parâmetro para `Console.WriteLine` acima indica a posição na cadeia a ser mostrada quando o valor da primeira variável nomeada, `name` nesse exemplo, está para ser colocado.

Ruby inclui iteradores para cada uma de suas classes contêiner pré-definidas. Como elas são associadas com blocos, na verdade subprogramas em Ruby, são discutidas no Capítulo 9.

Lua inclui uma sentença de laço chamada de **for genérico**, uma construção de iteração para estruturas de dados. Como Lua tem apenas uma estrutura de dados, a tabela, o **for** genérico itera sobre essa estrutura. A forma geral do **for** genérico é:

```
for variável_1 [,variável_2] in iterador(tabela) do
...
end
```

Se apenas uma variável for informada, ela recebe as chaves da tabela. Se uma segunda variável é informada, ela recebe os valores da tabela. Então, para obter os valores, duas variáveis devem ser informadas.

As duas funções de iteração pré-definidas mais úteis são `pairs` e `ipairs`. Para uma tabela geral, `pairs` é usada. Ela retorna o próximo elemento da tabela, independentemente do tipo das chaves. Para uma tabela usada como uma matriz, `ipairs` é usada. Ela retorna o próximo índice e valor da matriz, até a primeira chave inteira que não está na tabela. Por exemplo, o código a seguir mostra os valores de uma matriz:

```
for index, value in ipairs(list) do
    print(value)
end
```

É claro, um usuário pode definir qualquer iterador útil.

8.4 DESVIO INCONDICIONAL

Uma **sentença de desvio incondicional** transfere o controle da execução para uma posição especificada no programa. O debate mais aquecido acerca do projeto de linguagens no fim dos anos 1960 era se o uso de desvios incondicionais deveria ser parte de qualquer linguagem de alto nível e, se 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, imposta ou pelo projeto da linguagem ou por padrões de programação, as sentenças `goto` podem tornar os programas muito difíceis de serem lidos e, como um resultado, altamente não confiáveis e com alto custo para serem mantidos.

Esses problemas são oriundos de uma das capacidades do `goto` de forçar qualquer sentença de programação após qualquer ordem na sequência de execução, independentemente se a sentença precede ou segue a executada anteriormente em ordem textual. A legibilidade é melhor quando a ordem de execução das sentenças é quase a mesma que aquela na qual elas aparecem – em nosso caso, significa de cima para baixo, ou seja, a ordem a que estamos acostumados. Logo, restringir os `gos` de forma que possam transferir o controle apenas para baixo em um programa alivia parcialmente o problema. Isso permite o uso de `gos` 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. Entretanto, a maioria das linguagens populares o inclui. Kernighan e Ritchie (1978) afirmam que o uso do `goto` é exagerado, mas é incluído mesmo assim na linguagem de Ritchie, C. As linguagens que têm eliminado o `goto` fornecem sentenças de controle adicionais, normalmente na forma de saídas de laços, para codificar uma de suas aplicações mais justificáveis. Relativamente nova, C# inclui um `goto`, apesar de uma das linguagens nas quais ela se baseia, Java, não o fazer. 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 `gos` restritos e não prejudiciais à legibilidade. Na verdade, pode-se argumentar que elas melhoraram 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). Sua motivação primária era fornecer sentenças de controle que suportariam uma metodologia de projeto de programas para garantir a corretude durante o desenvolvimento, em vez de ter de depender de verificação ou testes de programas completos para garantir sua corretude. Essa metodologia é descrita em Dijkstra (1976). Outra motivação para desenvolver comandos protegidos é o não determinismo ser às vezes necessário em programas concorrentes, como será discutido no Capítulo 13. Uma motivação adicional é a clareza no entendimento de que é possível com os comandos protegidos. De forma simples, um segmento selecionável de uma construção de seleção em uma de comando protegido pode ser considerado independentemente de qualquer outra parte da construção, o que não é verdade para as de seleção das linguagens de programação comuns.

Comandos protegidos são cobertos neste capítulo porque formam a base para dois mecanismos linguísticos desenvolvidos posteriormente para programação concorrente em duas linguagens, CSP (Hoare, 1978) e Ada. A concorrência em Ada é discutida no Capítulo 13. Eles também são úteis para definir funções em Haskell, conforme discutido no Capítulo 15.

A construção 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 do 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 construção de seleção, consistindo em uma expressão booleana (uma guarda) e uma sentença ou sequência de sentenças é chamada de **comando protegido**.

A construção de seleção tem a aparência de uma seleção múltipla, mas sua semântica é diferente. Todas as expressões booleanas são avaliadas a cada vez que a construção é alcançada durante a execução. Se mais de uma expressão é verdadeira, uma das sentenças correspondentes pode ser escolhida de maneira não determinística para execução. Uma implementação pode sempre escolher a sentença associada com a primeira expressão booleana avaliada como verdadeira. Mas ela pode escolher qualquer sentença associada com uma expressão booleana verdadeira. Logo, a corretude do programa não depende de qual sentença é escolhida (dentre 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 força o programador a considerar e listar todas as possibilidades, como na sentença **case** de Ada.

Considere o 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 construção escolhe não deterministicamente entre a primeira e a terceira sentenças de atribuição. Se i é igual a j e não é zero, um erro em tempo de execução ocorre porque nenhuma das condições é igual a zero.

Essa construção 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 para 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 uma diferença prática entre essas duas construções. A primeira atribui x para max quando x e y são iguais; a segunda atribui y para max na mesma circunstância. A escolha entre as duas construções 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 construção é que todas as expressões booleanas são avaliadas em cada iteração. Se mais de uma for verdadeira, uma das sentenças associadas é não deterministicamente (talvez aleatoriamente) escolhida para execução, após o 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 inteiiras, q_1 , q_2 , q_3 e q_4 , rearranje seus valores de forma que $q_1 \leq q_2 \leq q_3 \leq q_4$. Sem comandos protegidos, uma solução direta é colocar os valores em uma matriz, ordená-la e então atribuir os valores da matriz de volta para as variáveis escalares q_1 , q_2 , q_3 e q_4 . Embora essa solução não seja difícil, ela requer uma 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 uma forma mais concisa e elegante⁷.

```
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 construções de controle de comandos protegidos de Dijkstra são interessantes, em parte porque ilustram como sintaxe e semântica das sentenças podem ter um impacto na verificação de programas e vice-versa. A verificação de programas é praticamente impossível quando sentenças `goto` são usadas. A verificação é amplamente simplificada se apenas laços lógicos e seleções ou apenas comandos protegidos forem usados. 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 os desvios incondicionais de uma vez só. É claro, já existem problemas práticos o suficiente com o `goto` para condená-lo sem ao menos usar a razão teórica. Uma das necessidades legítimas principais para os

⁷ Este código aparece de uma forma levemente diferente em Dijkstra (1975).

gotos – saídas prematuras de laços – pode ser solucionada com sentenças de desvio altamente 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 bastante usada tomou esse passo; além disso, duvidamos que alguma o fará, 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 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 de C#, sem um ponto negativo óbvio. Outro exemplo é a estrutura de laço de contagem de muitas linguagens, especialmente quando a sentença é simples, como em Ada.

Não é tão claro que a utilidade de muitas das outras estruturas de controle que têm sido propostas faz com que valha a pena incluí-las em linguagens (Ledgard e Marcotty, 1975). Essa questão se relaciona com o 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, a simplicidade significa que apenas algumas sentenças devem estar presentes em uma linguagem e que todas devem ser simples.

A rica variedade de estruturas de controle no nível de sentenças que tem sido inventada mostra a diversidade de opiniões entre os projetistas de linguagens. Após toda a invenção, discussão e avaliação, não existe uma 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.

Uma nota final: as estruturas de controle das linguagens de programação funcionais e programação lógica são bastante diferentes das descritas neste capítulo. Esses mecanismos são discutidos em detalhe nos Capítulos 15 e 16, respectivamente.

RESUMO

As sentenças de controle das linguagens imperativas 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 do C# elimina o problema de confiabilidade de suas antecessoras ao proibir a desida implícita por meio de um segmento selecionado para o próximo segmento selecionável.

Sentenças de laço diferentes têm sido inventadas em grande número para as linguagens de alto nível, começando com o laço de contagem do Fortran chamado Do. A sentença **for** de Ada é, em termos de complexidade, o oposto. Ela implementa de maneira elegante apenas a formas mais necessárias dos laços de contagem. A sentença **for** do C é a construção de iteração mais flexível, apesar de essa flexibilidade levar a alguns problemas de confiabilidade.

As linguagens baseadas em C têm sentenças de saída para seus laços; essas substituem um dos usos mais comuns das sentenças **goto**.

Iteradores baseados em dados são construções de laço para processar estruturas de dados, como listas encadeadas, dispersões e árvores. A sentença **for** das linguagens baseadas em C permitem ao usuário criar iteradores para dados definidos por ele. A sentença **foreach** de Perl e C# é um iterador pré-definido 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.

O desvio incondicional, ou **goto**, é parte da maioria das linguagens imperativas. Seus problemas são muito discutidos e 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 construções de controle alternativas com características teóricas positivas. Apesar de elas não terem sido adotadas como construções de controle de uma linguagem, parte da semântica aparece nos mecanismos de concorrência de CSP e de Ada e nas definições de funções em Haskell.

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. Quais são as soluções para o problema de aninhamento para os seletores de dois caminhos?
8. Quais são as questões de projeto para sentenças de seleção múltipla?
9. Dentre quais duas características de linguagem existe um compromisso ao decidir se mais de um segmento selecionável é executado em uma construção de seleção múltipla?
10. O que é não usual acerca da sentença de seleção múltipla de C?
11. Em que linguagem prévia a sentença **switch** do C foi baseada?
12. Explique como a sentença **switch** de C# é mais segura do que a de Java.
13. Quais são as questões de projeto para todas as sentenças de controle iterativas?
14. Quais são as questões de projeto para as sentenças de laços controlados por contador?

15. O que é uma sentença de laço com pré-teste? O que é uma sentença de laço com pós-teste?
16. Explique como a sentença **Do** do Fortran funciona.
17. Qual é a diferença entre a sentença **for** de C++ e a de Java?
18. De que maneira a sentença **for** do C é mais flexível do que a de muitas outras linguagens?
19. Qual é o escopo de uma variável de laço em Ada?
20. O que a função **range** faz em Python?
21. Cite uma diferença fundamental entre o **for** numérico de Lua e o **Do** do Fortran.
22. Que linguagens contemporâneas não incluem um **goto**?
23. Quais são as questões de projeto para sentenças de laços controlados logicamente?
24. Qual é a principal razão para a invenção das sentenças de controle de laço posicionadas pelo usuário?
25. Quais são as questões de projeto para mecanismos de controle de laço posicionados pelo usuário?
26. Qual a vantagem da sentença **break** de Java em relação à sentença **break** do C?
27. Quais são as diferenças entre a sentença **break** de C++ e a de Java?
28. O que é um controle de iteração definido pelo usuário?
29. Que linguagens de programação bastante usadas pegam emprestado parte de seu projeto dos comandos protegidos de Dijkstra?

CONJUNTO DE PROBLEMAS

1. Descreva três situações nas quais uma combinação entre construções 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, do uso de indentação em Python para especificar sentenças compostas em construções de controle?
6. Analise os problemas potenciais de legibilidade com o uso de palavras reservadas de fechamento para sentenças de controle que são o inverso de suas palavras reservadas correspondentes, como as palavras reservadas **case-escac** do ALGOL 68. Por exemplo, considere erros de digitação comuns como o inverso de dois caracteres adjacentes.
7. Use o *Science Citation Index* para encontrar um artigo que referencia 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 construção de controle de laço que permite múltiplas saídas. Leia o artigo e escreva uma descrição em semântica operacional da construção.

9. Quais são os argumentos a favor e contra o uso exclusivo de expressões booleanas nas sentenças de controle em Java (em oposto a permitir também o uso de expressões aritméticas, com em C++)?
10. Em Ada, as listas de escolha da construção **case** devem ser exaustivas, de forma que não podem existir valores não representados na expressão de controle. Em C++, valores não representados podem ser capturados em tempo de execução com o seletor **default**. Se não existe um **default**, um valor não representado faz com que a construção completa seja pulada. Quais são os prós e contras desses dois projetos (Ada e C++)?
11. Explique as vantagens e as desvantagens da sentença **for** de Java, comparada com o **for** de Ada.
12. Descreva uma situação de programação na qual a cláusula senão na sentença **for** de Python seria conveniente.
13. Descreva três situações de programação específicas que requerem um laço de pós-teste.
14. Especule acerca da razão pela qual o controle pode ser transferido para dentro de uma construção 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. Fortran 95
- b. Ada
- c. C, C++, Java, ou C#
- d. Python
- e. Ruby

Assuma que todas as variáveis sejam 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 Problema 1, de forma que todas as variáveis e constantes sejam do tipo ponto flutuante, e troque a sentença

`k = k + 1`

por

`kk = 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. Fortran 95 (você provavelmente precisará pesquisar essa)
- b. Ada
- c. C, C++, Java, e C#
- d. Python
- e. Ruby

Assuma que todas as variáveis são do tipo inteiro. Discuta os méritos relativos do uso dessas linguagens para esse código em particular.

4. Considere o seguinte segmento de programa em C. Reescreva-o sem usar gotos ou **breaks**.

```
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 do 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, C# ou Ada. Compare a legibilidade de seu código com o do exemplo.

6. Considere o seguinte problema de programação: os valores de três variáveis inteiros – primeira, segunda e terceira – devem ser colocados em três variáveis max, meio e min, com os significados óbvios, sem usar matrizes ou sub-programas definidos pelo usuário ou pré-definidos. Escreva duas soluções para esse problema, uma que usa seleções aninhadas e outra que não as usa. Compare a complexidade e a legibilidade esperada das duas.
7. Escreva a seguinte construção **for** de Java em Ada:

```
int i, j, n = 100;
for (i = 0, j = 17; i < n; i++, j--)
    sum += i * j + 3;
```

8. Reescreva o segmento de programa em C do Exercício de Programação 4 usando sentenças **if** e **goto** em C.
9. Reescreva o segmento de programa em C do Exercício de Programação 4 em Java sem usar uma construção **switch**.

Capítulo 9

Subprogramas

- 9.1** Introdução
- 9.2** Fundamentos de 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** Subprogramas sobrecarregados
- 9.8** Subprogramas genéricos
- 9.9** Questões de projeto para funções
- 9.10** Operadores sobrecarregados definidos pelo usuário
- 9.11** Corrotinas

Subprogramas são os blocos de construção fundamentais dos programas e estão dentre os conceitos mais importantes no projeto de linguagens de programação. Neste capítulo, exploraremos o projeto de subprogramas, incluindo métodos de passagem de parâmetros, ambientes de referenciamento local, subprogramas sobre-carregados, subprogramas genéricos e os problemas de apelidos e efeitos colaterais que estão associados com subprogramas. Também incluímos uma breve discussão sobre corrotinas, as quais fornecem controle simétrico de unidades.

Métodos de implementação para subprogramas são discutidos no Capítulo 10.

9.1 INTRODUÇÃO

Dois recursos fundamentais de abstração podem ser incluídos em uma linguagem de programação: abstração de processos e de dados. No início da história das linguagens de programação de alto nível, apenas a abstração de processos estava incluída, e ela é um conceito central em todas as linguagens de programação. Nos anos 1980, entretanto, passou-se 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 reusar coleções de cartões de instruções em diferentes lugares de um programa. Em uma linguagem de programação moderna, tal coleção de sentenças é escrita como um subprograma. Esse reúso resulta em diferentes tipos de economias, incluindo espaço em memória e tempo de codificação. Esse reúso é também uma abstração, porque os detalhes de computação do subprograma são substituídos em um programa por uma sentença que chama o subprograma. Em vez de explicar como uma computação é feita em um programa, essa explicação (a coleção de sentenças no subprograma) ocorre por meio de uma sentença chamadora, efetivamente abstraindo os detalhes, aumentando a legibilidade de um programa ao expor sua estrutura lógica enquanto oculta os detalhes de baixo nível.

Os métodos de linguagens orientadas a objetos são muito relacionados aos subprogramas discutidos neste capítulo. As principais maneiras pelas quais os métodos se diferem uns dos outros são a forma como eles são chamados e as suas associações com classes e objetos. Apesar de essas características especiais dos métodos serem abordadas no Capítulo 12, os recursos que eles compartilham com subprogramas, como parâmetros e variáveis locais, são tratados neste capítulo.

9.2 FUNDAMENTOS DE SUBPROGRAMAS

9.2.1 Características gerais de subprogramas

Todos os subprogramas discutidos neste capítulo, exceto as corrotinas descritas na Seção 9.11, 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, implicando a existência de apenas um subprograma em execução em qualquer momento no tempo.
- O controle sempre retorna para o chamador quando a execução do subprograma termina.

Alternativas a essas características resultam em corrotinas (Seção 9.11) e unidades concorrentes (Capítulo 13).

Apesar de os subprogramas em Fortran terem entradas múltiplas, esse tipo de entrada em particular é relativamente sem importância porque não fornece nenhuma capacidade fundamentalmente diferente. Logo, neste capítulo, ignoraremos a possibilidade de entradas múltiplas em subprogramas Fortran.

9.2.2 Definições básicas

Uma **definição de subprograma** descreve a interface e as ações da abstração deste. Uma **chamada a subprograma** é a requisição explícita para que ele seja executado. Um subprograma é dito **ativo** se, após ter sido chamado, começou sua execução, mas ainda não a terminou. Os dois tipos fundamentais de subprogramas, os procedimentos e as funções são definidos e discutidos na Seção 9.2.4.

Um **cabeçalho de subprograma**, isto é, a primeira parte da definição, serve a diversos propósitos. Primeiro, especifica a unidade sintática subsequente como uma definição de subprograma de algum tipo em particular¹. O tipo do subprograma é especificado com uma palavra especial. Segundo, o cabeçalho fornece um nome para o subprograma. Terceiro, ele pode especificar uma lista de parâmetros.

Considere os exemplos de cabeçalhos:

```
Subroutine Adder(parameters)
```

Esse é um cabeçalho de um subprograma na forma de uma sub-rotina chamada `Adder`. Em Ada, o cabeçalho para esse subprograma seria:

```
procedure Adder(parameters)
```

Em Python, o cabeçalho de um subprograma tem a seguinte forma:

```
def adder(parameters):
```

Nenhuma palavra especial aparece no cabeçalho de um subprograma em linguagens além de Fortran e Ada para especificar seu tipo. Essas outras linguagens têm apenas um tipo de subprograma, as funções (e/ou métodos), e o cabeçalho de uma função é reconhecido pelo contexto em vez de por uma palavra especial. Por exemplo, em C o cabeçalho da função seria:

```
void adder(parameters)
```

¹ Algumas linguagens de programação incluem ambos os tipos de subprogramas, procedimentos e funções.

Isso serviria como o cabeçalho de uma função chamada `adder`, onde `void` indica que ela não retorna um valor.

Como com as sentenças compostas, as sentenças no corpo de uma função em Python devem ser indentadas, e o final do corpo é indicado pela primeira sentença que não o é (a primeira sentença que segue a definição da função). Uma característica das funções de Python que as separam das funções das outras linguagens de programação comuns é as sentenças de função `def` seriam executáveis. Quando uma sentença `def` é executada, ela atribui o nome dado para o corpo dado à função. Até que o `def` de uma função tenha sido executado, a função não pode ser chamada. Considere o seguinte esqueleto de exemplo:

```
if ...
    def fun(...):
        ...
else
    def fun(...):
        ...
```

Se a cláusula então dessa construção de seleção for executada, a versão da função `fun` pode ser chamada, mas não a da cláusula senão. De forma similar, se a cláusula senão é escolhida, sua versão da função pode ser chamada, mas não a da cláusula então.

Os métodos de Ruby diferem dos subprogramas de outras linguagens de programação de maneiras interessantes. Os métodos de Ruby em geral são definidos em definições de classes, mas também podem ser definidos fora de definições de classes, o que no caso eles são considerados métodos do objeto raiz, `Object`. Tais métodos podem ser chamados sem um objeto recebedor, como se fossem funções em C ou C++. Se um método em Ruby é chamado sem um recebedor, assume-se que ele é `self`. Se não existe um método com aquele nome na classe, as classes que a envolvem são buscadas, até `Object`, se necessário.

Em Lua, funções são entidades de primeira classe – elas podem ser armazenadas em estruturas de dados, passadas como parâmetros e retornadas a partir de funções. Todas as funções são anônimas, apesar de poderem ser definidas usando uma sintaxe que faz com que pareçam ter nomes. Por exemplo, considere as duas definições idênticas da função `cube`:

```
function cube(x) return x * x * x end

cube = function (x) return x * x * x end
```

A primeira usa a sintaxe convencional, enquanto a forma da segunda ilustra mais precisamente o uso de funções não nomeadas.

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, se for uma função, seu tipo de retorno. Em uma linguagem na qual os subprogramas têm tipos, estes são definidos pelo protocolo do subprograma.

Subprogramas podem ter declarações e definições, de forma similar às declarações e às definições de variáveis em C; no qual as primeiras podem ser usadas para fornecer informações de tipos, mas não definem variáveis. Uma declaração de variável em C, que usa o especificador `extern`, é usada para especificar em uma função que uma variável que lá é usada é definida em outro lugar. Declarações de subprogramas fornecem o protocolo do subprograma, mas não incluem seus corpos. Elas são necessárias em linguagens que não permitem referências para frente em um programa. Tanto no caso de variáveis quanto no de subprogramas, as declarações são necessárias para a verificação estática de tipos. No caso dos subprogramas, é o tipo dos parâmetros que precisa ser verificado. Declarações de funções são comuns em programas C e C++, onde são chamadas de **protótipos** e normalmente colocadas em arquivos de cabeçalhos.

Na maioria das outras linguagens (além de C e C++), os subprogramas não precisam de declarações, porque não existe o requisito de que precisam ser definidos antes de serem chamados.

9.2.3 Parâmetros

Subprogramas normalmente descrevem computações. Existem duas maneiras pelas quais um subprograma que não é um método pode ganhar acesso aos dados que estão para ser processados: pelo acesso direto a variáveis não locais (declaradas em outros lugares, mas visíveis no subprograma) ou pela passagem de parâmetros. Dados passados por meio de parâmetros são acessados por nomes locais ao subprograma. A passagem de parâmetros é mais flexível do que o acesso direto a variáveis não locais. Essencialmente, um subprograma com acesso via parâmetros aos dados que serão processados é uma computação parametrizada. Ele pode realizar sua computação em quaisquer dados que receber por seus parâmetros (presumindo que os tipos dos parâmetros são os esperados). Se o acesso a dados é por variáveis não locais, a única maneira de a computação ser realizada em diferentes dados é pela atribuição de novos valores para essas variáveis não locais entre chamadas ao subprograma. O acesso extensivo a variáveis não locais pode reduzir a confiabilidade. Variáveis visíveis ao subprograma em que o acesso é desejado geralmente acabam visíveis também quando o acesso a elas não é necessário. Esse problema foi discutido no Capítulo 5.

Apesar de os métodos também acessarem dados externos por meio de referências não locais e de parâmetros, os dados primários a serem processados por um método são o objeto pelo qual o método é chamado. Entretanto, quando um método faz acesso a dados não locais, os problemas de confiabilidade são os mesmos de subprogramas que não são métodos. Além disso, em uma linguagem orientada a objetos, o acesso de métodos a variáveis de classe (associadas com a classe, em vez de com um objeto) é relacionado ao conceito de dados não locais e deve ser evitado sempre que possível. Nesse caso, como no de uma função em C acessando dados não locais, o método pode ter o efeito colateral de modificar algo além de seus parâmetros ou de seus dados locais. Tais modificações complicam a semântica do método e o tornam menos confiável.

Em algumas situações, é conveniente ser capaz de transmitir computações, em vez de dados, como parâmetros para os subprogramas. Nesses casos, o nome do subprograma que implementa essa computação pode ser usado como um parâmetro. Essa forma de parametrização é discutida na Seção 9.6. Parâmetros de dados são discutidos na Seção 9.5.

Os parâmetros no cabeçalho do subprograma são chamados de **parâmetros formais**. Eles são às vezes vistos como variáveis fictícias porque não são variáveis no sentido usual: na maioria dos casos, são vinculados ao armazenamento apenas quando o subprograma é chamado, e essa vinculação é normalmente feita por meio de alguma outra variável do programa.

Sentenças de chamadas a subprogramas devem incluir o nome do subprograma e uma lista de parâmetros para serem vinculados aos parâmetros formais do subprograma, chamados de **parâmetros reais**. Eles devem ser distinguidos dos parâmetros formais, porque os dois têm diferentes restrições em seus formatos e seus usos são um pouco distintos.

Em praticamente todas as linguagens de programação, a correspondência entre parâmetros reais e formais – ou a vinculação dos reais aos formais – é feita pela posição: o primeiro parâmetro real é vinculado ao primeiro formal e assim por diante. Tais parâmetros são chamados de **parâmetros posicionais**. Esse é um método efetivo e seguro de relacionar parâmetros reais e seus parâmetros formais correspondentes, desde que a lista de parâmetros seja relativamente curta.

Quando as listas são longas, entretanto, é fácil para um programador cometer equívocos na ordem dos parâmetros reais na lista. Uma solução para esse problema é fornecer parâmetros com palavras-chave, nos quais o nome do parâmetro formal com o qual um real deve ser vinculado é especificado com o real. A vantagem dos parâmetros com palavras-chave é poder aparecer em qualquer ordem na lista de parâmetros reais. As funções em Python podem ser chamadas usando 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 com palavras-chave é que o usuário do subprograma deve saber os nomes dos parâmetros formais.

Além dos parâmetros com palavras-chave, Ada, Fortran 95 e Python também permitem parâmetros posicionais. Os dois podem ser misturados em uma chamada, como em

```
sumer(my_length,
      sum = my_sum,
      list = my_array)
```

A única restrição com essa abordagem é que após a aparição de um parâmetro com palavra-chave na lista, os restantes devem ser com palavras-chave. Essa

restrição é necessária porque uma posição não pode mais ser bem definida após a aparição de um parâmetro com palavra-chave.

Em Python, Ruby, C++, Fortran 95, Ada 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 cabeçalho de função em Python:

```
def compute_pay(income, exemptions = 1, tax_rate)
```

O parâmetro formal `exemptions` pode estar ausente em uma chamada a `compute_pay`; quando ele está ausente, o valor 1 é usado. Nenhuma vírgula é usada para um parâmetro real ausente em uma chamada em Python, porque o único valor de tal vírgula seria indicar a posição do próximo parâmetro, o que no caso não é necessário porque todos os parâmetros reais após um parâmetro real ausente devem ser informados com palavras-chave. Por exemplo, considere a seguinte chamada:

```
pay = compute_pay(20000.0, tax_rate = 0.15)
```

Em C++, que não suporta parâmetros com palavras-chave, as regras para parâmetros padrão são diferentes. Os parâmetros padrão devem aparecer por último, porque os parâmetros são posicionalmente associados. Uma vez que um parâmetro padrão é omitido em uma chamada, todos os formais restantes devem ter valores padrão. Um cabeçalho de função C++ para a função `compute_pay` pode ser escrito como a seguir:

```
float compute_pay(float income, float tax_rate,
                  int exemptions = 1)
```

Note que os parâmetros são reorganizados de forma que o com o valor padrão fica por último. Uma chamada de exemplo 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 formais no cabeçalho de definição do subprograma. Entretanto, em C, C++, Perl, JavaScript e Lua, isso não é requerido. Quando existem menos parâmetros reais em uma chamada do que parâmetros formais em uma definição de função, é responsabilidade do programador garantir que a correspondência de parâmetros, sempre posicional, e a execução do subprograma sejam sensíveis a isso.

Apesar de esse projeto, que permite um número variável de parâmetros, ser claramente passível de erros, ele é também algumas vezes conveniente. Por exemplo, a função `printf` do C pode imprimir qualquer número de itens (valores de dados e/ou cadeias 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 tanto uma matriz quanto uma lista de expressões, cujos valores são colocados em uma matriz para o compilador e fornecidas 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` é definida 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:

```
myObject.DisplayList(myList);  
myObject.DisplayList(2, 4, 3 * x - 1, 17);
```

Ruby suporta uma configuração de parâmetros reais complicada, mas altamente flexível. Os parâmetros iniciais são expressões, cujos objetos de valores são passados para os parâmetros formais correspondentes. Os parâmetros iniciais podem ser seguidos por uma lista de pares chave => valor, que são colocados em uma dispersão anônima e uma referência para essa dispersão é passada para o próximo parâmetro formal. Elas são usadas como um substituto para os parâmetros com palavras-chave, não suportados por Ruby. O item de dispersão pode ser seguido por um único parâmetro precedido por um asterisco. Esse parâmetro é chamado de *parâmetro formal de matriz*. Quando o método é chamado, o parâmetro formal de matriz é configurado para referenciar um novo objeto `Array`. Todos os parâmetros reais restantes são atribuídos para os elementos do novo objeto `Array`. Se o parâmetro real que corresponde ao parâmetro formal de matriz for uma matriz, ele deve ser precedido por um asterisco e ser o último parâmetro real². Então, Ruby permite um número variável de parâmetros de uma maneira similar à de C#. Como as matrizes em Ruby podem armazenar tipos diferentes, não há um requisito que diga que parâmetros reais passados para a matriz tenham de ter o mesmo tipo.

O esqueleto de definição de função de exemplo e chamada ilustra a estrutura de parâmetros de Ruby:

```
list = [2, 4, 6, 8]  
def tester(p1, p2, p3, *p4)  
    ...  
end
```

² Isso não é bem verdade, porque o parâmetro formal de matriz pode ser seguido por uma referência a um método ou a uma função, que é precedida por um e comercial (&).

```
...
tester('first', mon => 72, tue => 68, wed => 59, *list)
```

Dentro de `tester`, os valores de seus parâmetros formais são:

```
p1 é 'first'
p2 é {mon => 72, tue => 68, wed => 59}
p3 é 2
p4 é [4, 6, 8]
```

Python suporta parâmetros similares àqueles de Ruby. Os parâmetros formais iniciais são como na maioria das linguagens comuns, podem ser seguidos por uma matriz de constantes (chamada de *tupla* em Python), que é especificada ao preceder o parâmetro formal com um asterisco. Esse parâmetro, que se torna uma matriz quando o subprograma é chamado, recebe todos os parâmetros reais sem palavras-chave além daqueles que correspondem aos parâmetros iniciais. Por fim, o último parâmetro formal, especificado precedendo o parâmetro formal com dois asteriscos, torna-se uma dispersão (chamada de *dicionário* em Python). Os parâmetros reais que correspondem a esse parâmetro são pares chave = valor, os quais são colocados no parâmetro formal de dispersão. Considere o esqueleto de função de exemplo e a chamada a ela:

```
def fun1(p1, p2, *p3, **p4):
    ...
fun1(2, 4, 6, 8, mon=68, tue=72, wed=77)
```

Em `fun1`, os parâmetros formais têm os seguintes valores:

```
p1 é 2
p2 é 4
p3 é [6, 8]
p4 é {'mon': 68, 'tue': 72, 'wed': 77}
```

Lua usa um mecanismo simples para suportar um número variado de parâmetros – representados por reticências (...), que podem ser tratadas como uma matriz ou como uma lista de valores atribuída para uma lista de variáveis. Por exemplo, considere as duas funções de exemplo:

```
function multiply (...)
    local product = 1
    for i, next in ipairs{...} do
        product = product * next
    end
    return sum
end
```

Lembre-se de que `ipairs` é um iterador para matrizes (ele retorna o índice e o valor dos elementos de uma matriz, um elemento de cada vez). `{ ... }` é uma matriz dos valores dos parâmetros reais.

```
function DoIt (...)  
    local a, b, c = ...  
    ...  
end
```

Suponha que `DoIt` seja chamada com:

```
doit(4, 7, 3)
```

Nesse exemplo, `a`, `b` e `c` seriam inicializadas na função com os valores 4, 7 e 3, respectivamente.

O parâmetro com reticências não precisa ser o único – ele pode aparecer no fim de uma lista de parâmetros formais nomeados.

9.2.4 Blocos em Ruby

Na maioria das outras linguagens de programação, o processamento dos dados em uma matriz ou outra coleção é feito iterando sobre a estrutura de dados com um laço, processando cada elemento de dados no laço. Lembre-se de que Ruby inclui métodos de iteração para suas estruturas de dados. Por exemplo, a estrutura `Array` tem o método iterador, `each`, que pode ser usado para processar qualquer matriz. Isso é feito em Ruby especificando um bloco de código na chamada ao iterador. Tal bloco, uma sequência de sentenças que devem ser delimitadas por chaves ou por um par `do-end`, pode aparecer apenas depois de uma chamada a método. Além disso, ele deve iniciar na mesma linha que ao menos a última parte da chamada. Os blocos podem ter parâmetros formais, especificados entre barras verticais. O bloco passado para o subprograma chamado é chamado com uma sentença `yield`, que consiste da palavra reservada `yield` seguida dos parâmetros reais. O `yield` não pode incluir mais parâmetros reais do que o número de formais do bloco. O valor retornado por um bloco (para o `yield` que o chamou) é o da última expressão avaliada no bloco.

Os iteradores são usados para processar dados em uma estrutura de dados existentes. Entretanto, é possível usá-los quando os dados estão sendo computados no método iterador. Considere o exemplo simples de um método e duas chamadas a ele, incluindo um bloco:

```
# Um método para calcular e retornar os números de Fibonacci  
# até um limite  
def fibonacci(last)  
    first, second = 1, 1  
    while first <= last  
        yield first  
        first, second = second, first + second
```

```

    end
end

# Chama fibonacci com um bloco para mostrar os números
puts "Fibonacci numbers less than 100 are:"
fibonacci(100) { |num| print num, " "}
puts # Output a newline

# Chama-o novamente para somar os números e mostrar a soma
sum = 0
fibonacci(100) { |num| sum += num}
puts "Sum of the Fibonacci numbers less than 100 is:
#{sum}"

```

Note o uso de atribuições paralelas, similares àquelas em Perl, no método. Além disso, repare na notação `#{...}` no parâmetro do tipo cadeia para `puts` sendo usada para especificar que o valor da expressão envolvida pela notação deve ser convertida para uma cadeia e inserida na cadeia de caracteres. A saída desse código é:

```

Fibonacci numbers less than 100 are:
1 1 2 3 5 8 13 21 34 55 89
Sum of the Fibonacci numbers less than 100 is: 232

```

Blocos são fechamentos (*closures*), que retêm o ambiente do local onde foram definidos, incluindo as variáveis locais e o objeto atual, independentemente de onde eles foram chamados.

9.2.5 Procedimentos e funções

Existem duas categorias de subprogramas – procedimentos e funções – e ambas podem ser vistas como abordagens para estender a linguagem. Procedimentos são coleções de sentenças que definem computações parametrizadas. Essas computações são realizadas por sentenças de chamadas únicas. Como efeito disso, os procedimentos definem novas sentenças. Por exemplo, como Ada não tem uma sentença de ordenação, um usuário poderia construir um procedimento para ordenar matrizes de dados e usar uma chamada para o procedimento definido no local da sentença de ordenação indisponível. Em Ada, os procedimentos são chamados de procedimentos mesmo; em Fortran, são sub-rotinas.

Procedimentos podem produzir resultados na unidade de programa que os chamam por dois métodos. Primeiro, se existirem variáveis que não são parâmetros formais, mas mesmo assim estão visíveis tanto no procedimento quanto na unidade de programa chamadora, o procedimento pode modificá-las. Segundo, se o subprograma tem parâmetros formais que permitem a transferência de dados para o chamador, esses parâmetros podem ser modificados.

As funções se parecem estruturalmente com os procedimentos, mas são semanticamente modeladas como funções matemáticas. Se uma função é um

modelo fiel, não produz efeitos colaterais; ou seja, não modifica nem seus parâmetros nem quaisquer variáveis definidas fora dela. Tal função pura retorna um valor – que é o único efeito desejado. Na prática, muitas funções em programas têm efeitos colaterais.

Funções são chamadas por meio da aparição de seus nomes em expressões, com os parâmetros reais requeridos. O valor produzido pela execução de uma função é retornado para o código chamador, substituindo a chamada propriamente dita. 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.

Funções definem novos operadores definidos pelo usuário. Por exemplo, se uma linguagem não tem um operador de exponenciação, uma função poderia ser escrita para retornar o valor de um de seus parâmetros elevado à potência de outro parâmetro. Seu cabeçalho em C++ poderia ser

```
float power(float base, float exp)
```

que poderia ser chamada com

```
result = 3.4 * power(10.0, x)
```

A biblioteca padrão de C++ já inclui uma função similar chamada `pow`. Compare-a com a mesma operação em Perl, na qual a exponenciação é uma operação já disponível na linguagem:

```
result = 3.4 * 10.0 ** x
```

Em Ada, Python, Ruby, C++ e C#, os usuários podem sobrepor operadores por meio da definição de novas funções. Nessas linguagens, o usuário pode definir um operador de exponenciação para ser usado de forma bastante parecida com o operador de exponenciação disponível em Perl. A sobreposição de operador definida pelo usuário é discutida na Seção 9.10.

Algumas linguagens de programação, como Fortran e Ada, fornecem tanto funções quanto procedimentos. As linguagens baseadas em C têm apenas funções (e/ou métodos); entretanto, essas funções podem se comportar como procedimentos. Elas podem ser definidas para não retornar valores se seu tipo de retorno for `void`. Como expressões nessas linguagens podem ser usadas como sentenças, uma chamada autocontida para uma função `void` é legal. Por exemplo, considere o cabeçalho de função e chamada:

```
void sort(int list[], int listlen);
...
sort(scores, 100);
```

Os métodos de Java, C++ e C# são sintaticamente similares às funções de C.

9.3 QUESTÕES DE PROJETO PARA SUBPROGRAMAS

Subprogramas são estruturas complexas em linguagens de programação e, de corrente disso, existe uma longa lista de questões envolvidas em seu projeto. Uma óbvia é a escolha de um ou mais métodos de passagem de parâmetros a serem usados. A ampla variedade de abordagens usadas em várias linguagens é um reflexo da diversidade de opiniões sobre o assunto. Uma questão fortemente relacionada é se os tipos dos parâmetros reais serão verificados em relação aos tipos dos parâmetros formais correspondentes.

A natureza do ambiente local de um subprograma dita em algum grau sua própria natureza. A questão mais importante aqui é se as variáveis locais são alocadas estaticamente ou dinamicamente.

A seguir, existe a questão de se as definições de subprogramas podem ser aninhadas. Outro ponto é se os nomes dos subprogramas podem ser passados como parâmetros. Se puderem e a linguagem permitir que os subprogramas sejam aninhados, existe a questão do ambiente de referenciamento correto de um subprograma passado como parâmetro.

Por fim, existem as questões acerca de os subprogramas poderem ser sobrecarregados ou genéricos. Um **subprograma sobrecarregado** (*overloaded*) é aquele que tem o mesmo nome de outro subprograma no mesmo ambiente de referenciamento. Um **subprograma genérico** é um cuja computação pode ser feita em dados de diferentes tipos em diferentes chamadas.

A seguir, é mostrado um resumo dessas questões de projeto para subprogramas em geral. Questões adicionais especificamente associadas com funções são discutidas na Seção 9.9.

- As variáveis locais são alocadas estaticamente ou dinamicamente?
- As definições de subprogramas podem aparecer em outras definições de subprogramas?
- Que método ou métodos de passagem de parâmetros são usados?
- Os tipos dos parâmetros reais são verificados em relação aos tipos dos parâmetros formais?
- Se os subprogramas puderem ser passados como parâmetros e puderem ser aninhados, qual é o ambiente de referenciamento de um subprograma passado como parâmetro?
- Os subprogramas podem ser sobrecarregados?
- Os subprogramas podem ser genéricos?

Essas questões e exemplos de projeto são discutidos nas seções a seguir.

9.4 AMBIENTES DE REFERENCIAMENTO LOCAL

Esta seção discute as questões relacionadas a variáveis definidas em subprogramas. A questão de definições de subprogramas aninhados é também brevemente coberta.

9.4.1 Variáveis locais

Subprogramas podem definir suas próprias variáveis, definindo ambientes de referenciamento local. Variáveis definidas dentro de subprogramas são chamadas de **variáveis locais**, já que seu escopo é normalmente o corpo do subprograma.

Na terminologia do Capítulo 5, as variáveis locais podem ser estáticas ou dinâmicas da pilha. Se são dinâmicas da pilha, podem ser vinculadas ao armazenamento quando o subprograma começar a execução e desvinculadas quando a execução terminar. Existem diversas vantagens de variáveis dinâmicas da pilha, e a principal é a flexibilidade que fornecem ao subprograma. É essencial que os subprogramas recursivos tenham variáveis locais dinâmicas da pilha. Outra vantagem dessas variáveis é que o armazenamento no subprograma ativo pode ser compartilhado com as variáveis locais em todos os subprogramas inativos. Essa era uma vantagem grande quando os computadores tinham memórias menores.

Uma das principais desvantagens das variáveis locais dinâmicas da pilha é que existe o custo em termos de tempo necessário para alocar, inicializar (quando necessário) e liberar as variáveis para cada chamada ao subprograma. Outra é que acessos às variáveis locais dinâmicas da pilha devem ser indiretos, enquanto o acesso às estáticas pode ser direto³. Essa indireção é necessária porque o lugar na pilha onde uma variável local em particular residirá pode ser determinado apenas em tempo de execução (veja o Capítulo 10). Por fim, quando todas as variáveis locais são dinâmicas da pilha, os subprogramas não podem ser sensíveis ao histórico; ou seja, eles não retêm valores de dados de variáveis locais entre as chamadas. Algumas vezes, é conveniente ser capaz de escrever subprogramas sensíveis ao histórico. Um exemplo comum de uma necessidade por um subprograma sensível ao histórico é um cuja tarefa é gerar números pseudoaleatórios. Cada chamada para tal subprograma computa um número pseudoaleatório, usando o último computado. Ele deve armazenar o último em uma variável local estática. Corrotinas e subprogramas usados em construções de laços de iteração (discutidos no Capítulo 8) são outros exemplos de subprogramas que precisam ser sensíveis ao histórico.

A vantagem principal das variáveis locais estáticas em relação às locais dinâmicas da pilha é serem levemente mais eficientes – elas não requerem nenhuma sobrecarga em tempo de execução para alocação e liberação. Além disso, se acessadas diretamente, os acessos são obviamente mais eficientes. E, é claro, elas permitem aos subprogramas serem sensíveis ao histórico. A maior desvantagem das variáveis locais estáticas é sua inabilidade de suportar recursão. Além disso, seu armazenamento não pode ser compartilhado com as variáveis locais de outros subprogramas inativos.

³ Em algumas implementações, as variáveis estáticas também são acessadas indiretamente, eliminando essa desvantagem.

Na maioria das linguagens contemporâneas, as variáveis em um subprograma são, por padrão, dinâmicas da pilha. Em funções C e C++, as variáveis são dinâmicas da pilha, ao menos que sejam especificamente declaradas como estáticas (usando `static`). Por exemplo, na seguinte função C (ou C++), 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];
    return sum;
}
```

Os subprogramas em Ada e os métodos de C++, Java e C# têm apenas variáveis locais dinâmicas da pilha.

Conforme discutido no Capítulo 5, os implementadores de Fortran 95 podem escolher se as variáveis locais serão estáticas ou dinâmicas da pilha. Na verdade, como as versões de Fortran anteriores à 90 não permitiam recursão, não existia motivo suficiente para torná-las dinâmicas da pilha. As economias em termos de armazenamento não são vistas como suficientes para justificar a perda em eficiência. Os usuários do Fortran 95 podem forçar uma ou mais variáveis locais como estáticas independentemente da implementação ao listar seus nomes em uma sentença `Save`.

Em Fortran 95, um subprograma pode ser explicitamente especificado como recursivo, fazendo suas variáveis locais serem dinâmicas da pilha por padrão. A ideia de especificar que um subprograma em particular pode ser recursivamente chamado surgiu com PL/I. O propósito dessa especificação explícita é permitir que os subprogramas não recursivos sejam implementados de uma maneira mais eficiente. A seguir, está um exemplo de um esqueleto de uma sub-rotina recursiva:

```
Recursive Subroutine Sub ()
    Integer :: Count
    Save, Real :: Sum
    ...
End Subroutine Sub
```

Nessa sub-rotina, `Count` é dinâmica da pilha, porque o subprograma é definido como recursivo. `Sum` é estática porque é marcada com `Save`.

Em Python, as únicas declarações usadas em definições de métodos são para variáveis globais. Qualquer variável declarada como global em um método deve ser definida fora do método. Uma variável definida fora do método pode ser referenciada no método sem declará-la como global, mas não pode ter valores atribuídos a ela no método. Se o nome de uma variável global recebe uma atribuição em um método, ela é implicitamente declarada como local e a atribuição não perturba a variável global. Todas as variáveis locais em métodos Python são dinâmicas da pilha.

Apenas variáveis com escopo restrito são declaradas em Lua. Qualquer bloco, incluindo o corpo de uma função, pode declarar variáveis locais com a declaração `local`, como:

```
local sum
```

Todas as variáveis não declaradas em Lua são globais. De acordo com Ierusalimschy (2006), o acesso às variáveis locais em Lua é mais rápido do que o acesso às globais.

9.4.2 Subprogramas aninhados

A ideia de aninhar subprogramas começou com o Algol 60. A motivação era ser capaz de criar uma hierarquia tanto da lógica quanto dos escopos. Se um subprograma é necessário apenas dentro de outro, por que não colocá-lo lá e ocultá-lo do resto do programa? Como o escopo estático é normalmente usado em linguagens que permitem que os subprogramas sejam aninhados, isso também fornece uma maneira altamente estruturada de dar acesso a variáveis não locais em subprogramas que envolvem outros. Lembre que os problemas introduzidos com isso foram discutidos no Capítulo 5. Por um bom tempo, as únicas linguagens que permitiam subprogramas aninhados eram aquelas que descendiam diretamente do Algol 60 – Algol 68, Pascal e Ada. Muitas outras linguagens, incluindo todos os descendentes diretos do C, não permitem o aninhamento de subprogramas. Recentemente, algumas novas linguagens permitem isso. Dentre elas, JavaScript, Python, Ruby e Lua.

9.5 MÉTODOS DE PASSAGEM DE PARÂMETROS

Métodos de passagem de parâmetros são as maneiras como os parâmetros são transmitidos para os subprogramas ou a partir deles. Primeiro, focamos nos diferentes modelos semânticos de métodos de passagem de parâmetros. Então, discutimos os vários modelos de implementação inventados pelos projetistas de linguagens para esses modelos semânticos. A seguir, falamos sobre as escolhas de projeto de diversas linguagens imperativas e discutimos os métodos reais usados para os modelos de implementação. Por fim, consideramos as questões de projeto que um projetista de linguagens encara ao escolher entre os métodos.

9.5.1 Modelos semânticos de passagem de parâmetros

Parâmetros formais são caracterizados por um dos três modelos semânticos distintos: (1) receber dados a partir do parâmetro real correspondente; (2) transmitir dados para o parâmetro real; ou (3) fazer ambos.

Esses três modelos semânticos são chamados de **modo de entrada**, **modo de saída** e **modo de entrada e saída**, respectivamente. Por exemplo, considere um subprograma que recebe duas matrizes de valores inteiros (`int`) como parâmetros – `list1` e `list2`. O subprograma deve adicionar `list1` à `list2`

e retornar o resultado como uma versão revisada de `list2`. Além disso, o subprograma deve criar uma nova matriz a partir das duas dadas e retorná-la. Para esse subprograma, `list1` deve estar no modo entrada, porque ele não é modificado pelo subprograma. `list2` deve estar no modo de entrada e saída, porque o subprograma precisa do valor dado à matriz e deve retornar seu novo valor. A terceira matriz deve estar no modo de saída, porque nenhum valor inicial para ela e seu valor computado deve ser retornado ao chamador.

Existem dois modelos conceituais de como a transferência de dados ocorre na transmissão dos parâmetros: ou um valor real é copiado (para o chamador, para o chamado ou para ambos) ou um caminho de acesso é transmitido. Em geral, o caminho de acesso é um simples ponteiro ou referência. A Figura 9.1 ilustra os três modelos semânticos de passagem de parâmetros quando os valores são copiados.

9.5.2 Modelos de implementação de passagem de parâmetros

Diversos modelos têm sido desenvolvidos pelos projetistas de linguagem para guiarem a implementação dos três modos básicos de transmissão de parâmetros. Nas seções a seguir, discutimos diversos deles, destacando suas forças e fraquezas.

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, que age como uma variável local no subprograma, implementando a semântica de modo de entrada.

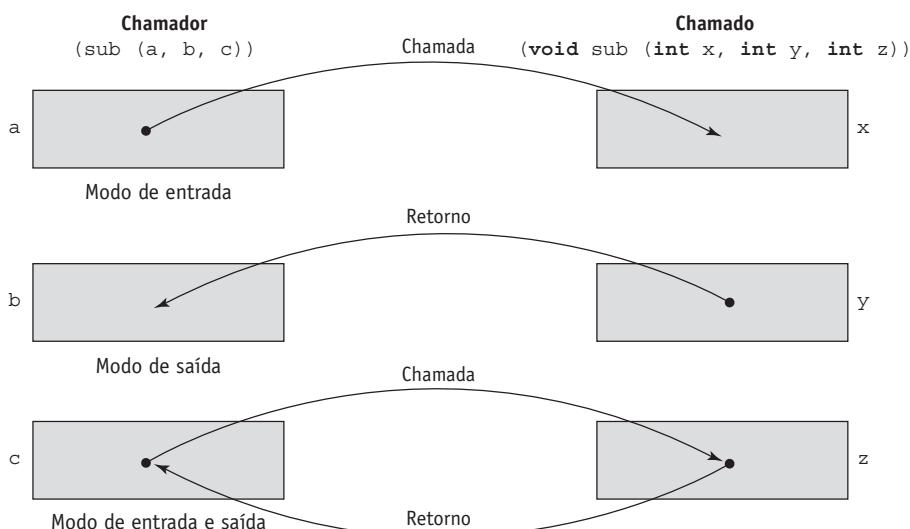


Figura 9.1 Os três modelos semânticos de passagem de parâmetros quando movimentações físicas são usadas.



Parte 2: Linguagens de *scripting* em geral e Perl em particular

LARRY WALL

Larry Wall desempenha vários papéis – está envolvido na publicação de livros, de linguagens, de software e na educação de crianças (ele tem quatro filhos). Passou algum tempo estudando na Seattle Pacific University, em Berkeley, e na UCLA. Larry também trabalhou na Unisys, na Jet Propulsion Laboratories e na Seagate. A publicação de linguagens trouxe a ele a maior parte da sua fama (“é a menor parte do dinheiro”, diz): Larry é o autor da linguagem de *scripting* Perl.

ALGUMAS ESPECIFICIDADES SOBRE LINGUAGENS DE *SCRIPTING*

Você pode compartilhar sua definição de uma linguagem de *scripting*? As pessoas frequentemente argumentam a respeito da diferença entre uma linguagem de *scripting* e uma linguagem de programação. Eu diria que um *script* é o que você dá aos atores, e um programa é o que você dá ao público. Ou seja, que um *script* é algo moldado em sua forma final pelo escritor, pelo diretor e pelos atores, enquanto um programa denota uma sequência relativamente fixa de eventos pré-determinados.

Não existe uma distinção forte e simples entre *scripting* e programação, e os *scripts* têm uma predisposição para se tornarem programas ao longo do tempo. Continuando com a nossa metáfora, enquanto você está escrevendo seu programa, ele se comporta mais como um *script*, e quando você termina de desenvolvê-lo, ele passa a atuar mais como um programa.

Certos tipos de linguagem facilitam a codificação no “espírito do momento”, assim tendem a ser classificadas como linguagens de *scripting*. Outras tentam simplificar a manipulação de dados complexos de maneira precisa e eficiente, às custas de especificar um monte de coisas desde o início antes mesmo de você saber o que realmente quer dizer. Essas linguagens tendem a ser chamadas de linguagens de “programação”. Perl geralmente é apresentada como uma linguagem de *scripting*, o que talvez seja uma redução demasiada de sua complexidade.

O que um leitor deveria entender sobre uma linguagem de *scripting*? O mais importante é que, embora as pessoas normalmente planejem jogar fora o protótipo, elas raramente o fazem. O que deveria ser um código exploratório e descartável geralmente vira “bom o suficiente” para permanecer em produção por muito mais tempo do que o imaginado. Assim, geralmente é melhor planejar com a expectativa de que seu protóti-

po se transformará em um projeto limpo e modular ao longo do tempo, assumindo que sua linguagem permitirá a evolução do programa, é claro. Nem todas as linguagens oferecem essa possibilidade.

MAIS SOBRE PERL E COMO ELA SURGIU

O que distingue Perl de outras linguagens de *scripting*? Acho que uma boa linguagem de *scripting* deve suportar a evolução de seu *script* em um “Programa Real”. Para esse fim, Perl oferece suporte completo à programação multiparadigma a fim de ajudar o suporte à escrita de *scripts*. Em outras palavras, Perl age como uma linguagem de computação madura quando necessário.

Scripts são como acessos para uma rodovia. A maioria das linguagens de programação tenta fornecer ou os acessos sem as rodovias ou as rodovias sem os acessos. Perl tenta oferecer ambos, por isso o slogan da linguagem é “Existe Mais de Uma Maneira de Fazer”. Perl não tenta forçar um estilo específico; alguns problemas podem ser mais bem resolvidos por meio de padrões, enquanto outros com programação funcional, com programação dirigida por eventos ou com programação orientada a objetos.

Em contraste, outras linguagens tentam forçar uma maneira particular de pensamento ou dificultam a interação com o mundo externo. Perl não pretende ser uma linguagem perfeita, mas útil, o que normalmente envolve cooperação com outras linguagens e sistemas de uma maneira humilde. Línguas não são uma utopia, nem tentam limitar a comunicação. O mesmo deveria valer para linguagens de computação.

Quando criou Perl, você pensou que ela seria o que é hoje? Por que ela é tão útil para a Internet? E, já que o marketing é tão importante, o que estava acontecendo na indústria ou no mundo da computação que ajudou Perl a decolar, em sua opinião? Eu sabia que Perl seria grande, mas não que seria tão grande quanto é.

Descobri desde o início que Perl tornaria as linguagens awk e sed obsoletas para o processamento de textos e, em um menor grau a programação em *shell* para unir interfaces aleatórias. Depois que adicionei todas as operações de administração de sistemas, não fiquei completamente surpreso por ela ter se tornado a linguagem escolhida pela maioria dos administradores de sistemas UNIX. O que eu não previa era que boa parte da Web seria prototipada em Perl. Como HTTP é um protocolo baseado em texto, entretanto, e os servidores Web precisam de código de cola para traduzir entre esse texto e os servidores de banco de dados, Perl estava no lugar certo na hora certa, com os recursos certos.

Esse é apenas mais um exemplo de como uma boa ferramenta é usada para objetivos não vislumbrados por seu criador. Mas isso não foi inteiramente um acidente; um bom criador tentará inserir recursos que podem ser usados por acaso quando a oportunidade aparecer.

“As pessoas frequentemente argumentam a respeito da diferença entre uma linguagem de *scripting* e uma linguagem de programação. Eu diria que um *script* é o que você dá aos atores, e um programa é o que você dá ao público.”

Qual parte da linguagem você mais gosta? A parte que eu mais gosto é que posso modificar as partes que menos gosto. Falando sério, as partes que mais gosto são aquelas conservadas no projeto do Perl 6. Por exemplo, Perl sempre foi ótima em evoluir e em cooperar com outros programas e linguagens em seu ambiente. Isso deve melhorar ainda mais. Perl continuará excelente em prototipagem rápida e em processamento de texto. Ela será melhor ainda em casamento de padrões. Tudo isso assumindo que terminaremos algum dia a versão 6 de Perl

Para finalizar, se você não trabalhasse com linguagens de *scripting*, o que faria? Provavelmente deixaria as pessoas a minha volta loucas. Ops!

A passagem por valor normalmente é implementada por cópia, porque os acessos são mais eficientes com essa abordagem. Ela poderia ser implementada transmitindo um caminho de acesso para o valor do parâmetro real no chamador, mas isso requereria que o valor estivesse em uma célula com proteção contra escrita (uma que pudesse ser apenas lida). Garantir a proteção contra escrita nem sempre é fácil. Por exemplo, suponha que o subprograma para o qual o parâmetro foi passado passou-o, em vez disso, para outro subprograma. Esse é outro motivo para usar transferência via cópia. Como veremos na Seção 9.5.4, C++ fornece um método conveniente e eficaz para forçar a proteção contra escrita em parâmetros com passagem por valor que são transmitidos por caminho de acesso.

A vantagem da passagem por valor é que para escalares ela é rápida, tanto em relação ao custo de vinculação quanto ao tempo de acesso. A principal desvantagem do método de passagem por valor se cópias são usadas é que é necessário armazenamento adicional para o parâmetro formal, no subprograma chamado ou em alguma área fora do chamador e do subprograma chamado. Além disso, o parâmetro real deve ser copiado para a área de armazenamento para o parâmetro formal correspondente. O armazenamento e as operações de cópia podem ser custosos se o parâmetro for grande, como uma matriz com muitos elementos.

9.5.2.2 Passagem por resultado

A **passagem por resultado** é um modelo de implementação para parâmetros do modo de saída. Quando um parâmetro é passado por resultado, nenhum valor é transmitido para o subprograma. O parâmetro formal correspondente age como uma variável local, mas logo antes de o controle ser transmitido de volta para o chamador, seu valor é transmitido de volta para o parâmetro real deste, que obviamente deve ser uma variável. (Como o chamador poderia referenciar o resultado computado se ele fosse um literal ou uma expressão?)

O método de passagem por resultado apresenta as mesmas vantagens e desvantagens da passagem por valor, mas algumas desvantagens adicionais. Se os valores são retornados por cópia (em vez de retornados por caminhos de acesso), como geralmente o são, a passagem por resultado também requer o armazenamento extra e as operações de cópia necessárias pela passagem por valor. Como com a passagem por valor, a dificuldade de implementar a passagem por resultado por meio da transmissão de um caminho de acesso normalmente resulta em sua implementação por cópia de dados. Nesse caso, o problema está em garantir que o valor inicial do parâmetro real não seja usado no subprograma chamado.

Um problema adicional do modelo de passagem por resultado é que pode existir uma colisão entre parâmetros reais, como a criada com a chamada

```
sub(p1, p1)
```

Em `sub`, assumindo que os dois parâmetros formais têm nomes diferentes, os dois podem obviamente receber valores diferentes. Assim, qualquer dos dois que for copiado para o seu parâmetro real correspondente por último se torna o valor de `p1`. Logo, a ordem pela qual os parâmetros reais são copiados determina seus valores. Por exemplo, considere o seguinte método em C#, que especifica o método passagem por resultado com o especificador `out` em seu parâmetro formal⁴.

```
void Fixer(out int x, out int y) {
    x = 17;
    y = 35;
}
...
f.Fixer(out a, out a);
```

Se ao final da execução de `Fixer` o parâmetro formal `x` é atribuído para seu parâmetro real correspondente primeiro, 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 é às vezes dependente da implementação para algumas linguagens, implementações diferentes podem produzir resultados distintos.

Chamar um procedimento com dois parâmetros reais idênticos também pode levar a diferentes tipos de problemas quando outros métodos de passagem de parâmetros são usados, conforme discutido na Seção 9.5.2.4.

Outro problema que pode ocorrer com a passagem por resultado é o implementador ser capaz de escolher entre dois tempos diferentes para avaliar os endereços dos parâmetros reais: no momento da chamada ou no do retorno. Por exemplo, considere o seguinte método C# e o 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 em que o endereço para o qual retornar o valor será determinado – na chamada ou no retorno. Se o endereço for computado na entrada do método, o valor 17 será retornado para `list[21]`; se computado logo antes do retorno, 17 será retornado para `list[42]`. Isso torna os programas não portáveis entre uma implementação que escolha avaliar os endereços para parâmetros no modo de saída no início de um subprograma e uma que escolha fazer essa avaliação no final.

⁴ O especificador `out` deve também ser especificado no parâmetro real correspondente.

9.5.2.3 Passagem por valor-resultado

A **passagem por valor-resultado** é um modelo de implementação para parâmetros no modo de entrada e saída no qual os valores reais são copiados. É, na verdade, uma combinação da passagem por valor com a passagem por resultado. O valor do parâmetro real é usado para inicializar o parâmetro formal correspondente, que então age como uma variável local. Na verdade, parâmetros formais com passagem por valor-resultado devem ter o armazenamento local associado com o subprograma chamado. Ao término do subprograma, o valor do parâmetro formal é transmitido de volta para o parâmetro real.

A passagem por valor-resultado é algumas vezes chamada de **passagem por cópia**, porque o parâmetro real é copiado para o formal na entrada do subprograma e então copiado de volta no término.

A passagem por valor-resultado compartilha com as passagens por valor e por resultado as desvantagens de requerer armazenamento múltiplo para parâmetros e tempo para copiar valores. Ela compartilha com a passagem por resultado os problemas associados com a ordem de atribuição dos parâmetros reais.

As vantagens da passagem por valor-resultado são relativas à passagem por referência, então são discutidas na Seção 9.5.2.4.

9.5.2.4 Passagem por referência

A **passagem por referência** é um segundo modelo de implementação para parâmetros no modo de entrada e saída. Em vez de copiar os valores de dados de um lado para o outro, como na passagem por valor-resultado, o método transmite um caminho de acesso, normalmente somente um endereço para o subprograma chamado. Isso fornece o caminho de acesso para a célula que está armazenando o parâmetro real. Logo, o subprograma chamado pode acessar o parâmetro real na unidade de programa chamadora. De fato, o parâmetro real é compartilhado com o subprograma chamado.

A vantagem da passagem por referência é que o processo de passagem por si só é eficiente, tanto em termos de tempo quanto de espaço. Não é necessário espaço duplicado nem cópias.

Existem, entretanto, diversas desvantagens com o método de passagem por referência. Primeiro, o acesso aos parâmetros formais será mais lento do que nos parâmetros com passagem por valor, por causa do nível de indireção adicional necessário⁵. Segundo, se apenas a comunicação de uma via para o subprograma chamado é necessária, mudanças inadvertidas e errôneas podem ser feitas ao parâmetro real.

Outro problema da passagem por referência é que podem ser criados apelidos. Esse problema deve ser esperado, porque a passagem por referência faz os caminhos de acesso ficarem disponíveis para os subprogramas chamados, fornecendo acesso para variáveis não locais. Existem diversas maneiras de criar os apelidos quando os parâmetros são passados por referência. O pro-

⁵ Isso é explicado com mais detalhes na Seção 9.5.3.

blema com esse tipo de apelido é o mesmo de outras circunstâncias: eles são prejudiciais para a legibilidade e, logo, para a confiabilidade e também tornam a verificação de programas mais difícil.

Existem várias maneiras pelas quais a passagem por referência pode criar apelidos. Primeiro, podem ocorrer colisões entre parâmetros reais. Considere uma função C++ com dois parâmetros para serem passados por referência (veja a Seção 9.5.3), como em

```
void fun(int &first, int &second)
```

Se a chamada a `fun` passasse a mesma variável duas vezes, como em

```
fun(total, total)
```

`first` e `second` em `fun` seriam apelidos.

Segundo, colisões entre elementos de uma matriz também podem causar apelidos. Por exemplo, suponha que a função `fun` fosse chamada com dois elementos de matriz especificados com índices variáveis, como em

```
fun(list[i], list[j])
```

Se esses dois parâmetros fossem passados por referência e `i` fosse igual a `j`, `first` e `second` seriam novamente apelidos.

Terceiro, se dois dos parâmetros formais de um subprograma forem um elemento de uma matriz e a matriz completa, e ambos forem passados por referência, então uma chamada como

```
fun1(list[i], list)
```

poderia resultar em um apelido em `fun1`, pois `fun1` pode acessar todos os elementos de `list` pelo segundo parâmetro e acessar um único elemento por seu primeiro parâmetro.

Além disso, outra maneira de obter apelidos com parâmetros com passagem por referência é por meio de colisões entre parâmetros formais e variáveis não locais que sejam visíveis. Por exemplo, considere o seguinte código em C:

```
int * global;
void main() {
    ...
    sub(global);
    ...
}
void sub(int * param) {
    ...
}
```

Dentro de `sub`, `param` e `global` são apelidos.

Todas essas possíveis situações que geram apelidos são eliminadas se a passagem por valor-resultado for usada em vez da passagem por referência. Entretanto, no lugar dos apelidos, outros problemas aparecem, conforme discutido na Seção 9.5.2.3

9.5.2.5 Passagem por nome

A **passagem por nome** é um método de transmissão de parâmetros com modo de entrada e saída que não corresponde a um modelo de implementação único. Quando os parâmetros são passados por nome, o parâmetro real é, na prática, textualmente substituído pelo parâmetro formal correspondente em todas as suas ocorrências no subprograma. Esse método é um tanto diferente daqueles discutidos até aqui, onde os parâmetros formais são vinculados aos valores reais ou endereços no momento da chamada ao subprograma. Um parâmetro formal com passagem por nome é vinculado a um método de acesso no momento da chamada ao subprograma, mas a vinculação real a um valor ou a um endereço é postergada até que o parâmetro formal receba uma atribuição ou seja referenciado.

Como a passagem por nome não é parte de nenhuma linguagem bastante usada, ela não é discutida em maiores detalhes aqui. Entretanto, ela é usada em tempo de compilação pelas macros em linguagens de montagem e para os parâmetros genéricos dos subprogramas genéricos em C++ e em Ada, como discutido na Seção 9.8.

9.5.3 Implementando métodos de passagem de parâmetros

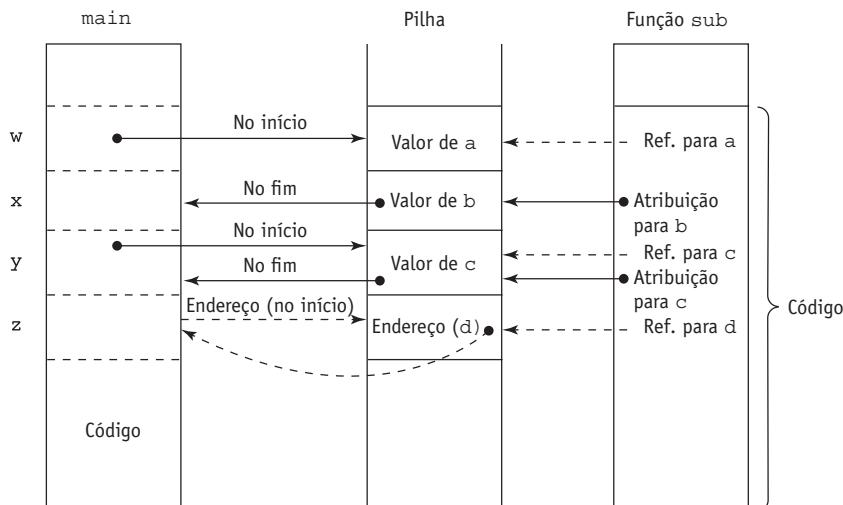
Na maioria das linguagens contemporâneas, a comunicação via parâmetros ocorre por meio da pilha de tempo de execução, inicializada e mantida pelo sistema de tempo de execução (um programa de sistema que gerencia a execução de programas). A pilha de tempo de execução é usada extensivamente para a ligação de controle de subprogramas e para a passagem de parâmetros, como será discutido no Capítulo 10. Na discussão a seguir, assumimos que a pilha é usada para todas as transmissões de parâmetros.

Parâmetros com passagem de valor têm seus valores copiados em posições da pilha, que servem como armazenamento para os parâmetros formais correspondentes. Parâmetros com passagem por resultado são implementados de maneira oposta. Os valores atribuídos para os parâmetros reais com passagem por resultado são colocados na pilha, onde podem ser obtidos pela unidade de programa chamadora após o término do subprograma chamado. Parâmetros com passagem por valor-resultado podem ser implementados diretamente a partir de suas semânticas como uma combinação de passagens por valor e por resultado. A posição da pilha para tal parâmetro é inicializada pela chamada e usada como uma variável local no subprograma chamado.

Parâmetros com passagem por referência são talvez 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 construir código para avaliá-la logo antes de transferir o controle para o subprograma chamado. O endereço da célula de memória na qual o código coloca o resultado de sua avaliação é então colocado na pilha. O compilador

deve se certificar de prevenir que o subprograma chamado modifique parâmetros que são literais ou expressões, como discutido posteriormente nesse capítulo. O acesso aos parâmetros formais no subprograma chamado é feito ao endereçar indiretamente a partir da posição da pilha do endereço. A implementação de passagem por valor, por resultado, por valor-resultado e por referência onde a pilha de tempo de execução é usada, é mostrada na Figura 9.2. O subprograma `sub` é chamado a partir de `main` com a chamada `sub (w, x, y, z)`, onde `w` é passado por valor, `x` por resultado, `y` por valor-resultado e `z` por referência.

Um erro sutil, mas fatal, pode ocorrer em parâmetros com passagens por referência e por valor-resultado se não forem tomados cuidados em sua implementação. Suponha que um programa contenha duas referências para a constante 10, a primeira como um parâmetro real em uma chamada a um subprograma. A seguir, suponha que o subprograma incorretamente modifique o parâmetro formal que corresponda ao 10 para o valor 5. O compilador para esse programa pode ter de construir uma única posição para o valor 10 durante a compilação, como os compiladores normalmente fazem, e usar essa posição para todas as referências para a constante 10 no programa. Mas, após o retorno do subprograma, todas as ocorrências subsequentes de 10 serão referências ao valor 5. Se for permitido que isso ocorra, é criado um problema de programação muito difícil de ser diagnosticado. Isso de fato ocorreu com muitas implementações do Fortran IV.



Cabeçalho da função: `void sub(int a, int b, int c, int d)`

Chamada da função em main: `sub (w, x, y, z)`
(passando `w` por valor, `x` por resultado, `y` por valor-resultado, `z` por referência)

Figura 9.2 Uma possível implementação e pilha dos métodos comuns de passagem de parâmetros.

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 atingida pelo uso de ponteiros como parâmetros. O valor do ponteiro é disponibilizado para a função chamada e nada é copiado de volta. Entretanto, como o que foi passado é um caminho de acesso para os dados do chamador, a função chamada pode modificar os dados do chamador. C copiou esse uso do método de passagem por valor do ALGOL 68. Tanto em C quanto em C++, os parâmetros formais podem ser tipados como ponteiros para constantes. Os parâmetros reais correspondentes não precisam ser constantes, mas, nesses casos, eles sofrem coerção para constantes, permitindo que parâmetros de ponteiros forneçam a eficiência da passagem por referência com a semântica de uma via da passagem por valor. A proteção contra escrita desses parâmetros na função chamada é implícita.

C++ inclui um tipo especial de ponteiro, chamado de *tipo referência*, conforme discutido no Capítulo 6, frequentemente usado para parâmetros. Parâmetros de referência são implicitamente desreferenciados na função ou método, e sua semântica é de passagem por referência. C++ também permite que os parâmetros de referência sejam definidos como constantes. Por exemplo, poderíamos ter

```
void fun(const int &p1, int p2, int &p3) { ... }
```

NOTA HISTÓRICA

O ALGOL 60 introduziu o método de passagem por nome. Ele também permite passagem por valor como uma opção. Principalmente por causa da dificuldade de implementação, a passagem por nome não foi reusada a partir do ALGOL 60 por nenhuma outra linguagem subsequente que tenha se tornado popular (além do SIMULA 67).

NOTA HISTÓRICA

O ALGOL W (Wirth e Hoare, 1966) introduziu o método de passagem de parâmetros por valor-resultado 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 modificado 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 com modo de entrada não são exatamente iguais. Parâmetros constantes claramente implementam o modo de entrada. Entretanto, em todas as linguagens imperativas comuns, exceto em Ada, os parâmetros em modo de entrada podem receber atribuições no subprograma independentemente de essas mudanças nunca serem refletidas nos valores dos parâmetros reais correspondentes. Parâmetros constantes nunca podem receber atribuições.

Como em C e C++, todos os parâmetros em Java são passados por valor. Entretanto, como os objetos podem ser acessados apenas por meio de variáveis de referência, parâmetros de objetos são, na prática, passados por referência. Apesar de uma referência a objeto passada como parâmetro não poder ela própria ser modificada no subprograma chamado, o objeto referenciado pode ser mudado se um método puder causar a mudança. Como variáveis de referência não podem apontar para variáveis escalares diretamente e Java não tem ponteiros, os escalares não podem ser

passados por referência em Java (apesar de uma referência para um objeto que contenha um escalar possa).

Os projetistas de Ada definiram versões dos três modos semânticos de transmissão de parâmetros: entrada, saída e entrada e saída. Os três são apropiadamente chamados com as palavras reservadas **in**, **out** e **in out**, onde **in** é o método padrão. Por exemplo, considere o seguinte cabeçalho de subprograma em Ada:

```
procedure Adder(A : in out Integer;
                 B : in Integer;
                 C : out Float)
```

Os parâmetros formais de Ada declarados do modo de saída (**out**) podem receber atribuições, mas não ser referenciados. Parâmetros que estão no modo de entrada (**in**) podem ser referenciados, mas não receber atribuições. Naturalmente, parâmetros do modo de entrada e saída (**in out**) podem ser tanto referenciados quanto receber atribuições.

Em Ada 95, todos os escalares são passados por cópia e todos os parâmetros estruturados são passados por referência.

O Fortran 95 é similar a Ada no sentido de que seus parâmetros formais podem ser declarados com os modos **in**, **out** ou **in out**, usando o atributo **Intent**. Por exemplo, considere o seguinte início de um subprograma em Fortran 95:

```
Subroutine Adder(A, B, C)
  Integer, Intent(Inout) :: A
  Integer, Intent(In) :: B
  Integer, Intent(Out) :: C
```

Os modos semânticos dos parâmetros desse subprograma são os mesmos que do exemplo em Ada anterior.

O método de passagem de parâmetros padrão de C# é a passagem por valor. A passagem por referência pode ser especificada precedendo tanto um parâmetro formal quanto 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 por valor.

C# suporta parâmetros no 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 **out**.

A passagem de parâmetros de PHP é similar à de C#, exceto que tanto o parâmetro real quanto o parâmetro formal podem especificar passagem por referência. A passagem por referência é especificada ao preceder um ou mais parâmetros com o comercial.

Perl emprega uma maneira primitiva de passagem de parâmetros. Todos os parâmetros reais são implicitamente colocados em uma matriz pré-definida chamada de @_ (de todas as coisas!). O subprograma obtém os valores dos parâmetros reais (ou endereços) dessa matriz. O fato mais peculiar sobre essa matriz é sua natureza mágica, exposta pelo fato de que seus elementos são na verdade apelidos para os parâmetros reais. Logo, se um elemento de @_ é modificado no subprograma chamado, essa mudança é refletida no parâmetro real correspondente na chamada, assumindo que existe um parâmetro real correspondente (o número de parâmetros reais não precisa ser o mesmo que o de parâmetros formais) e que ele seja uma variável.

O método de passagem de parâmetros de Python e Ruby é chamado de **passagem por atribuição**. Como todos os valores de dados são objetos, todas as variáveis são referências para objetos. Na passagem por atribuição, o valor do parâmetro real é atribuído ao parâmetro formal. Logo, a passagem por atribuição é, na prática, uma passagem por referência, porque os valores de todos os parâmetros reais são referências. Entretanto, apenas em certos casos isso resulta em uma semântica de passagem de parâmetros do tipo por referência. Por exemplo, muitos objetos são essencialmente imutáveis. Em uma linguagem orientada a objetos pura, o processo de modificar o valor de uma variável com uma sentença de atribuição, como em

```
x = x + 1
```

não muda o objeto referenciado por x. Em vez disso, ela pega o objeto referenciado por x, incrementa-o em 1, criando um novo objeto (com o valor x + 1) e troca x para referenciar ao novo objeto. Então, quando uma referência para um objeto escalar é passada para um subprograma, o objeto referenciado não pode ser mudado no local. Como a referência é passada por valor, mesmo que o parâmetro formal seja mudado no subprograma, essa mudança não tem efeito no parâmetro real no chamador.

Agora, suponha que uma referência para uma matriz seja passada como um parâmetro. Se o parâmetro formal correspondente receber a atribuição de um novo objeto do tipo matriz, não existem efeitos para o chamador. Entretanto, se o parâmetro formal é usado para atribuir um valor para um elemento da matriz, como em

```
list[3] = 47
```

o parâmetro real é afetado. Portanto, trocar a referência do parâmetro formal não tem efeito no chamador, mas trocar um elemento da matriz passada como um parâmetro tem.

9.5.5 Verificando os tipos dos parâmetros

Atualmente, é amplamente aceito que a confiabilidade de software requer que os tipos dos parâmetros reais sejam verificados em relação à sua consistência com os tipos dos parâmetros formais correspondentes. Sem a verificação de tipos, pequenos erros tipográficos podem levar a erros de programas difíceis de diagnosticar porque não são detectados nem pelo

compilador nem pelo sistema de tempo de execução. Por exemplo, na chamada a função

```
result = sub1(1)
```

o parâmetro real é uma constante inteira. Se o parâmetro formal de `sub1` é do tipo ponto flutuante, nenhum erro será detectado sem a verificação dos tipos dos parâmetros. Apesar de um inteiro com valor 1 e um ponto flutuante com valor 1 terem o mesmo valor, as representações de ambos são muito diferentes. O subprograma `sub1` não pode produzir um resultado correto dado um valor de parâmetro real inteiro quando espera um valor de ponto flutuante.

As primeiras linguagens de programação, como o Fortran 77 e a versão original do C, não requeriam a verificação de tipos dos parâmetros; a maioria das linguagens posteriores solicita isso. Entretanto, as relativamente recentes Perl, JavaScript e PHP não requerem.

C e C++ provocam alguma discussão especial na questão de verificação de tipos de parâmetros. No C original, nem o número dos parâmetros nem seus tipos eram verificados. No C89, os parâmetros formais das funções podem ser definidos de duas maneiras. Primeiro, eles podem ser como no C original; ou seja, os nomes dos parâmetros são listados em parênteses e as declarações de tipos para eles a seguir, como em

```
double sin(x)
double x;
{ ... }
```

O uso desse método evita a verificação de tipos, permitindo que chamadas como

```
double value;
int count;
...
value = sin(count);
```

sejam legais, apesar de elas nunca serem corretas.

A alternativa é chamada de método **protótipo**, no qual os tipos dos parâmetros formais são incluídos na lista, como em

```
double sin(double x)
{ ... }
```

Se essa versão de `sin` fosse chamada do mesmo jeito, ou seja

```
value = sin(count);
```

ela também seria legal. O tipo do parâmetro real (`int`) é verificado em relação ao parâmetro formal (`double`). Apesar de eles não casarem, `int` pode sofrer coerção para `double` (é uma coerção de alargamento), então a conversão é feita. Se a conversão não fosse possível (por exemplo, se o parâmetro real fosse uma matriz) ou se o número de parâmetros estivesse errado, um erro semântico seria detectado. Então, no C89, o usuário escolhe se os parâmetros serão verificados em relação ao tipo.

No C99 e em C++, todas as funções devem ter seus parâmetros formais na forma de protótipos. Entretanto, a verificação de tipos pode ser evitada para alguns dos parâmetros ao substituir a última parte da lista de parâmetros com reticências, como em

```
int printf(const char* format_string, ...);
```

Uma chamada a `printf` deve incluir ao menos um parâmetro, um ponteiro para uma cadeia de caracteres constante. Além disso, tudo (incluindo nada) é permitido. A maneira pela qual `printf` determina se existem parâmetros adicionais é pela presença de símbolos especiais no parâmetro do tipo cadeia. Por exemplo, o código de formato para uma saída inteira é `%d`. Isso aparece como parte da cadeia, como em

```
printf("The sum is %d\n", sum);
```

O `%` diz à função `printf` que existe mais um parâmetro.

Existe mais uma questão interessante com as coerções de parâmetros reais para formais quando tipos primitivos são passados como referência, como em C#. Suponha que uma chamada a um método passe um valor `float` para um parâmetro formal `double`. Se esse parâmetro é passado por valor, `float` sofre coerção para `double` e não há problema. Essa coerção em particular é muito útil, pois permite que uma biblioteca forneça versões para `double` de subprogramas que podem ser usados tanto para `float` quanto para `double`. Entretanto, suponha que o parâmetro seja passado por referência. Quando o valor do parâmetro formal `double` é retornado para o parâmetro real `float` no chamador, o valor transbordará sua posição de memória. Para evitar esse problema, o C# requer que o tipo de um parâmetro real `ref` case exatamente com o tipo de seu parâmetro formal (nenhuma coerção é permitida).

Em Python e em Ruby, não existe verificação de tipos de parâmetros, porque a tipagem nessas linguagens é um conceito diferente. Os objetos têm tipos, mas as variáveis não, então os parâmetros formais são desprovidos de tipos. Isso desabilita a própria ideia de verificação de tipos de parâmetros.

9.5.6 Matrizes multidimensionais como parâmetros

As funções de mapeamento de armazenamento usadas para mapear os valores de índices de referências a

NOTA HISTÓRICA

A definição da linguagem Ada 83 especifica que parâmetros escalares (não estruturados) devem ser passados por cópia; ou seja, parâmetros no modo de entrada ou no modo de entrada e saída devem ser variáveis locais inicializadas copiando o valor do parâmetro real correspondente. Parâmetros simples do modo de saída ou de entrada e saída devem ter seus valores copiados novamente para o parâmetro real correspondente no término do subprograma. A ordem dessas cópias, quando existe mais de uma, não é definida pela definição da linguagem. A avaliação de parâmetros com modo de saída ou de entrada e saída é feita antes que ocorra a transferência de controle para o subprograma chamado. Por exemplo, suponha que um parâmetro real com modo de saída tem o formato

`List (Index)`

O valor do endereço desse parâmetro é computado no momento da chamada. Se acontecer de `Index` ser visível no subprograma chamado e o subprograma o modificar, o endereço do parâmetro não será afetado.

No caso de parâmetros formais que são matrizes ou registros, foi dada a escolha aos implementadores de Ada 83 entre a passagem por valor-resultado e a passagem por referência. Ao não especificarem o método de implementação para
(continua)

NOTA HISTÓRICA

(continuação)

passar parâmetros estruturados, os projetistas de Ada 83 deixaram aberta a possibilidade de um problema sutil. O problema é que os dois métodos de implementação podem levar a diferentes resultados para certos programas. Essa diferença pode ocorrer porque o método de passagem por referência fornece acesso à posição no programa chamador que também pode ser fornecido se o parâmetro real é visível como uma variável global, ou se o mesmo parâmetro real é passado para dois parâmetros formais, em ambos os casos criando um apelido. Se a passagem por valor-resultado for usada no lugar da passagem por referência, esse acesso duplo ao parâmetro real não é possível.

Um problema adicional é: suponha que o subprograma termine de forma anormal (via uma exceção); o parâmetro real na implementação de passagem por valor-resultado permanecerá intocado, enquanto o método de implementação por passagem por referência pode ter trocado o parâmetro real correspondente antes que o erro acontecesse. Mais uma vez, pode existir uma diferença entre os dois métodos de implementação.

Programas em Ada 83 que produzem resultados diferentes dependendo de como o método de entrada e saída é implementado são tachados de errôneos. Independente desse rótulo, entretanto, não existe uma maneira de o compilador detectar a condição errônea. Então, o erro é normalmente detectado apenas quando o usuário move o programa de uma implementação para outra e percebe que ele não se comporta mais da mesma maneira. A filosofia de projeto do Ada 83 nessa situação é que os programadores devem se proteger contra apelidos: se eles criarem apelidos, devem combater os problemas em potencial.

elementos de matrizes multidimensionais para endereços em memória foram discutidas detalhadamente no Capítulo 6. Em algumas linguagens, como em C e C++, quando uma matriz multidimensional é passada como um parâmetro para um subprograma, o compilador deve ser capaz de construir a função de mapeamento para essa matriz vendo apenas o texto do subprograma (não do subprograma chamador). Isso é verdade porque os subprogramas podem ser compilados separadamente dos programas que os chamam. Considere o problema de passar uma matriz para uma função em C. Matrizes multidimensionais em C são realmente vetores de vetores, e elas são armazenadas em ordem principal de linha. A seguir, está uma função de mapeamento de armazenamento para ordem principal de linha para matrizes quando o limite inferior de todos os índices é 0 e o tamanho do elemento é 1:

$$\text{endereço}(\text{mat}[i, j]) = \text{endereço}(\text{mat}[0, 0]) + i * \text{número_de_colunas} + j$$

Note que essa função de mapeamento precisa do número de colunas, mas não do número de linhas. Logo, em C e C++, quando uma matriz é passada como um parâmetro, o parâmetro formal deve incluir o número de colunas no segundo par de colchetes. Isso é ilustrado neste esqueleto de programa em C:

```
void fun(int matrix[] [10]) {
    ...
}
void main() {
    int mat [5] [10];
    ...
    fun(mat);
    ...
}
```

O problema com esse método de passar matrizes como parâmetros é que ele não permite ao programador escrever uma função que possa aceitar matrizes com diferentes números de colunas; uma nova função deve ser escrita para cada matriz com um número diferente de colunas. Isso, na prática, proíbe a escrita de funções flexíveis que poderiam ser efetivamente reusáveis se as funções trabalhassem com matrizes multidimensionais. Em C e C++, existe uma forma de contornar o problema da inclusão de aritmética de ponteiros. A matriz pode ser

passada como um ponteiro, e as dimensões reais da matriz, incluídas como parâmetros. Então, a função pode avaliar a função de mapeamento de armazenamento escrita pelo usuário usando aritmética de ponteiros cada vez que um elemento da matriz precisar ser referenciado. Por exemplo, considere o protótipo de função:

```
void fun(float *mat_ptr,
        int num_rows,
        int num_cols);
```

A seguinte sentença pode ser usada para mover o valor da variável `x` para o elemento [linha] [coluna] do parâmetro `matrix` em `fun`:

```
* (mat_ptr + (row * num_cols) + col) = x;
```

Apesar de isso funcionar, obviamente é difícil de ler e, devido à sua complexidade, é passível de erros. A dificuldade com a leitura disso pode ser aliviada pelo uso de uma macro para definir a função de mapeamento de armazenamento, como

```
#define mat_ptr(r,c) (*mat_ptr + ((r) *
                                (num_cols) + (c)))
```

Com isso, a atribuição pode ser escrita como

```
mat_ptr(row,col) = x;
```

Outras linguagens usam abordagens diferentes para lidar com o problema de passagem de matrizes multidimensionais. Os compiladores Ada são capazes de determinar o tamanho das dimensões de todas as matrizes usadas como parâmetros no momento em que os subprogramas são compilados. Em Ada, tipos matrizes sem restrições podem ser parâmetros formais. Um tipo matriz sem restrição é um no qual as faixas de índices não são dadas na definição de tipo da matriz. Definições de variáveis de tipos matrizes sem restrições devem incluir faixas de índices. O código em um subprograma que recebe uma matriz sem restrições pode obter a informação de faixa de índices do parâmetro real associado com tais parâmetros. Por exemplo, considere as seguintes definições:

```
type Mat_Type is array (Integer range <>, Integer range
<>) of Float;
Mat_1 : Mat_Type(1..100, 1..20);
```

Uma função que retorna a soma dos elementos de matrizes de `Mat_Type` é mostrada a seguir:

```
function Sumer(Mat : in Mat_Type) return Float is
    Sum : Float := 0.0;
begin
    for Row in Mat'range(1) loop
```

```

for Col in Mat'range(2) loop
    Sum := Sum + Mat(Row, Col);
end loop; -- for Col ...
end loop; -- for Row ...
return Sum;
end Sumer;

```

O atributo **range** retorna a faixa de índices do índice nomeado da matriz do parâmetro atual, então funciona independentemente do tamanho ou das faixas de índice do parâmetro.

Em Fortran, o problema é tratado da seguinte forma. Os parâmetros formais que são matrizes devem ter uma declaração após o cabeçalho. Para matrizes de dimensão única, os índices nas declarações são irrelevantes. Mas para matrizes multidimensionais, os índices em tais declarações permitem que o compilador construa a função de mapeamento do armazenamento. Considere o seguinte esqueleto de sub-rotina em Fortran:

```

Subroutine Sub(Matrix, Rows, Cols, Result)
    Integer, Intent(In) :: Rows, Cols
    Real, Dimension(Rows, Cols), Intent(In) :: Matrix
    Real, Intent(In) :: Result
    ...
End Subroutine Sub

```

Isso funciona perfeitamente, desde que o parâmetro real `Rows` tenha o valor usado para o número de linhas na definição da matriz passada. O número de linhas é necessário porque o Fortran armazena as matrizes em ordem maior de coluna. Se a matriz a ser passada não está atualmente preenchida com dados úteis até o tamanho definido, tanto os tamanhos de índices definidos quanto os tamanhos de índices preenchidos podem ser passados para o subprograma. Então, os tamanhos definidos são usados na declaração local da matriz, e os tamanhos de índices preenchidos são usados para controlar a computação na qual os elementos da matriz são referenciados. Por exemplo, considere o subprograma em Fortran:

```

Subroutine Matsum(Matrix, Rows, Cols, Filled_Rows,
                   Filled_Cols, Sum)
    Real, Dimension(Rows, Cols), Intent(In) :: Matrix
    Integer, Intent(In) :: Rows, Cols, Filled_Rows,
                           Filled_Cols
    Real, Intent(Out) :: Sum
    Integer :: Row_Index, Col_Index
    Sum = 0.0
    Do Row_Index = 1, Filled_Rows
        Do Col_Index = 1, Filled_Cols
            Sum = Sum + Matrix(Row_Index, Col_Index)
        End Do
    End Do
End Subroutine Matsum

```

Java e C# usam uma técnica para passar matrizes multidimensionais como parâmetros similar a de Ada. Em Java e C#, matrizes são objetos. Elas são todas de uma única dimensão, mas os elementos podem ser matrizes. Cada matriz herda uma constante nomeada (`length` em Java e `Length` em C#) configurada para seu tamanho quando o objeto matriz é criado. O parâmetro formal para uma matriz aparece com dois conjuntos de colchetes vazios como no seguinte método em Java que faz o que a função de exemplo `Sumer` em Ada faz:

```
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, nelas as linhas podem ter diferentes tamanhos.

9.5.7 Considerações de projeto

Duas considerações importantes estão envolvidas na escolha dos métodos de passagem de parâmetros: a eficiência e se transferências de dados de uma via ou de duas vias são necessárias. Princípios contemporâneos de engenharia de software ditam que o acesso pelo código de subprogramas a dados de fora do subprograma deve ser minimizado. Com esse objetivo em mente, os parâmetros de modo de entrada devem ser usados quando nenhum dado deve ser retornado por meio de parâmetros para o chamador. Parâmetros no modo de saída devem ser usados quando nenhum dado é transferido para o subprograma chamado, mas o subprograma deve transmitir dados de volta para o chamador. Por fim, parâmetros no modo de entrada e saída devem ser usados apenas quando os dados devem se mover em ambas as direções entre o chamador e o subprograma chamado.

Existe uma consideração prática em conflito com esse princípio. Algumas vezes, é justificável passar caminhos de acesso para a transmissão de parâmetros de uma via. Por exemplo, quando uma grande matriz está para ser passada para um subprograma que não a modifica, um método de um caminho é preferido. Entretanto, a passagem por valor requereria que a matriz inteira fosse movida para uma área de armazenamento local do subprograma. Isso pode ser custoso tanto em tempo quanto em espaço. Por causa disso, grandes matrizes são normalmente passadas por referência. Essa é a razão pela qual a definição de Ada 83 permitia aos implementadores escolherem entre os dois métodos para parâmetros estruturados. Os parâmetros de referência constantes de C++ oferecem outra solução. Outra abordagem seria permitir que o usuário escolhesse entre os métodos.

A escolha de um método de passagem de parâmetros para funções é relacionada a outra questão de projeto: efeitos colaterais funcionais discutidos na Seção 9.9.

9.5.8 Exemplos de passagem de parâmetros

Considere a função em C:

```
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 = c      — Mover o primeiro valor de parâmetro para dentro
b = d      — Mover o segundo valor de parâmetro para dentro
temp = a
a = b
b = temp
```

Apesar de `a` terminar com o valor de `d` e `b` terminar com o valor de `c`, os valores de `c` e de `d` não são modificados porque nada é transmitido de volta para o chamador.

Podemos modificar a função `swap` em C para tratar parâmetros do tipo ponteiro para atingir o efeito da passagem por referência:

```
void swap2(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

`swap2` pode ser chamada com

```
swap2(&c, &d);
```

As ações de `swap2` podem ser descritas como

```
a = &c      — Mover o primeiro valor de parâmetro para dentro
b = &d      — Mover o segundo valor de parâmetro para dentro
temp = *a
*a = *b
*b = temp
```

Nesse caso, a operação de troca é bem-sucedida: os valores de *c* e *d* são trocados. *swap2* pode ser escrita em C++ usando parâmetros de referência como:

```
void swap2(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```

Essa operação simples de troca não é possível em Java, porque ela não tem nem ponteiros nem os tipos de referência 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 àquela da passagem por referência, exceto quando apelidos estão envolvidos. Lembre-se de que Ada usa passagem por valor-resultado para parâmetros escalares no modo de entrada e saída. Para explorar a passagem por valor-resultado, considere a seguinte função, *swap3*, que assumimos usar passagem por valor-resultado. Ela é escrita em uma sintaxe similar a 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	– Mover o primeiro valor de endereço para dentro
addr_d = &d	– Mover o segundo valor de endereço para dentro
a = *addr_c	– Mover o primeiro valor de parâmetro para dentro
b = *addr_d	– Mover o segundo valor de parâmetro para dentro
temp = a	
a = b	
b = temp	
*addr_c = a	– Mover o primeiro valor de parâmetro para fora
*addr_d = b	– Mover o segundo valor de parâmetro para fora

Então, mais uma vez, esse subprograma de troca opera corretamente. A seguir, considere a chamada

```
swap3(i, list[i]);
```

Nesse caso, as ações são

addr_i = &i	– Mover o primeiro valor de endereço para dentro
addr_listi = &list[i]	– Mover o segundo valor de endereço para dentro
a = *addr_i	– Mover o primeiro valor de parâmetro para dentro
b = *addr_listi	– Mover o segundo valor de parâmetro para dentro
temp = a	
a = b	
b = temp	
*addr_i = a	– Mover o primeiro valor de parâmetro para fora
*addr_listi = b	– Mover o segundo valor de parâmetro para fora

Mais uma vez, o subprograma opera corretamente, nesse caso porque os endereços para os quais os valores dos parâmetros serão retornados são computados no momento da chamada em vez de no momento do retorno. Se os endereços dos parâmetros reais fossem computados no momento do retorno, os resultados seriam errados.

Por fim, vamos explorar o que acontece quando apelidos estão envolvidos com as passagens por valor-resultado e por referência. Considere o seguinte esqueleto de programa escrito em sintaxe similar a 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 a passagem por referência for usada, `i` e `a` são apelidos. Se a passagem por valor-resultado é usada, `i` e `a` não são apelidos. As ações de `fun`, assumindo passagem por valor-resultado, são

addr_i = &i	– Mover o primeiro valor de endereço para dentro
addr_listi = &list[i]	– Mover o segundo valor de endereço para dentro

```
a = *addr_i           – Mover o primeiro valor de parâmetro  
b = *addr_listi      – para dentro  
i = b                – Mover o segundo valor de parâmetro  
*addr_i = a           – para dentro  
*addr_listi = b       – Configura i como 5  
                      – Mover o primeiro valor de parâmetro  
                      – para fora  
                      – Mover o segundo valor de parâmetro  
                      – para fora
```

Nesse caso, a atribuição para a variável global *i* em *fun* modifica seu valor de 3 para 5, mas a cópia de volta do primeiro parâmetro formal (a antepenúltima linha no exemplo) a configura novamente para 3. A observação importante aqui é que se a passagem por referência for usada, o resultado é que a cópia de volta não é parte da semântica, e *i* permanece como 5. Note também que dado que o endereço do segundo parâmetro é computado no início de *fun*, qualquer mudança para a variável global *i* não tem efeito no endereço usado no final para retornar o valor de *list[i]*.

9.6 PARÂMETROS QUE SÃO SUBPROGRAMAS

Algumas situações ocorrem em programação que são mais convenientemente manipuladas se os nomes dos subprogramas puderem ser enviados como parâmetros para outros subprogramas. Um exemplo comum ocorre quando um subprograma deve representar alguma função matemática. Por exemplo, um subprograma que faz integração numérica estima a área em um gráfico de uma função ao avaliar a função em um número de pontos diferentes. Quando um desses subprogramas é escrito, ele deve ser usável para qualquer função; não deve ser necessário que ele seja reescrito para cada função que precise ser integrada. Logo, é natural que o nome de uma função de um programa que avalie a função matemática a ser integrada seja enviado ao subprograma integrador como um parâmetro.

Apesar de a ideia ser natural e aparentemente simples, os detalhes de como isso funciona podem ser confusos. Se apenas a transmissão do código do subprograma fosse necessária, ela poderia ser feita se passando apenas um ponteiro. Entretanto, surgem duas complicações.

Primeiro, existe a questão da verificação de tipos dos parâmetros das ativações do subprograma passado como um parâmetro. Em C e C++, as 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 dos parâmetros, tais parâmetros podem ser completamente verificados em relação aos seus tipos. O Fortran 95 tem um mecanismo para fornecer tipos de parâmetros para subprogramas passados como parâmetros, já que eles precisam ser verificados. Ada não permite que subprogramas sejam passados como parâmetros. A funcionalidade de passar

subprogramas como parâmetros é, em vez disso, fornecida pelos recursos de tipos genéricos de Ada, discutidos na Seção 9.8.

A segunda complicação com parâmetros que são subprogramas aparece apenas com linguagens que permitem subprogramas aninhados. A questão é qual ambiente de referenciamento deve ser usado para executar o subprograma passado.

As três escolhas são:

1. O ambiente da sentença de chamada que chama o subprograma passado (**vinculação rasa – shallow binding**).
2. O ambiente da definição do subprograma passado (**vinculação profunda – deep binding**).
3. O ambiente da sentença de chamada que passou o subprograma como um parâmetro real (**vinculação ad hoc**).

O seguinte programa de exemplo, escrito com a sintaxe de 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 ele é chamado em `sub4`. Para a vinculação rasa, o ambiente de referenciamento dessa execução é o de `sub4`, então a referência a `x` em `sub2` é vinculada à variável local `x` em `sub4`, e a saída do programa é 4. Para a vinculação profunda, o ambiente de referenciamento da execução de `sub2` é o de `sub1`, então a referência a `x` em `sub2` é vinculada à variável local `x` em `sub1`, e a saída é 1. Para a vinculação *ad hoc*, a vinculação é para a variável local `x` em `sub3`, e a saída é 3.

Em alguns casos, o subprograma que declara um subprograma também o passa como um parâmetro. Nesses casos, a vinculação profunda e a vinculação *ad hoc* são a mesma. A vinculação *ad hoc* nunca foi usada, como pode-

NOTA HISTÓRICA

A definição original de Pascal (Jensen e Wirth, 1974) permitia que subprogramas fossem passados como parâmetros sem incluir a informação de tipos de seus parâmetros. Se a compilação independente fosse possível (que não era no Pascal original), nem seria permitido ao compilador verificar o número correto de parâmetros. Na ausência de compilação independente, a verificação de consistência de parâmetros é possível, mas é uma tarefa muito complexa, e normalmente não é feita. O Fortran 77 sofria do mesmo problema, mas como a consistência de tipos no Fortran 77 nunca era verificada, isso não era um problema adicional.

mos suspeitar, o ambiente no qual o procedimento aparece como um parâmetro não tem uma conexão natural com o subprograma passado.

A vinculação rasa não é apropriada para linguagens com escopo estático com subprogramas aninhados. Por exemplo, suponha que o procedimento `Sender` passasse o procedimento `Sent` como um parâmetro para o procedimento `Receiver`. O problema é que `Receiver` pode não estar no ambiente estático de `Sent`, tornando muito não natural para `Sent` ter acesso às variáveis de `Receiver`. Por outro lado, é perfeitamente normal em tal linguagem para qualquer subprograma, incluindo um enviado com um parâmetro, ter seu ambiente de referenciamento determinado pela posição léxica de sua definição. Logo, é mais lógico para essas linguagens usarem vinculação profunda. Algumas linguagens com escopo dinâmico usam a vinculação rasa.

9.7 SUBPROGRAMAS SOBRECARREGADOS

Um operador sobre carregado é um que tem múltiplos significados. O significado de uma instância em particular de um operador sobre carregado é determinado pelos tipos de seus operandos. Por exemplo, se o operador `*` tem dois operandos de ponto flutuante em um programa Java, ele especifica uma multiplicação de ponto flutuante. Mas se o mesmo operador tem dois operandos inteiros, ele especifica multiplicação de inteiros.

Um **subprograma sobre carregado** é um que tem o mesmo nome de outro subprograma no mesmo ambiente de referenciamento. Cada versão de um subprograma sobre carregado deve ter um protocolo único; ou seja, deve ser diferente das outras no número, ordem ou tipos de seus parâmetros, ou em seu tipo de retorno, se for uma função. O significado de uma chamada para um subprograma sobre carregado é determinado pela lista de parâmetros reais (e/ou possivelmente o tipo do valor retornado, no caso de uma função). Apesar de não ser necessário, os subprogramas sobre carregados normalmente implementam o mesmo processo.

C++, Java, Ada e C# incluem subprogramas sobre carregados pré-definidos. Por exemplo, muitas classes em C++, Java e C# têm construtores sobre carregados. Como cada versão de um subprograma sobre carregado tem um único perfil de parâmetro, o compilador pode desambiguar as ocorrências de chamadas a eles por meio de diferentes parâmetros de tipo. Infelizmente, não é tão simples. Coerções de parâmetros, quando permitidas, complicam o processo de desambiguação enormemente. Colocando de uma maneira simples, a questão é que, se nenhum perfil de parâmetro do método casa com o número e com os tipos dos parâmetros reais em uma chamada a método, mas dois ou mais métodos têm perfis de parâmetros que podem ser casados por coerções, qual método deve

ser chamado? Para um projetista de linguagem responder essa questão, ele deve decidir como ordenar os diferentes tipos de coerções, de forma que o compilador possa escolher o método que melhor casa com a chamada. Isso pode ser uma tarefa extremamente complexa. Para ver o alto nível de complexidade dessa situação, sugerimos que o leitor procure pelas regras de desambiguação de chamadas a métodos usadas em C++ (Stroustrup, 1997).

Em Ada, o tipo de retorno e uma função sobre carregada podem ser usados para desambiguar chamadas. Logo, duas funções sobre carregadas podem ter o mesmo perfil de parâmetros e diferirem apenas em seus tipos de retorno. Isso funciona porque Ada não permite expressões de modo misto, então o contexto de uma chamada a função pode especificar o tipo que é retornado da função. Por exemplo, se um programa Ada tiver duas funções chamadas `Fun`, ambas recebendo um parâmetro do tipo `Integer`, onde uma retorna um `Integer` e a outra retorna um `Float`, a seguinte chamada seria permitida:

```
A, B : Integer;
...
A := B + Fun(7);
```

Nesse código, a chamada a `Fun` é vinculada à versão de `Fun` que retorna um `Integer`, porque escolher a versão que retorna um `Float` causaria um erro de tipo.

Como C++, Java e C# permitem expressões de modo misto, o tipo de retorno é irrelevante para a desambiguação de funções sobre carregadas (ou métodos). O contexto da chamada não permite a determinação do tipo de retorno. Por exemplo, se um programa C++ tem duas funções chamadas `fun` e ambas recebem um `int` como parâmetro, mas uma retorna um `int` e a outra retorna um `float`, o programa não compilaria, porque o compilador não poderia determinar que versão de `fun` deveria ser usada.

Também é permitido aos usuários escrever múltiplas versões de subprogramas com o mesmo nome em Ada, Java, C++ e C#. Mais uma vez, em C++, Java e C#, os métodos sobre carregados mais comuns definidos pelo usuário são os construtores.

Os subprogramas sobre carregados com parâmetros padrão podem levar a chamadas a subprogramas ambíguas. Por exemplo, considere o seguinte código em C++:

```
void fun(float b = 0.0);
void fun();
...
fun();
```

A chamada é ambígua e causará um erro de compilação.

9.8 SUBPROGRAMAS GENÉRICOS

O reúso de software pode ter uma contribuição importante para sua produtividade. Uma maneira de aumentar a reusabilidade de software é diminuir

a necessidade de criar diferentes subprogramas que implementam o mesmo algoritmo em diferentes tipos de dados. Por exemplo, um programador não deveria precisar escrever quatro diferentes programas de ordenação para quatro matrizes que diferem apenas no tipo do elemento.

Um subprograma **polimórfico** recebe parâmetros de vários tipos em diferentes ativações. Subprogramas sobre carregados fornecem um tipo particular de polimorfismo chamado de **polimorfismo ad hoc**. Subprogramas sobre carregados não precisam se comportar de maneira similar.

Um tipo mais geral de polimorfismo é fornecido pelos métodos de Python e de Ruby. Lembre-se de que as variáveis nessas linguagens não têm tipos, então os parâmetros formais não têm tipos. Logo, 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. Instanciações diferentes de tais subprogramas podem receber parâmetros genéricos diferentes, produzindo subprogramas que recebem diferentes tipos de parâmetros. Todas as definições parametrizadas de subprogramas se comportam da mesma forma. Subprogramas parametricamente polimórficos são chamados de subprogramas **genéricos**. Ada, C++, Java 5.0 e C# 2005 fornecem um tipo de polimorfismo paramétrico em tempo de compilação.

9.8.1 Subprogramas genéricos em Ada

Ada fornece polimorfismo paramétrico por meio de uma construção que suporta a construção de múltiplas versões de unidades de programas para aceitar parâmetros de diferentes tipos de dados. As diferentes versões do subprograma são instanciadas, ou construídas, pelo compilador em resposta ao programa do usuário. Como as versões do subprograma têm o mesmo nome, há a ilusão de que um único subprograma pode processar dados de diferentes tipos em diferentes chamadas. Como unidades de programa desse tipo são genéricas por natureza, são às vezes chamadas de **unidades genéricas**.

O mesmo mecanismo pode ser usado para permitir execuções diferentes de um subprograma para chamar instanciações de um subprograma genérico. Isso é útil para fornecer a funcionalidade de subprogramas passados como parâmetros.

O seguinte exemplo ilustra um procedimento com três parâmetros genéricos, permitindo que o subprograma receba como parâmetro uma matriz genérica. É um procedimento de ordenação por troca projetado para funcionar com qualquer matriz com elementos de tipos numéricos elementares, usando qualquer faixa de índices de tipos ordinais:

```
generic
    type Index_Type is (<>);
    type Element_Type is private;
    type Vector is array (Integer range <>) of
        Element_Type;
```

```

procedure Generic_Sort(List : in out Vector);
procedure Generic_Sort(List : in out Vector) is
    Temp : Element_Type;
begin
    for Top in List'First..Index_Type'Pred(List'Last) loop
        for Bottom in Index_Type'Succ(Top)..List'Last loop
            if List(Top) > List(Bottom) then
                Temp := List(Top);
                List(Top) := List(Bottom);
                List(Bottom) := Temp;
            end if;
        end loop; - for Bottom ...
    end loop; - for Top ...
end Generic_Sort;

```

Partes desse procedimento genérico podem aparecer estranhas se você não está familiarizado com Ada. Entretanto, não é importante entender todos os detalhes da sintaxe. O tipo matriz e o tipo de seus elementos são os dois parâmetros genéricos desse procedimento. A matriz é declarada com qualquer tipo de índice (ou seja, qualquer tipo que seja permitido como um índice) com qualquer faixa.

Essa ordenação genérica nada mais é que um *template* para um procedimento; nenhum código é gerado para ele pelo compilador, e não tem efeitos em um programa, ao menos que seja instanciado para algum tipo. A instanciação é realizada com uma sentença de declaração como a seguinte:

```

procedure Integer_Sort is new Generic_Sort (
    Index_Type => Integer,
    Element_Type => Integer,
    Vector => Int_Array);

```

O compilador reage a essa sentença construindo uma versão de `Generic_Sort` chamada `Integer_Sort` que ordena matrizes do tipo `Int_Array` com elementos do tipo `Integer` e faixas de índices do tipo `Integer`.

`Generic_Sort`, do modo que foi escrita, assume que o operador `>` é definido para os elementos da matriz a ser ordenada. A genericidade de `Generic_Sort` pode ser aumentada com a inclusão de uma função de comparação entre seus parâmetros genéricos.

Lembre-se de que Ada não permite que subprogramas sejam passados como parâmetros para outros subprogramas. Para fornecer essa funcionalidade, Ada usa subprogramas formais genéricos. Em uma linguagem como Fortran, os subprogramas são passados como parâmetros, de forma que uma chamada em particular de um subprograma pode executar usando o subprograma específico passado para calcular seu resultado. Em Ada, o mesmo resultado é atingido ao permitir que o usuário instancie um subprograma genérico qualquer número de vezes, cada uma podendo ser usado um subprograma diferente. Por exemplo, considere o procedimento genérico em Ada:

```

generic
    with function Fun(X : Float) return Float;

```

```
procedure Integrate (Lowerbd : in Float;
                     Upperbd : in Float;
                     Result : out Float);
procedure Integrate (Lowerbd : in Float;
                     Upperbd : in Float;
                     Result : out Float) is
    Funval : Float;
begin
    ...
    Funval := Fun(Lowerbd);
    ...
end Integrate;
```

Esse procedimento de código pode ser instanciado para uma função definida pelo usuário `Fun1` com

```
procedure Integrate_Fun1 is new Integrate(Fun => Fun1);
```

Agora, `Integrate_Fun1` é um procedimento para integrar a função `Fun1`.

9.8.2 Funções genéricas em C++

Funções genéricas em C++ têm o nome descritivo de *funções template*. A definição de uma função *template* tem a forma geral

```
template <parâmetros de template>
```

- uma definição de função que pode incluir os parâmetros de template.

Um parâmetro de *template* (deve existir ao menos um) tem uma das seguintes formas

```
class identificador  
typename identificador
```

A forma `class` é usada para nomes de tipos. A forma `typename` é usada para passar um valor para a função *template*. Por exemplo, algumas vezes é conveniente passar um valor inteiro para o tamanho de uma matriz na função *template*.

Um *template* pode receber outro – na prática normalmente uma classe *template* que define um tipo genérico definido pelo usuário, como um parâmetro, mas não consideramos essa opção aqui⁶.

Como um exemplo de uma função *template*, considere o código:

```
template <class Type>  
Type max(Type first, Type second) {  
    return first > second ? first : second;  
}
```

onde `Type` é o parâmetro que especifica o tipo de dados no qual a função operará. Essa função *template* pode ser instanciada para qualquer tipo para o qual

⁶ Classes *template* são discutidas no Capítulo 11.

o operador `>` esteja definido. Por exemplo, se fosse instanciada com `int` como o parâmetro, ela seria

```
int max(int first, int second) {
    return first > second ? first : second;
}
```

Apesar de esse processo poder ser definido com uma macro, esta teria a desvantagem de não operar 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 no sentido de que funciona para qualquer tipo numérico. Entretanto, nem sempre funciona corretamente se chamada com um parâmetro que tem um efeito colateral, como

```
max(x++, y)
```

que produz

```
((x++) > (y) ? (x++) : (y))
```

Sempre que o valor `x` for maior do que o de `y`, `x` será incrementado duas vezes. Funções *template* em C++ são instanciadas implicitamente, quando a função é nomeada em uma chamada ou quando seu endereço é obtido com o operador `&`. Por exemplo, a função *template* de exemplo definida seria instanciada duas vezes pelo seguinte segmento de código – uma vez para os parâmetros de tipo `int` e outra para os parâmetros de tipo `char`:

```
int a, b, c;
char d, e, f;
...
c = max(a, b);
f = max(d, e);
```

A seguir, temos a versão C++ do subprograma de ordenação genérico dado na Seção 9.8.1. Ela é de certa forma diferente, porque os índices de matrizes em C++ são restritos ao tipo inteiro, com o limite inferior fixado em zero.

```
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] ...
```

```
}
```

Uma instanciação de exemplo dessa função *template* é

```
float flt_list[100];  
...  
generic_sort(flt_list, 100);
```

Os subprogramas genéricos de Ada e as funções *template* de C++ são como um “primo pobre” de um subprograma no qual os tipos dos parâmetros formais são dinamicamente vinculados aos tipos dos parâmetros reais em uma chamada. Nesse caso, apenas uma cópia do código é necessária, enquanto com as abordagens de Ada e C++, uma cópia precisa ser criada em tempo de compilação para cada tipo diferente necessário e a vinculação das chamadas a subprogramas àqueles chamados é estática.

9.8.3 Métodos genéricos em Java 5.0

Supporte para tipos e métodos genéricos foi adicionado à linguagem Java na versão 5.0. O nome de uma classe genérica em Java 5.0 é especificado por um nome seguido de uma ou mais variáveis de tipos delimitadas por sinais de menor que e de maior que (<>). Por exemplo,

```
GenType<T>
```

onde T é a variável de tipo. Tipos genéricos são discutidos em mais detalhes no Capítulo 11.

Os métodos genéricos de Java se diferem dos subprogramas genéricos de Ada e de C++ de diversas maneiras. Primeiro, os parâmetros genéricos devem ser classes – eles não podem ser tipos primitivos. Esse requisito desabilita um método genérico que mimetiza nossos exemplos em Ada e C++, nos quais os tipos dos componentes das matrizes são genéricos e podem ser tipos primitivos. Em Java, os componentes da matriz (de maneira oposta aos contêineres) não podem ser genéricos. Segundo, apesar de os métodos genéricos de Java poderem ser instanciados qualquer número de vezes, apenas uma cópia do código é construída. A versão interna de um método genérico, chamada de um método *cru*, opera em objetos da classe `Object`. No ponto onde o valor genérico de um método genérico é retornado, o compilador insere uma conversão explícita (*cast*) para o tipo apropriado. Terceiro, em Java, restrições podem ser especificadas em relação à faixa de classes que podem ser passadas para o método genérico como parâmetros genéricos. Tais restrições são chamadas de **limites**.

Como um exemplo de um método genérico em Java 5.0, considere o esqueleto de definição de método:

```
public static <T> T doIt(T[] list) {  
    ...  
}
```

Isso define um método chamado `doIt` que recebe uma matriz de elementos de um tipo genérico. O nome do tipo genérico é `T` e ele deve ser uma matriz. A seguir, está uma chamada de exemplo a `doIt`:

```
doIt<String>(myList) ;
```

Agora, considere a seguinte versão de `doIt`, que tem um limite em seu parâmetro genérico:

```
public static <T extends Comparable> T doIt(T[] list) {  
    ...  
}
```

Isso define um método que recebe um parâmetro do tipo matriz genérico cujos elementos são de uma classe que implementa a interface `Comparable`. Essa é a restrição, ou limite, no parâmetro genérico. A palavra reservada `extends` parece implicar que a classe genérica é uma subclasse da classe seguinte. Nesse contexto, entretanto, `extends` tem um significado diferente. A expressão `<T extends BoundingType>` especifica que `T` deve ser um subtipo do tipo limitante. Então, `extends` nesse contexto significa que a classe genérica (ou interface) ou deve estender a classe limitadora (o limite se for uma classe) ou implementar a interface limitadora (se o limite é uma interface). O limite garante que os elementos de qualquer instanciação do tipo genérico pode ser comparado com o método `compareTo`, da interface `Comparable`.

Se um método genérico tiver duas ou mais restrições em seu tipo genérico, elas são adicionadas à cláusula `extends`, separadas por e comerciais (&). Além disso, métodos genéricos podem ter mais de um parâmetro genérico.

Java 5.0 suporta *tipos coringa* – por exemplo, `Collection<?>` é um tipo coringa 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:

```
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 coringa. Por exemplo, como os componentes de um objeto em particular desse tipo têm um tipo, objetos de outros não podem ser adicionados à coleção. Por exemplo, considere:

```
Collection<?> c = new ArrayList<String>();
```

Seria ilegal usar o método `add` para colocar algo nessa coleção, a menos que seu tipo fosse `String`.

Tipos coringa podem ser restritos (como no caso dos tipos não coringa), sendo chamados de *tipos coringa limitados*. Por exemplo, considere o seguinte cabeçalho de método:

```
public void drawAll(ArrayList<? extends Shape> things)
```

O tipo genérico aqui é um tipo coringa que é uma subclasse da classe `Shape`. Esse método pode ser escrito para desenhar qualquer objeto cujo tipo é uma subclasse de `Shape`.

9.8.4 Métodos genéricos em C# 2005

Os métodos genéricos do C# 2005 são similares em termos de recursos àqueles de Java 5.0, exceto que não há suporte para tipos coringa. Um recurso único dos métodos genéricos do C# 2005 é que os parâmetros de tipo reais em uma chamada podem ser omitidos se o compilador puder inferir o tipo não especificado. Por exemplo, considere o seguinte esqueleto de definição de classe:

```
class MyClass {  
    public static T DoIt<T>(T p1) {  
        ...  
    }  
}
```

O método `DoIt` pode ser chamado sem especificar o parâmetro genérico se o compilador puder inferir o tipo genérico do parâmetro real na chamada. Por exemplo, ambas as chamadas a seguir são permitidas:

```
int myInt = MyClass.DoIt(17); // Chama DoIt<int>  
string myStr = MyClass.DoIt('apples'); // Chama  
DoIt<string>
```

9.9 QUESTÕES DE PROJETO PARA FUNÇÕES

As três questões de projeto a seguir são específicas às funções:

- Os efeitos colaterais são permitidos?
- Que tipos de valores podem ser retornados?
- Quantos valores podem ser retornados?

9.9.1 Efeitos colaterais funcionais

Devido aos problemas dos efeitos colaterais de funções chamadas em expressões, conforme descrito no Capítulo 5, os parâmetros para funções devem ser sempre parâmetros no modo de entrada. Algumas linguagens requerem isto; por exemplo, as funções em Ada podem ter somente parâmetros formais do modo de entrada. Esse requisito efetivamente previne que uma

função cause efeitos colaterais por meio de seus parâmetros ou de apelidos de parâmetros e de variáveis globais. Na maioria das outras linguagens, entretanto, as funções podem ter tanto parâmetros com passagem por valor quanto por passagem por referência, permitindo funções que causem efeitos colaterais e apelidos.

9.9.2 Tipos dos valores retornados

A maioria das linguagens de programação imperativas restringe os tipos que podem ser retornados por suas funções. C permite que quaisquer tipos sejam retornados por suas funções, exceto matrizes e funções. Ambos podem ser manipulados por valores de retorno do tipo ponteiro. C++ é como C, mas também permite que tipos definidos pelo usuário, ou classes, sejam retornados a partir de suas funções. Ada, Python, Ruby e Lua são as linguagens dentre as imperativas cujas funções (e/ou métodos) podem retornar valores de qualquer tipo. No caso de Ada, entretanto, dado que as funções não são tipos em Ada, elas não podem ser retornadas a partir de funções. É claro, ponteiros para funções podem ser retornados por funções.

Em algumas linguagens de programação, subprogramas são objetos de primeira-classe, ou seja, eles podem ser tratados de forma bastante similar a objetos de dados. Em JavaScript, por exemplo, funções podem ser passadas como parâmetros e retornadas a partir de funções. Em Python, Ruby e Lua, os métodos são objetos que podem ser tratados como qualquer outro objeto. O mesmo é verdade para muitas linguagens funcionais (veja o Capítulo 15). Nem Java nem C# têm funções, apesar de seus métodos serem similares às funções. Em ambas, qualquer tipo ou classe pode ser retornado pelos métodos. Como os métodos não são tipos, eles não podem ser retornados.

9.9.3 Número de valores retornados

Na maioria das linguagens, apenas um valor único pode ser retornado de uma função. Entretanto, isso nem sempre é o caso. Ruby permite o retorno de mais de um valor de um método. Se uma sentença `return` em um método Ruby não for seguida por uma expressão, `nil` é retornado. Se for seguida por uma expressão, o valor da expressão é retornado. Se seguido por mais de uma expressão, uma matriz de valores de todas as expressões é retornada.

Lua também permite que as funções retornem múltiplos valores, que seguem a sentença `return` como uma lista separada por vírgulas, como:

```
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, ou seja, como uma sentença, todos os valores de retorno são ignorados. Se a função retornou três valores e todos eles serão armazenados pelo chamador, a função seria chamada como no seguinte exemplo:

```
a, b, c = fun()
```

9.10 OPERADORES SOBRECARREGADOS DEFINIDOS PELO USUÁRIO

Os operadores podem ser sobrecarregados pelo usuário em Ada, C++, Python e Ruby. Como um exemplo disso, considere a seguinte função em Ada que sobrecarrega o operador de multiplicação (*) para calcular o produto escalar de duas matrizes. O produto escalar de duas matrizes de tamanho igual é a soma dos produtos de cada um dos pares de elementos correspondentes das duas matrizes. Suponha que `Vector_Type` tenha sido definido como um tipo matriz com elementos inteiros (`Integer`):

```
function "*" (A, B : in Vector_Type) return Integer is
    Sum : Integer := 0;
begin
    for Index in A'range loop
        Sum := Sum + A(Index) * B(Index);
    end loop; -- for Index ...
    return Sum;
end "*";
```

O produto escalar, como especificado nessa definição de função, é computado sempre que o asterisco aparece com dois operandos do tipo `Vector_Type`. O asterisco pode ser sobrecarregado qualquer número de vezes, desde que as funções definidoras tenham protocolos únicos.

A função de produto escalar de exemplo poderia também ser escrita em C++. O protótipo de tal função poderia ser:

```
int operator * (const vector &a, const vector &b, int len);
```

9.11 CORROTINAS

NOTA HISTÓRICA

A origem real do conceito de controle de unidade simétrico é difícil de determinar. Uma das primeiras aplicações de corrotinas publicadas foi na área de análise sintática (Conway, 1963). A primeira linguagem de alto nível a incluir recursos para corrotinas foi o SIMULA 67. Lembre-se de que o objetivo original de SIMULA era a simulação de sistemas, que geralmente requer a modelagem de processos independentes. Essa necessidade foi a motivação para o desenvolvimento das corrotinas em SIMULA.

Uma **corrotina** é um tipo especial de subprograma. Em vez do relacionamento mestre-escravo entre um subprograma chamador e um chamado que existe com os subprogramas convencionais, as corrotinas chamadora e chamada estão em um relacionamento mais igualitário. Na verdade, o mecanismo de controle das corrotinas é normalmente chamado de modelo de controle de unidades simétrico.

Corrotinas podem ter múltiplos pontos de entrada, controlados pelas próprias corrotinas. Elas também possuem os meios de manter seu estado entre ativações. Isso significa que as corrotinas devem ser sensíveis ao histórico e logo possuem variáveis locais estáticas. Execuções secundárias de uma corrotina em geral começam em pontos que não são seu início. Por causa disso, a invocação de uma corrotina é chamada de uma **continuação** em vez de uma chamada.

Por exemplo, considere o seguinte esqueleto de uma corrotina:

```
sub co1 () {
    ...
    resume co2 () ;
    ...
    resume co3 () ;
    ...
}
```

Na primeira vez em que `co1` é continuada, sua execução começa na primeira sentença e executa para baixo, incluindo a continuação de `co2`, que transfere o controle para `co2`. Na próxima vez em que `co1` for continuada, sua execução começa na primeira sentença após sua chamada a `co2`. Quando `co1` for continuada pela terceira vez, sua execução começa na primeira sentença após a continuação de `co2`.

Uma das características usuais de subprogramas é mantida nas corrotinas: apenas uma é executada em um dado momento no tempo.

Como vimos no exemplo acima, em vez de executarem até o fim, as corrotinas em geral executam parcialmente e então transferem o controle para alguma outra, e quando reiniciadas, uma corrotina continua a execução logo após a sentença que usou para transferir o controle para outro local. Esse tipo de sequência de execução intercalada é relacionada com a maneira pela qual os sistemas operacionais com multiprogramação funcionam. Apesar de poder existir apenas um processador, todos os programas executando em tal sistema parecem rodar concorrentemente enquanto compartilham o processador. No caso das corrotinas, isso é algumas vezes chamado de **quasi-concorrência**.

Normalmente, corrotinas são criadas em uma aplicação por uma unidade de programa chamada de unidade principal, que não é uma corrotina. Quando criadas, as corrotinas executam seu código de inicialização e retornam o controle para a unidade principal. Quando todas as corrotinas de uma família são construídas, o programa principal continua uma das corrotinas, e os membros da família de corrotinas continuam umas as outras em alguma ordem até o trabalho estar completo, se é que de fato podem ser terminadas. Se a execução de uma corrotina chega ao fim de sua seção de código, o controle é transferido para a unidade principal que a criou. Esse é o mecanismo para terminar a execução de coleções de corrotinas, quando isso é desejável. Em alguns programas, as corrotinas rodam sempre que o computador estiver rodando.

Um exemplo de um 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 que usam a mesma estratégia. Tal jogo pode ser simulado por meio de uma unidade de programa principal que cria uma família de corrotinas, cada uma com uma coleção, ou mão, de cartas. O programa principal poderia iniciar a simulação pela continuação da corrotina de um dos jogadores, que após ter jogado em seu turno, pudesse continuar a corrotina do próximo jogador, e assim por diante até o término do jogo.

A mesma forma de sentença de continuação pode ser usada tanto para iniciar quanto para reiniciar a execução de uma corrotina.

Suponha que as unidades de programa A e B são corrotinas. A Figura 9.3 mostra duas maneiras pelas quais uma sequência de execução envolvendo A e B poderiam proceder.

Na Figura 9.3a, a execução da corrotina A é iniciada pela unidade principal. Após alguma execução, A inicia B. Quando a corrotina B na figura 9.3 causa pela primeira vez o controle para retornar a corrotina A, a semântica é que A continua a partir do ponto de onde ela terminou em sua última execução. Em particular, suas variáveis locais têm os valores deixados a elas pela ativação prévia. A Figura 9.3b mostra uma sequência de execução alternativa de corrotinas A e B. Nesse caso, B é iniciada pela unidade principal.

Em vez de ter os padrões mostrados na Figura 9.3, uma corrotina em geral tem um laço contendo uma continuação. A Figura 9.4 mostra a sequência de execução desse cenário. Nesse caso, A é iniciada pela unidade principal.

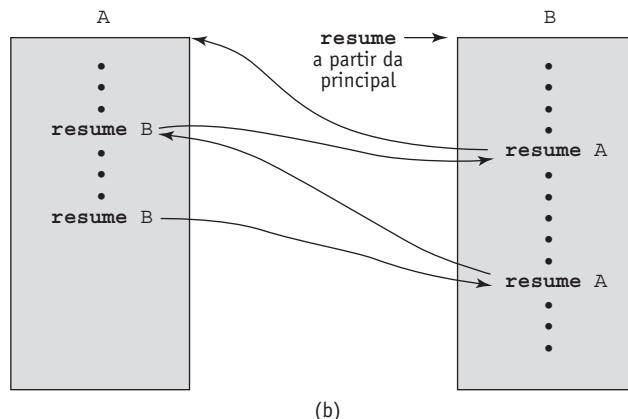
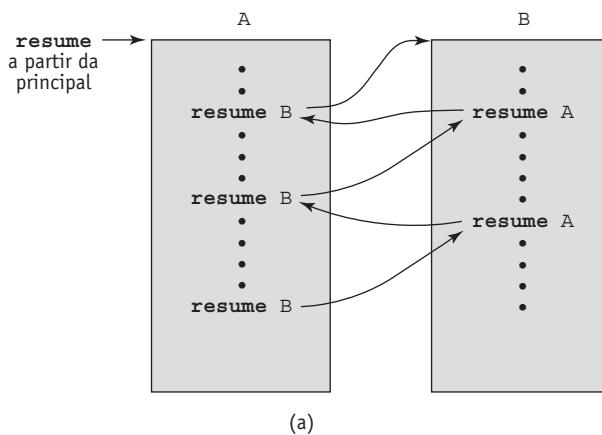


Figura 9.3 Duas sequências de controle de execução possíveis para duas corrotinas sem laços.

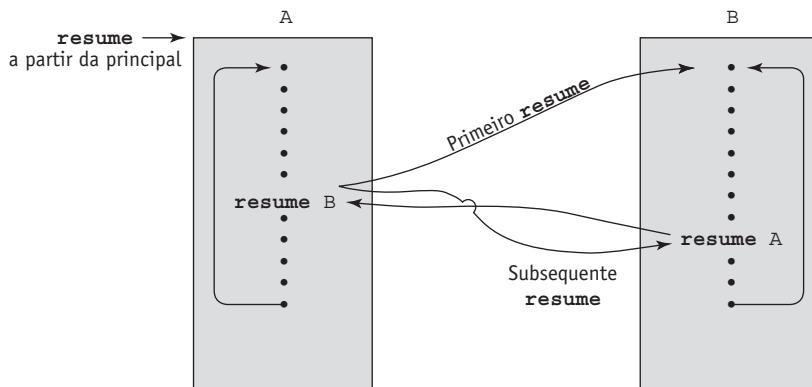


Figura 9.4 Sequência de execução de corrotinas com laços.

Dentro de seu laço principal, A inicia B, o que por sua vez continua A em seu laço principal.

Dentre as linguagens contemporâneas, apenas Lua suporta corrotinas.

RESUMO

Abstrações de processos são representadas em linguagens de programação por subprogramas. Uma definição de subprograma descreve as ações representadas por ele. Uma chamada a um subprograma realiza essas ações. Um cabeçalho de subprograma identifica sua definição e fornece sua interface, denominada seu protocolo.

Os parâmetros formais são os nomes que os subprogramas usam para se referirem aos parâmetros reais dados nas chamadas a subprogramas. Em Python e Ruby, os parâmetros dos tipos matriz e resumo são usados para suportar números variáveis de parâmetros. Lua e JavaScript também suportam um número variável. Parâmetros reais podem ser associados com os formais por sua posição ou por palavras-chave.

Ruby permite que blocos sejam anexados a chamadas a métodos. Tais blocos podem ter parâmetros. Eles são chamados com uma sentença `yield` no método chamado.

Subprogramas podem ser funções, que modelam funções matemáticas e são usadas para definir novas operações, ou podem ser procedimentos, que definem novas sentenças.

Variáveis locais em subprogramas podem ser dinâmicas da pilha, fornecendo suporte para recursão, ou estáticas, fornecendo eficiência e variáveis locais sensíveis ao histórico. JavaScript, Python, Ruby e Lua permitem que as definições de subprogramas sejam aninhadas.

Existem três modelos fundamentais de passagem de parâmetros – modo de entrada, modo de saída e modo de entrada e saída – e um número de abordagens para implementá-los. São elas as passagens por valor, por resultado, por valor-resultado, por referência e por nome. Na maioria das linguagens, os parâmetros são passados na pilha em tempo de execução e seus tipos são verificados.

Apelidos podem ocorrer quando parâmetros com passagem por referência são usados, tanto entre dois ou mais parâmetros quanto entre um parâmetro e uma variável não local acessível.

Parâmetros que são matrizes multidimensionais levantam algumas questões para o projetista de linguagens, porque o subprograma passado precisa saber como calcular a função de mapeamento de armazenamento para elas. Isso requer mais do que apenas o nome da matriz.

Parâmetros que são nomes de subprogramas fornecem um serviço necessário, mas são às vezes difíceis de serem entendidos. A transparência reside no ambiente de referenciamento que está disponível quando um subprograma que foi passado como parâmetro está executando.

Ada, C++, C#, Ruby e Python permitem tanto a sobrecarga de subprogramas quanto de operadores. Subprogramas podem ser sobre carregados desde que as várias versões possam ser desambiguadas pelos tipos de seus parâmetros ou de seus valores de retorno. Definições de funções podem ser usadas para construir significados adicionais para operadores.

Subprogramas em Ada, C++, Java 5.0 e C# 2005 podem ser genéricos, usando polimorfismo paramétrico, de forma que os tipos desejados de seus objetos de dados possam ser passados para o compilador, que pode então construir unidades para os tipos requisitados.

O projetista de um recurso de funções em uma linguagem deve decidir que restrições serão colocadas nos valores retornados, assim como o número de valores de retorno.

Uma corrotina é um subprograma especial que tem múltiplas entradas. Elas podem ser usadas 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 é informado no cabeçalho de um subprograma?
4. Que característica dos subprogramas em Python os separaram daqueles das outras linguagens?
5. Que linguagens permitem um número variável de parâmetros?
6. O que é um parâmetro formal de matriz em Ruby?
7. O que é um perfil de parâmetro? 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 de parâmetros com palavras-chave?
10. Qual é o uso mais comum de blocos Ruby na linguagem propriamente dita?
11. Quais são as diferenças entre uma função e um procedimento?
12. Quais são as questões de projeto para subprogramas?
13. Quais são as vantagens e desvantagens das variáveis locais dinâmicas?
14. Quais são as vantagens e desvantagens das variáveis locais estáticas?
15. Que linguagens permitem que as definições de subprogramas sejam aninhadas?
16. Quais são os três modelos semânticos de passagem de parâmetros?
17. Quais são os modos, os modelos conceituais de transferência, as vantagens e as desvantagens dos seguintes métodos de passagem de parâmetros: por valor, por resultado, por valor-resultado e por referência?
18. Descreva as maneiras pelas quais os apelidos podem ocorrer com parâmetros com passagem por referência.

19. Qual é a diferença entre a maneira pela qual o C original e o C89 tratam com um parâmetro real cujo tipo não é idêntico àquele do parâmetro formal correspondente?
20. Qual é o problema com a política de Ada de permitir que os implementadores decidam quais parâmetros passar como referência e quais passar por valor-resultado?
21. Quais são as duas considerações de projeto fundamentais para os métodos de passagem de parâmetros?
22. Descreva o problema de passar matrizes multidimensionais como parâmetros.
23. Qual é o nome do método de passagem de parâmetros usado em Python e Ruby?
24. Quais são as duas questões que surgem quando os nomes dos subprogramas são parâmetros?
25. Defina vinculação rasa e profunda para ambientes de referenciamento de subprogramas passados como parâmetros?
26. O que é um subprograma sobrecarregado?
27. O que é polimorfismo paramétrico?
28. O que faz com que uma função *template* de C++ seja instanciada?
29. De que maneiras fundamentais os parâmetros genéricos de métodos genéricos em Java 5.0 são diferentes daqueles dos métodos de C++?
30. Se um método de Java 5.0 retorna um tipo genérico, que tipo de objeto é na verdade retornado?
31. Se um método genérico de Java 5.0 é chamado com três parâmetros genéricos diferentes, quantas versões do método serão geradas pelo compilador?
32. Quais são as questões de projeto para funções?
33. Quais são as duas linguagens que permitem que múltiplos valores sejam retornados de uma função?
34. Que linguagens permitem que o usuário sobrecarregue operadores?
35. De que maneiras as corrotinas são diferentes dos subprogramas convencionais?

CONJUNTO DE PROBLEMAS

1. Quais são os argumentos a favor e contra um programa de usuário construir definições adicionais para operadores existentes, como pode ser feito em Ada e C++? Você acredita que tal sobrecarga de operadores definida pelo usuário é boa ou ruim? Justifique sua resposta.
2. Na maioria das implementações do Fortran IV, os parâmetros eram passados por referência, usando apenas a transmissão de um caminho de acesso. Diga tanto as vantagens quanto as desvantagens dessa escolha de projeto.
3. Argumente a favor da decisão dos projetistas de Ada 83 de permitirem que o implementador escolha entre implementar os parâmetros no modo de entrada e saída por meio de cópia ou de referência.
4. Suponha que você queira escrever um método que imprima um cabeçalho em uma nova página de saída, com um número de página que é 1 na primeira ativação e que aumente em 1 em cada ativação subsequente. Isso pode ser feito sem parâmetros e sem referências a variáveis não locais em Java? Isso pode ser feito em C#?

5. Considere o programa escrito na sintaxe de 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 métodos de passagem de parâmetros, quais são todos os valores das variáveis `value` e `list` após cada uma das três chamadas a `swap`?

- a. Passadas por valor
 - b. Passadas por referência
 - c. Passadas por valor-resultado
6. Apresente um argumento contra fornecer tanto variáveis locais estáticas quanto dinâmicas em subprogramas.
7. Considere o programa 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 métodos de passagem de parâmetros, quais são os valores da matriz `list` após a execução?

- a. Passadas por valor
 - b. Passadas por referência
 - c. Passadas por valor-resultado
8. Argumente contra o projeto de C de fornecer apenas subprogramas na forma de funções.
9. A partir de um livro texto de Fortran, aprenda a sintaxe e a semântica das sentenças relativas a funções. Justifique a sua existência no Fortran.
10. Estude os métodos de sobrecarga de operadores definida pelo usuário em C++ e Ada e escreva um relatório comparando as duas usando nossos critérios para avaliar linguagens.
11. C# suporta parâmetros no modo de saída, mas nem Java nem C++ o fazem. Dê uma explicação para essa diferença.
12. Pesquise acerca do dispositivo de Jensen (*Jensen Device*), uso bastante conhecido de parâmetros com passagem por nome, e escreva uma descrição curta sobre o que é e como pode ser usado.

13. Estude os mecanismos de iteração de Ruby e CLU e liste suas similaridades e diferenças.
14. Especule acerca da questão de permitir subprogramas aninhados em linguagens de programação – por que eles não são permitidos em muitas linguagens contemporâneas?
15. Cite ao menos dois argumentos contra o uso de parâmetros com passagem por nome.
16. Escreva uma comparação detalhada dos subprogramas genéricos de Java 5.0 e C# 2005.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Escreva um programa Fortran que determine se um compilador Fortran para o qual você tenha acesso implementa variáveis locais como estáticas ou dinâmicas da pilha. *Dica:* A maneira mais fácil de verificar é fazer com que seu programa teste a sensitividade ao histórico de um subprograma.
2. Escreva um programa em uma linguagem que você conheça para determinar a taxa de tempo necessária para passar uma grande matriz por referência e por valor. Faça com que a matriz seja o maior possível na máquina e na implementação que você usa. Passe a matriz tantas vezes quantas forem necessárias para obter tempos razoavelmente precisos das operações de passagem.
3. Escreva um programa em C# ou em Ada que determine quando o endereço de um parâmetro no modo de saída é computado (no momento da chamada ou no momento em que a execução do subprograma terminar).
4. Escreva um programa em Perl que passa por referência um literal para um subprograma, que tente modificar o parâmetro. Dada a filosofia de projeto geral de Perl, explique os resultados.
5. Repita o Exercício de Programação 4 em C#.
6. Escreva um programa em alguma linguagem que tenha tanto variáveis locais estáticas quanto dinâmicas da pilha em subprogramas. Crie seis grandes matrizes (ao menos 100 por 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 no subprograma deve realizar um grande número de operações de multiplicação de matrizes nas matrizes estáticas e cronometrar o processo. Então, ele deve repetir esse processo com as matrizes dinâmicas da pilha. Compare e explique os resultados.
7. Escreva um programa em C# que inclua dois métodos chamados muitas vezes. Para ambos os métodos, é passada uma grande matriz, para um por valor, para outro por referência. Compare os tempos necessários para chamar esses dois métodos e explique a diferença. Certifique-se de chamá-los um número suficiente de vezes para ilustrar a diferença no tempo necessário.
8. Escreva um programa em Ada que determine se é permitido chamar uma função passada por meio da passagem de um ponteiro a ela para outra função.
9. Escreva um programa, usando a sintaxe de qualquer linguagem que você goste, que produza um comportamento diferente dependendo se a passagem por referência ou a passagem por valor-resultado é usada em sua passagem de parâmetros.
10. Escreva uma função genérica em Ada que receba uma matriz de elementos genéricos e um escalar do mesmo tipo dos elementos da matriz. O tipo dos elementos da matriz e do escalar é o parâmetro genérico. Os índices da matriz

são inteiros positivos. A função deve buscar a matriz informada para o escalar informado e retornar o índice do escalar na matriz. Se o escalar não estiver na matriz, a função deve retornar -1. Instancie a função para os tipos `Integer` e `Float` e teste ambos.

11. Escreva uma função genérica em C++ que receba uma matriz de elementos genéricos e um escalar do mesmo tipo dos elementos da matriz. O tipo dos elementos da matriz e do escalar é o parâmetro genérico. Os índices da matriz são inteiros positivos. A função deve buscar a matriz informada para o escalar informado e retornar o índice do escalar na matriz. Se o escalar não estiver na matriz, a função deve retornar -1. Teste a função para os tipos `int` e `float`.
12. Crie um subprograma e o código de chamada no qual as passagens por referência e por valor-resultado de um ou mais parâmetros produzem resultados diferentes.

Capítulo 10

Implementando Subprogramas

10.1 A semântica geral de chamadas e retornos

10.2 Implementando subprogramas “simples”

10.3 Implementando subprogramas com variáveis locais dinâmicas da pilha

10.4 Subprogramas aninhados

10.5 Blocos

10.6 Implementando 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 o 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, por fim, para 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. Técnicas para a implementação de blocos são cobertas brevemente. Por fim, são discutidos vários métodos de implementação de acesso a variáveis não locais em uma linguagem de escopo dinâmico.

As Seções 10.1 até a 10.5 tratam exclusivamente de linguagens com escopo estático. A Seção 10.6 discute a implementação de subprogramas em linguagens de escopo dinâmico.

10.1 A SEMÂNTICA GERAL DE CHAMADAS E RETORNOS

As operações de chamada e retorno de subprogramas são juntas 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 fazer o armazenamento ser alocado 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 aquilo necessário para retomar a execução da unidade de programa chamadora, o que inclui valores de registro, os bits de estado da CPU e o ponteiro de ambiente (PE). O PE, discutido com mais detalhes na Seção 10.3.2, 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 a variáveis não locais visíveis para o subprograma chamado.

As ações necessárias a um retorno de subprograma também são complicadas. Se o subprograma tem parâmetros do modo de saída ou do modo de entrada e saída e 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 IMPLEMENTANDO SUBPROGRAMAS “SIMPLES”

Começamos com a tarefa de implementar subprogramas simples; por “simples” queremos dizer que os subprogramas não podem ser aninhados e que todas as variáveis locais são estáticas.

As primeiras versões do Fortran são exemplos de idiomas que tinham esse tipo de subprograma.

A semântica de uma chamada a um subprograma “simples” requer as seguintes ações:

1. Salvar 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 subprogramma simples requer as seguintes ações:

1. Se existirem parâmetros com passagem por valor-resultado ou parâmetros no modo de saída, os valores atuais deles são movidos para os parâmetros reais correspondentes.
2. Se o subprogramma 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 requerem 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.

Essas, com as variáveis locais e o código de subprogramma, formam a coleção completa de informações que um subprogramma precisa para executar e retornar o controle para o chamador.

A questão óbvia 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. Salvar o estado da execução do chamador pode ser feito por ambos. No caso do retorno, a primeira, a terceira e a quarta ações devem ser feitas 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, no início da execução ou no final, denominados também de prólogo e epílogo da ligação de subprogramas. 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 mudar 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 um registro de ativação, uma coleção de dados na forma de um 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 de 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 completamente pelo compilador. Na verdade, devido à 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 para um arquivo. O programa executável mostrado na Figura 10.2 é unido pelo **ligador**, que é parte do sistema operacional. (Algumas vezes, os ligadores são chamados de carre-

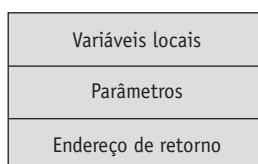


Figura 10.1 Um registro de ativação para subprogramas simples.

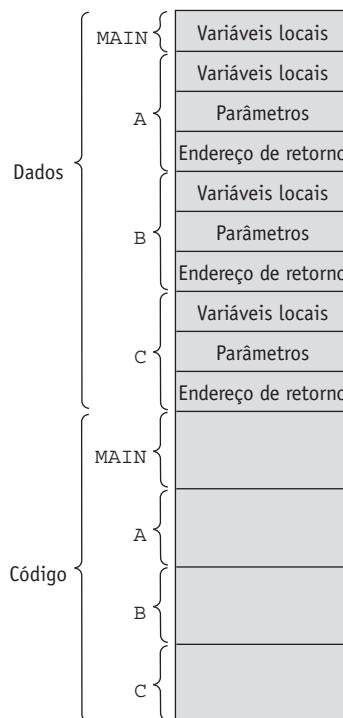


Figura 10.2 Um registro de ativação para subprogramas simples.

gadores, 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 vinculador deve configurar os endereços de destino de todas as chamadas aos subprogramas no programa principal para os endereços de entrada desses subprogramas. O mesmo deve ser feito para as chamadas de subprogramas nos subprogramas carregados e 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 em `A`, `B` e `MAIN`.

10.3 IMPLEMENTANDO 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 focando nas 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 da 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 deve 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 um dado momento, com ao menos uma chamada de fora do subprograma e uma ou mais chamadas recursivas. A recursão, então, requer múltiplas instâncias de registros de ativação, uma para cada ativação do subprograma que puder existir ao mesmo tempo. O número de ativações é limitado apenas pelo tamanho da memória da máquina. Cada uma delas requer sua instância de registro de ativação.

O formato de um registro de ativação para um subprograma na maioria das linguagens 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.

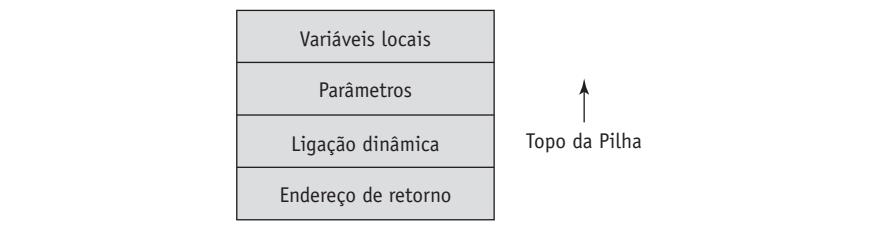


Figura 10.3 Um registro de ativação típico para uma linguagem com variáveis locais dinâmicas da pilha.

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 são às vezes 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, então aparecem por último.

Considere o esqueleto de função em C:

```
void sub(float total, int part) {
    int list[5];
    float sum;
    ...
}
```

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 o subprograma. Conforme mencionado anteriormente, o formato do registro de ativação é fixado em tempo de compilação, apesar de seu tamanho poder depender da chamada em algumas linguagens. Como a semântica de chamada e de retorno especifica que o último subpro-

Local	sum
Local	list [4]
Local	list [3]
Local	list [2]
Local	list [1]
Local	list [0]
Parâmetro	part
Parâmetro	total
Ligação dinâmica	
Endereço de retorno	

Figura 10.4 O registro de ativação para a função `sub`.

grama chamado é o primeiro a ser completado, é razoável criar instâncias desses registros de ativação em uma pilha. Essa pilha é parte do sistema de tempo de execução e logo 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 nova 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. Logo, 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 que está sendo executada. Quando um subprograma é chamado, o PE atual é salvo na nova instância de registro de ativação como a ligação dinâmica. O PE 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. Reiniciar o topo da pilha efetivamente remove a instância de registro de ativação do topo.

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 nas instâncias de registro de ativação como as ligações dinâmicas.

Esta seção adicionou diversas ações para o processo de ligação. A lista da Seção 10.2 deve ser revisada para levar essas ações em consideração. 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, os valores atuais desses parâmetros são movidos para os parâmetros reais correspondentes.

2. Se o subprograma é uma função, o valor funcional é movido 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 completa. No momento em que ele se tornar inativo, seu escopo local cessa sua existência e seu ambiente de referenciamento não é mais significativo. Então, 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 as máquinas RISC normalmente têm muito mais registradores do que máquinas CISC. No restante deste capítulo, entretanto, assumimos que os parâmetros são passados na pilha. É bem direta a modificação dessa abordagem para que os parâmetros sejam passados em registradores.

10.3.2 Um exemplo sem recursão

Considere o esqueleto de programa em C:

```
void fun1(float r) {
    int s, t;
    ...
    fun2(s);
    ...
}

void fun2(int x) {
    int y;
    ...
    fun3(y);
    ...
}

void fun3(int q) {
    ...
}

void main() {
    float p;
    ...
    fun1(p);
    ...
}
```

A sequência de chamadas a procedimentos 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 de 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. Um processo similar ocorre 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 instância de registro de ativação na pilha para funções main, como a mostrada na figura. Entretanto, pode ser feito dessa forma, e isso simplifica tanto a implementação quanto nossa discussão. Nesse exemplo e em todos os outros neste capítulo, assumimos que a pilha cresce do menor para o maior endereço, apesar de, em uma implementação em particular, a pilha pode crescer na direção oposta.

A coleção de ligações dinâmicas presentes na pilha em um dado momento é chamada de **cadeia dinâmica** ou de **cadeia de chamadas**. Ela representa a história dinâmica de como a execução chegou a sua posição atual, que está

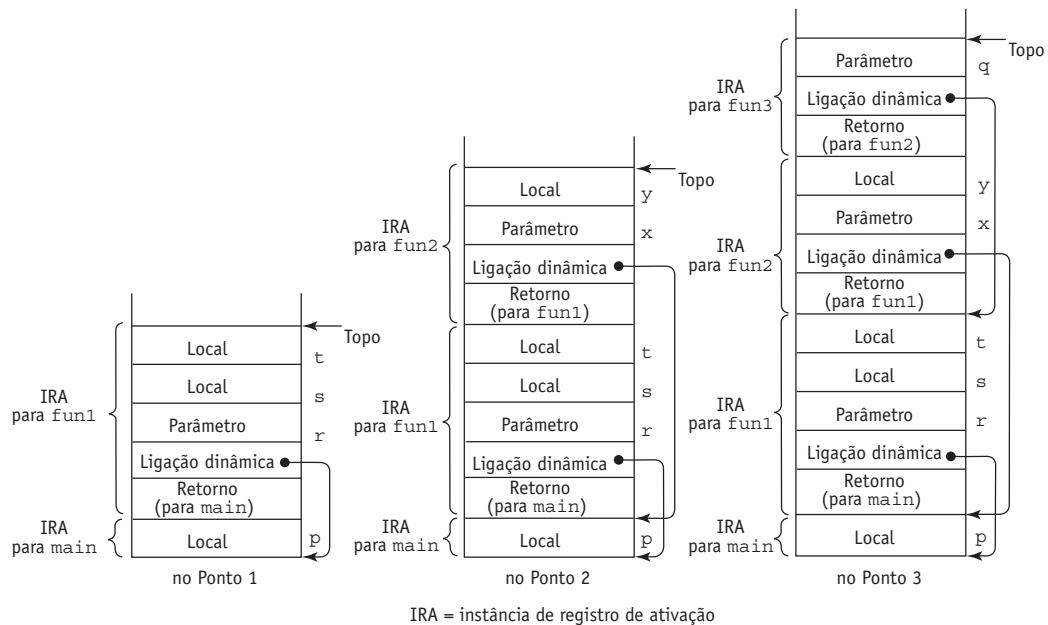


Figure 10.5 Conteúdo da pilha para três pontos em um programa.

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, assumimos 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 do 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 o exemplo de programa em C, que usa recursão para calcular a função factorial:

```
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 de 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.

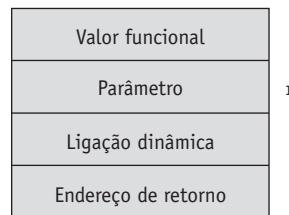


Figura 10.6 O registro de ativação para factorial.

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 ser executado, mas antes que o registro de ativação fosse 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

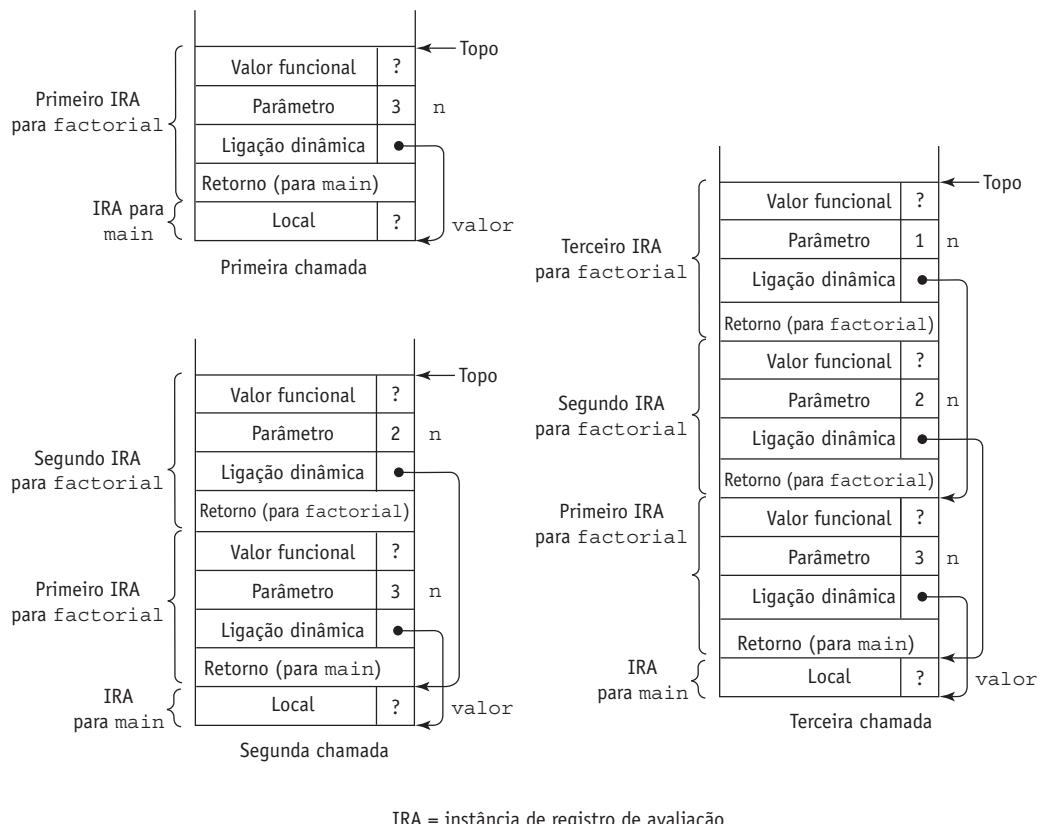


Figura 10.7 Conteúdo da pilha na posição 1 de factorial.

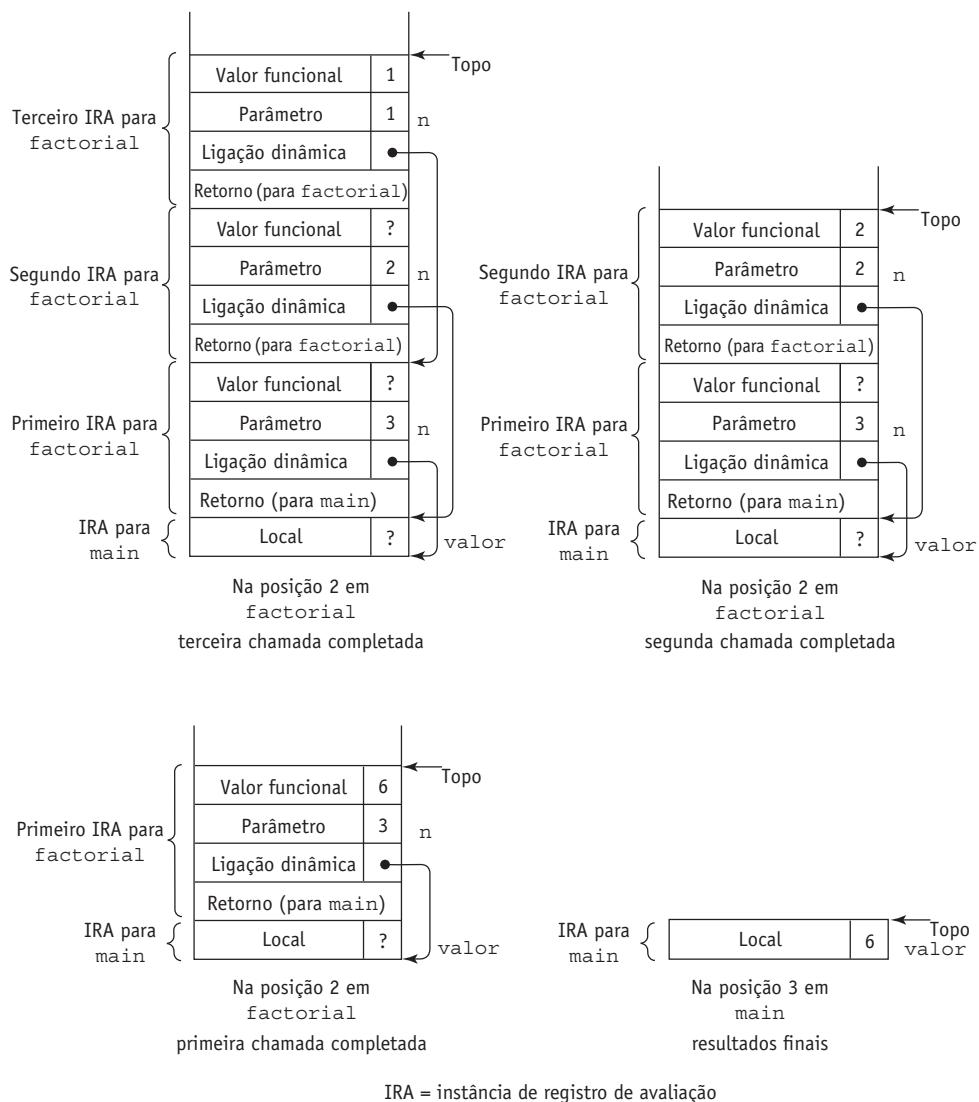


Figura 10.8 Conteúdo da pilha durante a execução de `main` e `factorial`.

a função. O primeiro retorno 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 de 6, que é então retornado para a primeira chamada a `factorial` em `main`.

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. Dentre elas, estão Fortran 95, Ada, Python, JavaScript e Lua. Nesta seção, examinamos a abordagem mais usada para implementar subprogramas que podem ser aninhados.

10.4.1 O básico

Uma referência para uma variável não local em uma linguagem com escopo estático com subprogramas aninhados requer 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, estão 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. A segunda parte 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 um 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 pode ser chamado apenas quando todos os seus subprogramas ancestrais estáticos estiverem ativos. 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 por escopos que envolvem a referência, o mais proximamente aninhado primeiro. Para suportar referências não locais, deve ser possível encontrar todas as instâncias de registros de ativação na pilha que correspondam a esses ancestrais estáticos. Essa observação leva à abordagem de implementação descrita na próxima subseção.

Não tratamos das questões de blocos até a Seção 10.5. No restante desta seção, assume-se que todos os escopos são definidos pelos subprogramas. Como funções não podem ser aninhadas nas linguagens baseadas em C (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 abordagem, 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 de quando a ligação estática não é incluída. Em vez de ter dois elementos do registro de ativação antes dos parâmetros, existem agora três: o endereço de retorno, a ligação estática e a ligação dinâmica.

Um **encadeamento estático** é uma cadeia de ligações estáticas que conectam certas instâncias de registro de ativação na pilha. Durante a execução de um procedimento *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 que está sendo 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 uma maneira relativamente direta. Quando é feita uma referência para uma variável não local, a instância de registro de ativação contendo a variável pode ser encontrada por meio de uma busca no encadeamento estático até que uma instância de registro de ativação de um ancestral estático que contenha a variável seja encontrada. Entretanto, pode ser muito mais fácil do que isso. 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 um escopo estático que indica o quanto profundamente ele está aninhado no escopo mais externo. Um procedimento principal em Ada tem uma profundidade estática de 0. Se o procedimento *A* é definido em um procedimento principal, sua profundidade estática é 1. Se o procedimento *A* contém a definição de um procedimento 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



Mantendo a simplicidade

NIKLAUS WIRTH

Niklaus Wirth começou sua carreira em Engenharia no Instituto Federal de Tecnologia da Suíça (ETH), em Zurique. Estudou no Canadá e obteve o doutorado (PhD) na Universidade da Califórnia, em Berkeley, em 1963. Ao longo de sua carreira, ocupou os cargos de professor assistente em Ciência da Computação (na Universidade de Stanford e na Universidade de Zurique), professor de Informática no ETH de Zurique e pesquisador na Xerox PARC, na Califórnia. Wirth ganhou o prêmio de Pioneiro da Computação da IEEE e recebeu o prêmio Turing da ACM por “desenvolver uma sequência de linguagens de programação inovadoras: EULER, ALGOL-W, PASCAL e MODULA”.

SOBRE PROJETOS E SOLUÇÕES

A estrutura clara e uniforme do Pascal criou um novo padrão. O que havia no Pascal que melhorou de forma tão significativa usando essas ferramentas? Principalmente o fato de o Pascal expressar os elementos de programação mais básicos e deixá-los serem compostos e combinados de maneira livre e geral. É uma linguagem estruturada tanto para sentenças quanto para dados.

Você investiu muito tempo pensando em soluções que casassem hardware e software: o computador Litith, Oberon e Modula. Como considerar soluções ferramentais de linguagens dentro de um framework mais amplo de hardware lhe ajuda a considerar ambos quando está projetando soluções? Se a arquitetura de computadores e os sistemas de software são construídos de maneira adequada e se encaixam harmoniosamente, ambas as partes se tornam mais simples e econômicas. Mais importante, tornam-se fáceis de serem entendidas.

Li a seguinte citação em uma entrevista que você concedeu: “Um bom projetista deve se basear na experiência, na precisão, no pensamento lógico e na exatidão. Nenhuma mágica fará o trabalho.” Como alguém é capaz de distinguir entre a mágica e a melhor solução e abandonar a primeira em favor da última? Na maioria das vezes, por meio da experiência. Bons professores que tenham essa habilidade para distinguir as coisas ajudam bastante.

Em outra entrevista, você mencionou o estado do projeto atualmente: “Os desejos dos usuários con-

tam mais do que suas necessidades, e as pessoas são mais facilmente atraídas por recursos ‘legais’ – mesmo que raramente usados – do que pelo objetivo de programas confiáveis e transparentes.” Se o oposto fosse verdade, como você acha que os sistemas operacionais e as interfaces Web de hoje pareceriam para o usuário comum? Como eles seriam melhores? Os clientes que estão especificando um sistema de software em geral não sabem exatamente o que querem. Seus “desejos” podem não refletir suas necessidades verdadeiras. Os resultados são especificações com muitos itens que em uma inspeção mais cuidadosa são bastante supérfluos, ou ao menos não importantes. Concentrar-se na essência e deixar de fora os adereços nos levaria a sistemas mais simples que seriam econômicos, fáceis de serem entendidos e operados e talvez menos suscetíveis a erros.

Você expressou a ideia de que o que a tecnologia torna possível é, muitas vezes, exagerado – por exemplo, usar inteligência artificial em uma missão para tentar criar máquinas que “pensem”. Por que acha que essa crença exagerada leva os cientistas a caminhos infrutíferos? O hype acerca de muitos itens modernos de TI – não só inteligência artificial – não engana tanto os cientistas, mas sim os consumidores (e as agências de fomento). Uma coisa é fomentar pesquisa de longo prazo com risco calculado outra coisa é vender produtos com promessas exageradas impossíveis de serem cumpridas.

SOBRE O ESTADO DA ARTE E SUAS FERRAMENTAS

Quais são os avanços em recursos de linguagem que mais contribuíram para uma programação melhor no últimos 10 anos? Estruturas de dados e de programas e a ideia associada a asserções e invariantes de laços. Projeto modular e compilação separada de módulos. Hierarquias de tipos de dados (chamada de subclasses com herança em POO).

Nessa disciplina comparativa de linguagens, os alunos estudam linguagens funcionais, lógicas, procedurais e POO. Como eles deveriam considerar a programação procedural em seus estudos e trabalhos? É um estilo histórico? Ou em breve se tornará um estilo de nicho? É importante que os estudantes de ciência da computação conheçam os vários paradigmas de programação: procedural, fun-

***“Concentrar-se na essência e deixar de fora os ade-
reços nos levaria a sistemas mais simples que seriam
econômicos, fáceis de serem entendidos e operados e
talvez menos suscetíveis a erros.”***

cional, lógico e orientado a objetos. É óbvio, porém, que o estilo procedural permanece o mais próximo do computador no qual os programas são interpretados (“executados”). O recurso característico de um computador é a memória com células individualmente atualizadas. Sua correspondência em linguagens de programação é a variável. O estilo orientado a objetos é baseado no estilo procedural; é uma variante dele, não muito diferente, ainda que os procedimentos agora sejam chamados de “métodos”, e chamar um procedimento seja “enviar uma mensagem”. Considero a programação funcional e a lógica “estilos de nicho” muito mais do que a programação procedural.

variável x é exatamente a diferença entre a profundidade estática do procedimento que contém a referência a x e a profundidade estática do procedimento contendo 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), ou 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 esqueleto de programa:

```
procedure A is
  procedure B is
    procedure C is
      ...
      end; -- de C
      ...
    end; -- de B
    ...
  end; -- de A
```

As profundidades estáticas de A, B e C são 0, 1 e 2, respectivamente. Se o procedimento C referencia uma variável declarada em A, o deslocamento de encadeamento dessa referência seria 2 (profundidade estática de C menos a profundidade estática de A). Se o procedimento C referencia uma variável declarada em B, o deslocamento de encadeamento dessa referência seria 1. Referências a variáveis locais podem ser manipuladas usando o mesmo mecanismo, com um deslocamento de encadeamento de 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 Ada:

```
procedure Main_2 is
  X : Integer;
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
      begin -- de Sub1
      A := B + C; <———————1
      ...
    end; -- de Sub1
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
        C, E : Integer;
        begin -- de Sub3
        ...
      end; -- de Sub3
```

```

    Sub1;
    ...
E := B + A; <-----2
end; -- de Sub3
begin -- de Sub2
...
Sub3;
...
A := D + E; <-----3
end; -- de Sub2
begin -- de Bigsub
...
Sub2(7);
...
end; -- de Bigsub
begin -- de Main_2
...
Bigsub;
...
end; -- de Main_2

```

A sequência de chamadas a procedimentos é

```

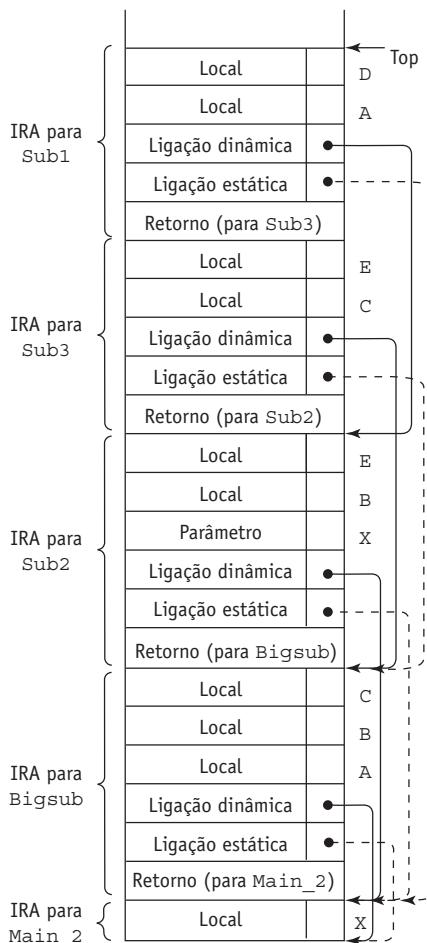
Main_2 chama Bigsub
Bigsub chama Sub2
Sub2 chama Sub3
Sub3 chama Sub1

```

A situação da pilha quando a execução chega pela primeira vez no ponto 1 desse programa é mostrada na Figura 10.9.

Na posição 1 no procedimento Sub1, a referência é para a variável local A, não para a variável não local A de Bigsub. Essa referência para A tem o par deslocamento de encadeamento /deslocamento local (0,3). A referência para 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 de 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 usada. 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 para a variável B é para aquela declarada em Sub2, porque ele é o



IRA = instância de registro de ativação

Figura 10.9 Conteúdo da pilha na posição 1 do programa.

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 um parâmetro. A referência para a 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_2, Bigsub e Sub2. Na posição 3 em Sub2, a referência para a variável A é para o A em Bigsub, com 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`, que 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)`

É razoável perguntar nesse ponto 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 efetivo não será importante. Assumimos 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 de 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, a instância de registro de ativação mais recente do escopo pai deve ser encontrada no momento da chamada.

Isso pode ser feito buscando as instâncias de registro de ativação no encadeamento dinâmico até que a primeira do escopo pai seja encontrada. Entretanto, essa busca pode ser evitada tratando as declarações e referências a procedimento 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 com um número de ligações igual à profundidade de aninhamento calculada em tempo de compilação.

Considere novamente o programa `Main_2` 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 para 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 (aquele 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 para apontar para a instância de registro de ativação de `Bigsub`.

Esse método funciona para todas as ligações de subprogramas, exceto quando parâmetros que são subprogramas estão envolvidos.

Uma crítica ao uso de encadeamento estático para acessar variáveis não locais é que referências a variáveis em escopos além do pai estático custam mais do 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 não é um problema sério. Outra crítica sobre a abordagem baseada em encadeamento estático é a dificuldade que um programador trabalhando em um programa com limitação de tempo tem de 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 em código longe das mudanças.

Algumas alternativas ao encadeamento estático têm sido desenvolvidas, mais notavelmente uma abordagem que usa uma estrutura de dados auxiliar chamada de mostrador (*display*). Entretanto, nenhuma das alternativas se mostrou superior ao método de encadeamento estático, ainda a abordagem mais usada. Logo, nenhuma delas é discutida aqui.

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 com outras variáveis com o mesmo nome declaradas em outros lugares do programa.

Os blocos podem ser implementados pelo uso do processo de encadeamento estático descrito na Seção 10.4 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 mais 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;
        ...
        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 `a` e `b` são retiradas da pilha quando o controle deixa seu bloco (antes de `f` e `g` serem alocadas).

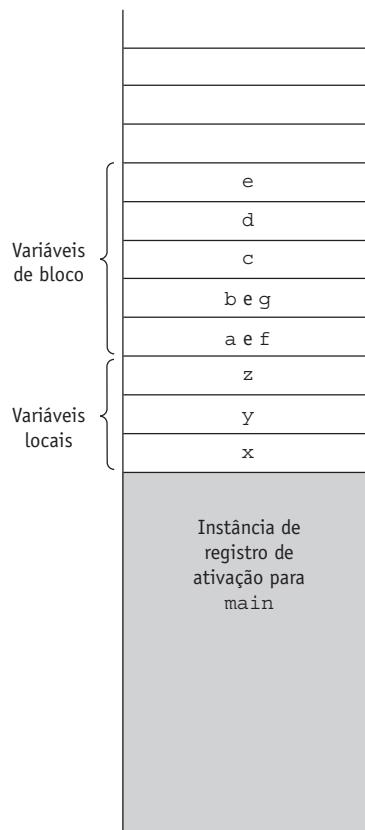


Figura 10.10 Armazenamento de variáveis de blocos quando os blocos não são tratados como procedimentos sem parâmetros.

10.6 IMPLEMENTANDO 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 as 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 o encadeamento dinâmico – em vez do estático – ser seguido. O encadeamento dinâmico liga todas as instâncias de registro de ativação na ordem inversa pela qual elas foram ativadas. Logo, o encadeamento dinâmico é exatamente o necessário para referenciar variáveis não locais em uma linguagem de escopo dinâmico. Esse método é chamado de **acesso profundo**, porque pode requerer buscas profundas na pilha.

Considere o seguinte programa de exemplo:

```
void sub3() {
    int x, z;
    x = u + v;
    ...
}
void sub2() {
    int w, x;
    ...
}
void sub1() {
    int v, w;
    ...
}
void main() {
    int v, u;
    ...
}
```

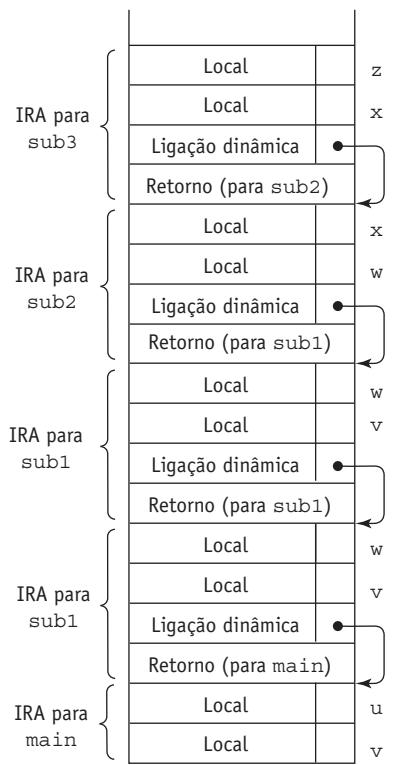
Esse programa está escrito em uma sintaxe que dá a ele uma aparência de ser em uma linguagem baseada em C, mas ele não é de uma linguagem em particular.

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 para as variáveis `x`, `u` e `v` na função `sub3`. A referência a `x` é encontrada na instância de registro de ativação de `sub3`. A referência para `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 procedimento `sub1`.



IRA = instânciā de regisrto de ativação

Figura 10.11 Conteúdo da pilha para um programa de escopo dinâmico.

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 de execução mais lentas do que 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

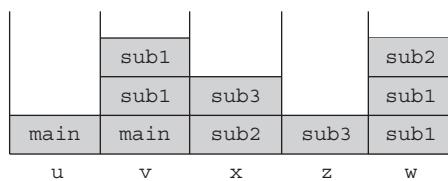
idêntica à do acesso raso. No método de acesso raso, as variáveis declaradas em subprogramas não são armazenadas nos registros de ativação desses subprogramas. Como com o escopo dinâmico existe, no máximo, uma versão visível de uma variável de um nome específico em um momento do tempo, uma abordagem 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 nova variável com um nome em particular 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 com 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 custoso.

A Figura 10.12 mostra as pilhas de variáveis para o programa de exemplo anteriormente mostrado na mesma situação mostrada com a pilha da Figura 10.11.

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, um bit chamado **ativo** é mantido 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, então 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 direto. 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 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



(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.

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 cara 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, então 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 uma ligação rápida de subprogramas, mas as referências às variáveis não locais, especialmente referências às variáveis não locais distantes (em termos do encadeamento de chamadas) são caras. O método de acesso raso fornece referências muito mais rápidas para variáveis não locais, especialmente para variáveis não locais distantes, mas é mais custosa 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 simples. 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. Podem 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 para 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, dentre 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.

CONJUNTO DE 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. Assuma que `Bigsub` está no nível 1.

```

procedure Bigsub is
  procedure A is
    procedure B is
      begin -- de B
      ... <-----1
    end; -- de B
  procedure C is
    begin -- de C
    ...
  B;
  ...
end; -- de C

```

```
begin -- de A
...
C;
...
end; -- de A
begin -- de Bigsub
...
A;
...
end; -- 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. Assuma que Bigsub está no nível 1.

```
procedure Bigsub is
    MySum : Float;
    procedure A is
        X : Integer;
        procedure B(Sum : Float) is
            Y, Z : Float;
            begin -- de B
            ...
            C(Z)
            ...
        end; -- de B
        begin -- de A
        ...
        B(X);
        ...
    end; -- de A
    procedure C(Plums : Float) is
        begin -- de C
        ...<-----1
        end; -- de C
        L : Float;
        begin -- de Bigsub
        ...
        A;
        ...
    end; -- 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. Assuma que Bigsub está no nível 1.

```
procedure Bigsub is
    procedure A(Flag : Boolean) is
        procedure B is
        ...
        A(false);
        end; -- de B
        begin -- de A
```

```

if flag
    then B;
    else C;
    ...
end; -- de A
procedure C is
    procedure D is
        ...
        ...
    end; -- de D
    ...
D;
end; -- de C
begin -- de Bigsub
    ...
A(true);
    ...
end; -- de Bigsub

```

A sequência de chamadas para esse programa para que a execução alcance D é
Bigsub calls 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 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;
    ...
}
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. Assuma que o programa do Problema 4 seja implementado usando o método de acesso raso, com uma pilha para cada nome de variável. Mostre as pilhas no momento da execução de `fun3`, assumindo 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 uma ativação em particular 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 usando 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 requereria uma sequência de caras comparações de cadeias representando nomes. Projete uma alternativa a essas comparações de nomes que seria mais rápida.
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 (veja Seção 10.4.2).
9. O método de encadeamento estático pode ser expandido levemente com o uso de duas ligações estáticas em cada instância de registro de ativação onde o segundo aponta para a instância de registro de ativação do avô estático. Como essa abordagem 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 abordagem 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 curta comparação de suas capacidades. (O projeto dessas instruções normalmente determina em parte o projeto de ligação de subprogramas por parte do escritor de compiladores).

EXERCÍCIOS DE PROGRAMAÇÃO

1. Escreva um programa que inclua dois subprogramas, um que receba um único parâmetro e que realiza 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 uma grande quantidade de vezes. Inclua no programa código para cronometrar e mostrar o tempo de execução das chamadas de cada um dos subprogramas. Rode o programa em uma máquina RISC e em uma máquina CISC e compare as taxas de tempo necessárias pelos dois subprogramas. Baseado nesses resultados, o que você pode dizer acerca da velocidade de passagem de parâmetros nas duas máquinas?

Capítulo 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** Nomeando encapsulamentos

Neste capítulo, exploramos as construções de linguagens de programação que suportam abstrações de dados. Dentre as novas ideias dos últimos 50 anos nas metodologias e no projeto de linguagens de programação, a abstração de dados é uma das mais profundas.

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 Ada, C++, Java, C# e Ruby. Implementações da mesma abstração de dados de exemplo são apresentadas em Ada, C++, Java C# e Ruby para mostrar as similaridades e as diferenças no projeto dos recursos de linguagem que suportam essas abstrações. A seguir, as capacidades de Ada, C++, Java 5.0 e C# 2005 para construir tipos de dados abstratos são discutidas.

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.

As classes de C++, Java e C# tratam suas variáveis de instância e de classe de forma diferente das variáveis definidas em seus métodos. O escopo de uma variável definida em um método começa em sua definição. Entretanto, independentemente de onde uma variável de instância ou de classe é definida em uma classe, seu escopo é a classe inteira.

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 cole 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 precisa incluir esses atributos. O mesmo ocorre para os pisco-de-peito-ruivo, pardais e pica-paus de barriga amarela. Esses atributos comuns nas descrições de espécies específicas de pássaros podem ser abstraídos. Dentro de uma espécie em particular, apenas os atributos que a distinguem precisam ser considerados, resultando em uma simplificação significativa das descrições. Uma visão menos abstrata de uma espécie, aquela de um pássaro, pode ser considerada quando for necessário ver um alto nível de detalhes.

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 foquem em atributos essenciais, enquanto ignoram 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 dentre os mais antigos no projeto de linguagens de programação – Plankalkül já suportava a abstração de processo. 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 é realizado (ao menos no programa chamador). Por exemplo, quando um programa precisa ordenar um vetor de objetos 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 em particular que `sortInt` implementa é um atributo não essencial para o usuário, que precisa ver apenas o nome e o protocolo do subprograma de ordenação para ser capaz de usá-lo.

A evolução da abstração de dados necessariamente seguiu a partir das abstrações de processo, visto que uma parte integral e central de todas as abstrações de dados são suas operações, definidas como abstrações de processos.

11.2 INTRODUÇÃO À ABSTRAÇÃO DE 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 usam. Unidades de programa que usam um tipo de dados abstrato podem declarar variáveis de tal tipo, mesmo que a representação real seja deles oculta. Um exemplar de um tipo de dados abstrato é chamado de **um objeto**.

Uma das motivações para a abstração de dados é similar à motivação para a abstração de processos: é uma arma contra a complexidade; uma forma de tornar programas grandes e/ou complicados mais gerenciáveis. Outras motivações e as vantagens dos tipos de dados abstratos são discutidas posteriormente nesta seção.

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 mais importantes.

11.2.1 Ponto flutuante como um tipo de dados abstrato

O conceito de um tipo de dados abstratos não é uma evolução recente. Todos os tipos de dados pré-definidos, mesmo os do Fortran I, são abstratos, apesar de raramente serem chamados assim. Por exemplo, considere um tipo de dados de ponto flutuante. A maioria das linguagens inclui ao menos um desses tipos. Um tipo de ponto flutuante fornece uma maneira de criar variáveis para 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 em uma célula de memória de ponto flutuante 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 usando as operações pré-definidas. O usuário não pode manipular diretamente as partes da representação real dos objetos porque essa representação é oculta. É esse o recurso que permite a portabilidade entre implementações de uma linguagem, mesmo que as implementações possam usar representações diferentes para tipos de dados em particular. Por exemplo, antes de o padrão de representações de ponto flutuante IEEE 754 aparecer no meio dos anos 1980, existiam diversas representações sendo usadas por diferentes arquiteturas de computadores. Entretanto, essa variação não impedia os programas que usavam tipos de ponto flutuante de serem portáveis para 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 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** satisfaçõas as duas condições a seguir:

- 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. A implementação do tipo e de suas operações podem estar na mesma unidade sintática ou em uma unidade separada. Além disso, outras unidades de programa podem criar variáveis do tipo definido.

- A representação dos objetos do tipo é ocultada das unidades de programa que o usam, então as únicas operações diretas possíveis nesses objetos são aquelas fornecidas na definição do tipo.

A principal vantagem de empacotar as declarações do tipo e suas operações em uma única unidade sintática é fornecer um método de organizar um programa em unidades lógicas possíveis de serem compiladas separadamente. Ter a implementação do tipo e suas operações em uma unidade sintática diferente mantém as especificações e suas implementações de maneira separada. Unidades de programa que usam um tipo de dados abstrato específico, chamadas de **clientes** desse tipo, precisam ver a especificação, mas não pode ser permitido que elas vejam a implementação. Se tanto as declarações quanto as definições de tipos e operações estão na mesma unidade sintática, deve existir alguma forma de ocultar dos clientes as partes da unidade que especifiquem as definições.

A vantagem de a interface não depender da representação ou da implementação das operações é permitir que a representação e/ou a implementação sejam modificadas sem mudanças aos clientes do tipo.

Um benefício importante do ocultamento de informação é um aumento na confiabilidade. Clientes não podem manipular as representações subjacentes dos objetos diretamente, seja intencionalmente ou por acidente, aumentando a integridade de tais objetos. Objetos podem ser modificados apenas por meio das operações fornecidas.

11.2.3 Um exemplo

Suponha um tipo de dados abstrato construído para uma pilha que com as seguintes operações abstratas:

<code>create(stack)</code>	Cria e possivelmente inicializa um objeto de pilha
<code>destroy(stack)</code>	Libera o armazenamento para a pilha
<code>empty(stack)</code>	Um predicado ou função booleana que retorna verdadeiro (<i>true</i>) se a pilha especificada é vazia e falso (<i>false</i>) caso contrário.
<code>push(stack, element)</code>	Insere o elemento especificado na pilha especificada
<code>pop(stack)</code>	Remove o elemento do topo da pilha especificada
<code>top(stack)</code>	Retorna uma cópia do elemento do topo da pilha especificada

Note que algumas implementações de tipos de dados abstratos não requerem 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);
if(! empty(stk1))
    temp = top(stk1);
...
```

Suponha que a implementação original da abstração pilha use uma representação com adjacência (que implementa uma pilha em um vetor). Posteriormente, por problemas de gerenciamento de memória com a representação usando adjacência, ela é modificada para uma representação baseada em lista encadeada. 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. Em particular, a sequência de código de exemplo não precisa ser modificada. É claro, uma mudança no protocolo de qualquer operação pode requerer alterações nos clientes.

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 pré-definidas 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. Dentre essas, estão operações para atribuição e comparações para igualdade e diferença. Se a linguagem não permite que os usuários sobrecarreguem a atribuição, ela deve ser pré-definida. Comparações de igualdade e diferença devem ser pré-definidas 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 usuário pode querer que seja a das estruturas referenciadas pelos ponteiros.

Algumas operações são necessárias por muitos tipos de dados abstratos, mas como não são universais, devem ser fornecidas pelo projetista do tipo. Dentre elas estão os iteradores, métodos de acesso, construtores e destrutores. Iteradores foram discutidos no Capítulo 8. Os métodos de acesso fornecem um caminho para dados ocultos para acesso direto pelos clientes. Os construtores são usados para a inicialização de partes de objetos recém-criados. Os destrutores servem para recuperar armazenamento no monte para ser 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++, Java e C#, suportam tipos de dados abstratos diretamente. Um método alternativo, usado por Ada, é fornecer uma construção de encapsulamento mais generalizada que possa definir qualquer número de entidades, qualquer das quais pode ser especificada para ser visível de fora de sua unidade. Esses invólucros não são tipos de dados abstratos, mas generalizações, que podem ser usadas para definir tipos de dados abstratos. Apesar de discutirmos a construção de encapsulamento de Ada nesta seção, a tratamos como um encapsulamento mínimo para tipos de dados únicos. Encapsulamentos generalizados são discutidos na Seção 11.6.

Então, a primeira questão de projeto para tipos de dados abstratos é a forma do contêiner para a interface do tipo. A segunda é se os tipos de dados abstratos podem ser parametrizados. Por exemplo, se a linguagem suporta tipos de dados abstratos parametrizados, alguém deve projetar um tipo de dados abstrato para filas que poderiam armazenar elementos de qualquer tipo. Tipos de dados abstratos parametrizados são discutidos na Seção 11.5. A terceira questão de projeto é quais controles de acesso são fornecidos e como são especificados.

11.4 EXEMPLOS DE LINGUAGEM

O conceito de abstração de dados tem suas origens no SIMULA 67, apesar de essa linguagem não fornecer suporte completo para tipos de dados abstratos. Nesta seção, descrevemos o suporte para abstração fornecido por Ada, C++, Java, C# e Ruby.

11.4.1 Tipos de dados abstratos em Ada

Ada fornece uma construção de encapsulamento que pode ser usada para definir um tipo de dados abstrato único, incluindo a habilidade de ocultar sua representação. Ada foi uma das primeiras linguagens a oferecer suporte completo para esses tipos de dados.

11.4.1.1 Encapsulamento

As construções de encapsulamento em Ada são chamadas de **pacotes**, que podem ter duas partes, chamadas de **pacote de especificação** (que fornece a interface do encapsulamento) e **pacote de corpo** (que fornece a implementação da maioria das entidades nomeadas no pacote de especificação associado, ou de todas elas). Nem todos os pacotes têm a parte do corpo (os que encapsulam apenas tipos e constantes não têm ou não precisam de um corpo).

Um pacote de especificação e seu pacote de corpo associado compartilham o mesmo nome. A palavra reservada **body** em um cabeçalho de pacote o identifica como de corpo. Pacotes de especificação e de corpo podem ser compilados separadamente, desde que o de especificação seja compilado primeiro.

11.4.1.2 Ocultamento de informação

O projetista de um pacote Ada que define um tipo de dados pode escolher tornar o tipo inteiramente visível aos clientes ou fornecer apenas as informações de interface. É claro, se a representação não é oculta, o tipo definido não é um tipo de dados abstrato. Existem duas abordagens para ocultar a representação dos clientes no pacote de especificação. Uma é incluir duas seções no pacote de especificação – uma na qual as entidades são visíveis aos clientes e outra que oculta seus conteúdos. Para um tipo de dados abstrato, uma declaração aparece na parte visível da especificação, fornecendo apenas o nome do tipo e a informação de que sua representação é oculta. A representação aparece em uma parte da especificação chamada de **privada**, introduzida pela palavra reservada **private**. A cláusula privada sempre aparece no final do pacote de especificação. Ela é visível para o compilador, mas não para as unidades de programa cliente.

A segunda maneira de ocultar a representação é definir o tipo de dados abstrato como um ponteiro e fornecer a definição da estrutura apontada no pacote de corpo, cujo conteúdo inteiro é oculto para os clientes.

A seguir, temos um exemplo da primeira abordagem para ocultar a representação de um tipo de seus clientes. Suponha que um tipo de dados abstrato chamado `Node_Type` será definido em um pacote. `Node_Type` é declarado na parte visível do pacote de especificação sem seus detalhes de representação, como em

```
type Node_Type is private;
```

Tipos declarados como privados são chamados de **tipos privados** e têm operações pré-definidas para atribuição e comparações para igualdade e para diferença. Qualquer outra operação deve ser declarada no pacote de especificação que definiu o tipo.

Note que na cláusula privada do exemplo a seguir, a declaração de `Node_Type` é repetida, mas com a definição de tipo completa:

```

package Linked_List_Type is
  type Node_Type is private;
  ...
  private
    type Node_Type;
    type Ptr is access Node_Type;
    type Node_Type is
      record
        Info : Integer;
        Link : Ptr;
      end record;
  end Linked_List_Type;

```

A cláusula privada desse pacote tem tanto uma declaração quanto uma definição de `Node_Type`. A declaração é necessária por causa da referência a `Node_Type` na definição de `Ptr`, que deve preceder a definição de `Node_Type`. Como eles são definidos na cláusula privada, nem `Info` nem `Link` são visíveis para os clientes de `Linked_List_Type`.

Se nenhuma das entidades em um pacote será oculta, não existe um objetivo ou uma necessidade para a parte privada da especificação. É claro, tal pacote não pode definir um tipo de dados abstrato.

A razão pela qual a representação de um tipo aparece no pacote de especificação tem tudo a ver com questões de compilação. Um cliente pode ver apenas o pacote de especificação (não o pacote de corpo), mas o compilador deve ser capaz de alocar objetos do tipo exportado quando estiver compilando o cliente. Além disso, o cliente é compilável quando apenas o pacote de especificação para o tipo de dados abstrato tenha sido compilado e estiver presente. Logo, o compilador deve ser capaz de determinar o tamanho de um objeto a partir do pacote de especificação. Assim, a representação do tamanho deve ser visível ao compilador, mas não ao código do cliente. Essa é exatamente a situação especificada pela cláusula privada em um pacote de especificação.

De certa forma, é problemático que o pacote de especificação forneça parte dos detalhes de implementação (a definição dos dados) acima, enquanto os detalhes de implementação restantes (a definição das operações) estão no pacote de corpo. Seria muito mais claro se a especificação fornecesse apenas a interface e o corpo fornecesse todos os detalhes de implementação. Esse problema pode ser aliviado tornando o tipo de dados abstrato um ponteiro, como no pacote a seguir:

```

package Linked_List_Type is
  type Node is private;
  function Create_Node() return Node;
  private
    type Node_Record;
    type Node is access Node_Record;
  end Linked_List_Type;

```



C++: nascimento, onipresença e críticas comuns

BJARNE STROUSTRUP

Bjarne Stroustrup é o projetista e implementador original de C++ e autor de *The C++ Programming Language* e *The Design and Evolution of C++*. Seus interesses em pesquisa incluem sistemas distribuídos, simulação, projeto, programação e linguagens de programação. Dr. Stroustrup é professor de Ciência da Computação na Faculdade de Engenharia da Universidade do Texas A&M e está ativamente envolvido na padronização ANSI/ISO do C++. Após mais de duas décadas na AT&T, mantém uma ligação com os laboratórios da companhia, pesquisando como membro do Laboratório de Pesquisa em Informação e Sistemas de Software. Bjarne é ACM Fellow, AT&T Bell Laboratories Fellow e AT&T Fellow. 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.”

UM BREVE HISTÓRICO SOBRE VOCÊ E 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 no “C com Classes” (que mais tarde se tornou C++)? Trabalhei em alguns projetos relacionados à computação distribuída antes de iniciar o 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 levou você 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 um fraco, mas ainda isso é muito melhor do que não saber nada de matemática. No momento em que me inscrevi, nunca havia visto um computador. O que eu amo na computação é a programação, e não os campos mais matemáticos.

DISSECANDO UMA LINGUAGEM BEM-SUCEDIDA

Gostaria de começar de trás para frente, listando alguns itens que eu acredito terem o C++ onipresente, e ver sua reação. É “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 que existem 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 (que, no momento do nascimento de C++, era o maior setor do mercado de desenvolvimento).

Sim, C++ é um forte concorrente para qualquer projeto de programação de sistemas. Ele também é eficaz para a programação de sistemas embarcados, atualmente o setor de maior crescimento. Outra área

de crescimento para C++ é a programação de alto desempenho numérica, de engenharia e científica.

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 multiparadigmas. Ou seja, ela suporta diversos estilos de programação fundamentais (incluindo a programação orientada a objetos) e combinações desses estilos. Quando bem usadas, isso leva a bibliotecas mais limpas, flexíveis e eficientes do que poderiam ser fornecidas usando apenas um paradigma. Os contêineres e algoritmos da biblioteca padrão de C++, basicamente um *framework* de programação genérica, são um exemplo. Quando usadas 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 na 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 unicamente colaborativo para pesquisas práticas. C++ emergiu do mesmo laboratório de pesquisa de C e se beneficiou da mesma tradição intelectual, experiência e pessoas excepcionais. Por meio disso, a AT&T suportou a padronização de C++. Entretanto, C++ não teve uma campanha de marketing massiva, 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 obter suas reações: C++ é uma linguagem imensa/selvagem. 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++ “selvagem”. O programa “hello world” em C++ não é maior do 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 do 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 do 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. C++ é adequado para sistemas embarcados hoje. 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++. Jogos é uma área relacionada na qual se usa muito C++.

Esqueci alguma pergunta que você ouve com frequência? Claro.

Agora, todos os detalhes da implementação podem ser fornecidos no pacote de corpo, como em:

```
package body Linked_List_Type is
    type Node_Record is
        record
            Info : Integer;
            Link : Node;
        end record;
        ...
end Linked_List_Type;
```

Existem diversos problemas com essa versão de certa forma mais clara. Primeiro, existem as dificuldades inerentes de lidar com ponteiros. Segundo, comparações entre dois objetos do novo tipo de dados abstrato serão entre ponteiros, o que não produz os resultados esperados – porque os ponteiros são comparados, em vez de os objetos para os quais eles apontam. Outro problema de definir um tipo de dados abstrato como um ponteiro é a inabilidade do tipo de controlar a alocação e a liberação de objetos do tipo. Por exemplo, um cliente pode criar um ponteiro para um objeto (com uma declaração de variável) e usá-lo sem criar um objeto.

Uma alternativa aos tipos privados é uma forma mais restrita: os **tipos privados limitados**. Tipos privados limitados não baseados em ponteiros são descritos na seção privada de um pacote de especificação, assim como os tipos privados não baseados em ponteiros. A única diferença sintática é que os tipos privados limitados são declarados como **limited private** na parte visível da especificação do pacote. A diferença semântica é que objetos de um tipo que é declarado como privado limitado não têm operações pré-definidas. Tal tipo é útil quando as operações pré-definidas de atribuição e de comparação não são significativas nem úteis. Por exemplo, atribuições e comparações raramente são usadas para pilhas. Se a atribuição e as comparações forem necessárias, mas as versões pré-definidas não forem úteis, essas operações devem ser fornecidas pelo pacote de especificação. Essa é uma maneira de evitar os problemas de comparação quando o tipo de dados abstrato é um ponteiro. A operação de atribuição deve estar na forma de um procedimento normal, enquanto os operadores de igualdade e de diferença podem ser fornecidos pela sobrecarga desses operadores para o novo tipo.

11.4.1.3 Um exemplo

A seguir, temos o pacote de especificação para um tipo de dados abstrato pilha:

```
package Stack_Pack is
    -- As entidades visíveis, ou interface pública
    type Stack_Type is limited private;
```

```

Max_Size : constant := 100;
function Empty(Stk : in Stack_Type) return Boolean;
procedure Push(Stk : in out Stack_Type;
              Element : in Integer);
procedure Pop(Stk : in out Stack_Type);
function Top(Stk : in Stack_Type) return Integer;
-- A parte que é oculta dos clientes
private
  type List_Type is array (1..Max_Size) of Integer;
  type Stack_Type is
    record
      List : List_Type;
      Topsub : Integer range 0..Max_Size := 0;
    end record;
end Stack_Pack;

```

Note que nenhuma operação de criação ou destruição é incluída, porque elas não são necessárias.

O pacote de corpo para `Stack_Pack` é:

```

with Ada.Text_IO; use Ada.Text_IO;
package body Stack_Pack is
  function Empty(Stk: in Stack_Type) return Boolean is
  begin
    return Stk.Topsub = 0;
  end Empty;

  procedure Push(Stk : in out Stack_Type;
                 Element : in Integer) is
  begin
    if Stk.Topsub >= Max_Size then
      Put_Line("ERROR - Stack overflow");
    else
      Stk.Topsub := Stk.Topsub + 1;
      Stk.List(Topsub) := Element;
    end if;
  end Push;

  procedure Pop(Stk : in out Stack_Type) is
  begin
    if Stk.Topsub = 0
      then Put_Line("ERROR - Stack underflow");
    else Stk.Topsub := Stk.Topsub - 1;
    end if;
  end Pop;

  function Top(Stk : in Stack_Type) return Integer is
  begin

```

```
if Stk.Topsub = 0
    then Put_Line("ERROR - Stack is empty");
    else return Stk.List(Stk.Topsub);
end if;
end Top;
end Stack_Pack;
```

A primeira linha do código desse pacote de corpo contém duas cláusulas: **with** e **use**. A cláusula **with** torna os nomes definidos em pacotes externos visíveis; nesse caso Ada.Text_IO, que fornece funções para entrada e saída de texto. A cláusula **use** elimina a necessidade de qualificação explícita de referências a entidades a partir do pacote nomeado. As questões de acesso a encapsulamentos externos e qualificações de nomes serão discutidas na Seção 11.6.

O pacote de corpo deve ter as definições de subprogramas com cabeçalhos que casem com os de subprogramas no pacote de especificação associado. O pacote de especificação promete que esses subprogramas serão definidos no pacote de corpo associado.

O seguinte procedimento, Use_Stacks, é um cliente do pacote Stack_Pack. Ele ilustra como o pacote poderia ser usado.

```
with Stack_Pack;
use Stack_Pack;
procedure Use_Stacks is
    Topone : Integer;
    Stack : Stack_Type;    -- Cria um objeto Stack_Type
begin
    Push(Stack, 42);
    Push(Stack, 17);
    Topone := Top(Stack);
    Pop(Stack);
    ...
end Use_Stacks;
```

Uma pilha é um exemplo tolo para a maioria das linguagens contemporâneas, porque o suporte para pilhas é incluído em suas bibliotecas de classe padrão. Entretanto, pilhas fornecem um exemplo simples que podemos usar para permitir comparações das linguagens discutidas nesta seção.

11.4.2 Tipos de dados abstratos em C++

C++ 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 componentes primários da programação orientada a objetos são os tipos de dados abstratos, C++ obviamente deve suportá-los.

Enquanto Ada fornece um encapsulamento que pode ser usado para simular tipos de dados abstratos, C++ fornece duas construções bastante simi-

lares uma a outra, a classe (**class**) e a estrutura (**struct**), que suportam mais diretamente os tipos de dados abstratos. Como essas estruturas são mais comumente usadas quando apenas dados são incluídos, não as discutimos aqui.

Classes em C++ são tipos; conforme mencionado, os pacotes em Ada são encapsulamentos mais generalizados que podem definir qualquer número de tipos. Uma unidade de programa que ganha visibilidade para um pacote Ada pode acessar qualquer uma de suas entidades públicas diretamente por seus nomes. 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. Essa é uma maneira mais limpa e direta de fornecer tipos de dados abstratos.

11.4.2.1 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. Neste capítulo, apenas os membros de instância de uma classe são discutidos. Todas as instâncias de uma classe compartilham um conjunto único de funções membro, mas cada instância 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. Objetos de classe dinâmicos do monte são criados com o operador **new** e destruídos 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: a definição completa pode aparecer na classe ou apenas 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 requerer o processo 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 sejam internalizadas era economizar tempo 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, resultando em uma redução na legibilidade.

Colocar as definições das funções membro fora da definição da classe serve para a especificação da implementação, um objetivo comum da programação moderna.

11.4.2.2 Ocultamento 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 **public**. Logo, a cláusula **public** descreve a interface para objetos da classe. Existe ainda uma terceira categoria de visibilidade, **protected**, discutida no contexto de herança no Capítulo 12.

11.4.2.3 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 sendo criados. Um construtor pode também 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 ele inicializa. Os construtores podem ser sobreescritos, mas é claro que cada construtor de uma classe deve ter um perfil de parâmetros único.

Uma classe C++ pode também incluir uma função chamada de um **destrutor**, implicitamente chamado quando o tempo de vida de uma instância da classe termina. Conforme mencionado, instâncias de classes 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 eles referenciam. Os destrutores são usados como uma ajuda à depuração, simplesmente 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.

11.4.2.4 Um exemplo

Nosso exemplo de um tipo de dados abstrato em C++ é, mais uma vez, uma pilha:

```
#include <iostream.h>
class Stack {
    private: /** Esses membros são visíveis apenas para outros
               *** membros e amigos (veja Seção 11.6.4)
    int *stackPtr;
```

```

int maxLen;
int topPtr;
public: /** Esses membros são visíveis para os clientes
Stack() { /** Um construtor
    stackPtr = new int [100];
    maxLen = 99;
    topPtr = -1;
}
~Stack() {delete [] stackPtr;} /** Um destrutor
void push(int number) {
    if (topPtr == maxLen)
        cerr << "Error in push--stack is full\n";
    else stackPtr[+topPtr] = number;
}
void pop() {
    if (topPtr == -1)
        cerr << "Error in pop--stack is empty\n";
    else topPtr--;
}
int top() {return (stackPtr[topPtr]);}
int empty() {return (topPtr == -1);}
}

```

Discutimos poucos aspectos dessa definição de classe, porque não é necessário 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 `topPtr` – 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 elas 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 do monte (`int`). Ele também inicializa `maxLen` e `topPtr`. O propósito da função destrutora é liberar o armazenamento para o vetor usado para implementar a pilha quando o tempo de vida de um objeto `Stack` termina. Esse vetor foi alocado pelo construtor.

Um programa de exemplo que usa o tipo de dados abstrato `Stack` é

```

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` com apenas os protótipos das funções membro. Esse código é armazenado em um arquivo de cabeçalho com a extensão do nome do 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 de nome `.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 (veja Seção 11.6.4)
    int *stackPtr;
    int maxLen;
    int topPtr;
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;
    topPtr = -1;
}
Stack::~Stack() {delete [] stackPtr;} //** Um destrutor

void Stack::push(int number) {
    if (topPtr == maxLen)
        cerr << "Error in push--stack is full\n";
    else stackPtr[+topPtr] = number;
}

void Stack::pop() {
    if (topPtr == -1)

```

```

        cerr << "Error in pop--stack is empty\n";
    else topPtr--;
}

int Stack::top() {return (stackPtr[topPtr]);}

int Stack::empty() {return (topPtr == -1);}

```

11.4.2.5 Avaliação

O suporte de C++ para tipos de dados abstratos, por meio de sua construção de classes, é similar ao poder de expressão do suporte de Ada, por seus pacotes. Ambos fornecem mecanismos efetivos para o encapsulamento e o ocultamento de informação de tipos de dados abstratos. A diferença primária é que classes são tipos, enquanto os pacotes de Ada são encapsulamentos mais gerais. Além disso, a construção de classe foi projetada para mais do que abstração de dados, como discutido no Capítulo 12.

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 tipos de dados definidos pelo usuário em Java são classes (Java não inclui *structs*), e todos os objetos são alocados do monte e acessados por meio de variáveis de referência. Outra diferença é que os métodos em Java devem ser definidos completamente em uma classe. Um corpo de método deve aparecer com seu cabeçalho de método correspondente. Logo, um tipo de dados abstrato é declarado e definido em uma única unidade sintática. Um compilador Java pode internalizar qualquer método não sobreescrito. As definições são ocultas dos clientes tornando-as privadas.

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.

A seguir, temos uma definição de classe em Java para nosso exemplo da pilha:

```

import java.io.*;
class StackClass {
    private int [] stackRef;
    private int maxLen,
            topIndex;
    public StackClass() { // Um construtor
        stackRef = new int [100];
        maxLen = 99;
        topIndex = -1;
    }
    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 (topIndex == -1)
        System.out.println("Error in pop-stack is empty");
    else --topIndex;
}
public int top() {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 por causa da coleção de lixo implícita em Java.

Nosso exemplo não ilustra uma das diferenças importantes entre o suporte para tipos de dados abstratos de C++ e de Java. Entretanto, como será discutido na Seção 11.6, existem mais diferenças entre os encapsulamentos de C++ e os de Java. Além disso, quando consideramos outros aspectos da programação orientada a objetos, como é feito no Capítulo 12, mais diferenças entre as classes Java e as de C++ serão discutidas.

11.4.4 Tipos de dados abstratos em C#

Lembre que C# é baseado tanto em C++ quanto em Java e inclui algumas construções novas.

C# usa os modificadores de acesso **private**, **public** e **protected**¹ exatamente como eles são usados em Java. Entretanto, inclui também dois modificadores que Java não tem, **internal** e **protected internal**. O modificador **internal** é descrito na Seção 11.6, onde encapsulamentos generalizados são discutidos.

Assim como em Java, todas as instâncias de classe em C# são dinâmicas do monte. Construtores padrão, que fornecem valores iniciais para dados

¹ O modificador de acesso **protected** é discutido no Capítulo 12.

de instância, são pré-definidos para todas as classes. Esses construtores fornecem valores iniciais típicos, como 0 para tipos `int` e `false` para tipos booleanos (`boolean`). Um usuário pode fabricar um construtor para qualquer classe que definir. Esse construtor pode atribuir valores iniciais a alguns ou 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.

Como C# usa coleção de lixo para a maioria de seus objetos do monte, os destrutores são raramente usados.

Apesar de os princípios de tipos de dados abstratos ditarem que membros de dados de objetos sejam ocultos dos clientes, surgem muitas situações nas quais eles devem acessar esses membros. A solução comum é fornecer métodos de acesso, 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. As razões pelas quais os métodos de acesso são melhores:

1. Acesso apenas de leitura pode ser fornecido, tendo 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 uma 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 a única forma de acesso.

C# fornece propriedades, herdadas do Delphi, como uma maneira de implementar leitores e escritores sem requerer chamadas a métodos explícitos. 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`, que fornece um método de leitura escrita e um método de escrita para acesso ao membro de dados privado `degreeDays`. No código cliente logo a seguir da definição da classe, `degreeDays` é tratado como uma variável membro pública, apesar de o acesso a ela estar disponível apenas pela propriedade. Note o uso da variável implícita `value` no método de escrita. Esse é o mecanismo pelo qual o novo valor da propriedade é referenciado.

Conforme mencionado na Seção 11.4.2, C++ inclui tanto classes quanto estruturas (*structs*), 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 fundamental 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. Se forem passadas como parâmetros, elas são passadas por valor. Todos os tipos de valores em C#, incluindo seus tipos primitivos, são estruturas. Apesar de isso parecer estranho, objetos de estruturas são criados com o mesmo operador `new` usado para criar objetos de classe.

Estruturas são usadas em C# primariamente 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.

11.4.5 Tipos de dados abstratos em Ruby

Ruby fornece suporte completo 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.

Em Ruby, uma classe é definida em uma sentença composta aberta com a palavra reservada `class`. Variáveis locais têm nomes com a forma de nomes de variáveis em outras linguagens de programação. Os nomes das variáveis de instância começam com o sinal arroba (@). Classes podem ter variáveis de classes (instanciadas com a classe, em vez de com a instância), cujos nomes devem começar com dois sinais arroba (@@). Métodos de instância têm a mesma sintaxe que as funções em Ruby: começam com a palavra reservada `def` e são fechados 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`.

Se uma classe precisa de mais de um construtor, todos devem ter nomes únicos. O construtor `initialize` é implicitamente chamado quando o método de classe `new` é chamado. Outros construtores são definidos como outros métodos, mas apenas chamam `new` com os parâmetros a serem enviados para `initialize`, que será implicitamente chamado. Por exemplo, considere o seguinte construtor em uma classe chamada `Circle`:

```
def initialize(red, green, blue, radius)
  @red, @green, @blue, @radius = red, green, blue, radius
end
```

Um segundo construtor que cria um objeto da classe `Circle` com o raio de 5,0 é mostrado a seguir:

```
def Circle.r5circle(red, green, blue) :
  new(red, green, blue, 5.0)
end
```

Métodos de uma classe podem ser marcados como privados ou públicos, sendo públicos o padrão². Acessos privados e públicos em Ruby têm o mesmo significado que em Java. Todos os membros de dados devem ser privados para suportar ocultamento de informação.

Classes em Ruby são dinâmicas no sentido que seus membros podem ser adicionados em qualquer momento, simplesmente incluindo definições de classe adicionais que especificam novos membros. Métodos também podem ser removidos de uma classe, ao se fornecer outra definição de classe na qual o método a ser removido é enviado ao `remove_method` como um parâmetro. As classes dinâmicas de Ruby são outro exemplo de um projetista de linguagem trocando a legibilidade (e a confiabilidade) por flexibilidade. Para determinar a definição atual de uma classe, é necessário encontrar suas definições no programa e considerar todas elas.

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 um vetor

class StackClass

# Constructor

def initialize
  @stackRef = Array.new
```

² Ruby também suporta o modo de acesso protegido, conforme discutido no Capítulo 12.

```
    @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 @topIndex == -1
    puts "Error in pop - stack is empty"
  else
    @topIndex = @topIndex - 1
  end
end

# top method

def top
  @stackRef[@topIndex]
end

# empty method
def empty
  @topIndex == -1
end
end # of Stack class

# Test code for 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 que a notação `#{variável}` converte o valor da variável para uma cadeia, então inserida na cadeia na qual aparece. Essa classe define uma estrutura de pilha que pode armazenar objetos de qualquer tipo. Em Ruby, tudo são objetos e as matrizes são, na verdade, matrizes de referências a objetos, o que torna essa pilha mais flexível do que os exemplos similares em Ada, 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.

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 essa é apenas uma questão para linguagens estaticamente tipadas. Em uma linguagem dinamicamente tipada, como Ruby, qualquer pilha pode implicitamente armazenar elementos de qualquer tipo. Nas quatro seções seguintes, são discutidas as capacidades de Ada, C++, Java 5.0 e C# 2005 para construir tipos de dados abstratos parametrizados.

11.5.1 Ada

Procedimentos genéricos em Ada foram discutidos e ilustrados no Capítulo 9. Pacotes também podem ser genéricos, então podemos construir tipos de dados abstratos genéricos, ou parametrizados.

O exemplo do tipo de dados abstrato pilha em Ada mostrado na Seção 11.4.1 sofre de duas restrições: (1) as pilhas desse tipo podem armazenar apenas elementos do tipo inteiro e (2) as pilhas podem ter apenas 100 elementos no máximo. Ambas as restrições podem ser eliminadas usando um pacote genérico, que pode ser instanciado para outros tipos de elementos e de qualquer tamanho desejado. (Essa é uma instanciação genérica, bem diferente da instanciação de uma classe para criar um objeto). A seguinte especificação de pacote descreve a interface de um tipo de dados abstrato pilha genérico com esses recursos:

```
generic
  Max_Size : Positive;    -- Um parâmetro genérico para
                           -- o tamanho da pilha
  type Element_Type is private;  -- Um parâmetro genérico
                           -- para o tipo de elemento
```

```
package Generic_Stack is
    -- As entidades visíveis, ou interface pública
    type Stack_Type is limited private;
    function Empty(Stk : in Stack_Type) return Boolean;
    procedure Push(Stk : in out Stack_Type;
                   Element : in Element_Type);
    procedure Pop(Stk : in out Stack_Type);
    function Top(Stk : in Stack_Type) return Element_Type;
    -- A parte oculta
private
    type List_Type is array (1..Max_Size) of Element_Type;
    type Stack_Type is
        record
            List : List_Type;
            Topsub : Integer range 0..Max_Size := 0;
        end record;
end Generic_Stack;
```

O pacote de corpo para `Generic_Stack` é o mesmo que para `Stack_Pack` na seção anterior, exceto que o tipo do parâmetro formal `Element` em `Push` e `Top` é `Element_Type` em vez de `Integer`. A seguinte sentença instancia `Generic_Stack` para uma pilha de 100 elementos do tipo `Integer`:

```
package Integer_Stack is new Generic_Stack(100, Integer);
```

Alguém poderia também construir um tipo de dados abstrato para uma pilha de tamanho 500 para elementos `Float`, como em

```
package Float_Stack is new Generic_Stack(500, Float);
```

Essas instanciações constroem duas versões de código fonte diferentes de `Generic_Stack` em tempo de compilação.

11.5.2 C++

C++ também suporta tipos de dados abstratos parametrizados. Para tornar o exemplo da classe pilha em C++ da Seção 11.4.2 genérico no tamanho da pilha, apenas a função construtora precisa ser modificada, como no código:

```
Stack(int size) {
    stkPtr = new int [size];
    maxLen = size - 1;
    top = -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 possam 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 <class Type> // Type é o parâmetro de template
class Stack {
    private:
        Type *stackPtr;
        int maxlen;
        int topPtr;
    public:
        // Um construtor para pilhas de 100 elementos
        Stack() {
            stackPtr = new Type [100];
            maxlen = 99;
            topPtr = -1;
        }
        // A constructor for a given number of elements
        Stack(int size) {
            stackPtr = new Type [size];
            maxlen = size - 1;
            topPtr = -1;
        }
        ~Stack() {delete stackPtr;} // Um destrutor
        void push(Type number) {
            if (topPtr == maxlen)
                cout << "Error in push-stack is full\n";
            else stackPtr[++topPtr] = number;
        }
        void pop() {
            if (topPtr == -1)
                cout << "Error in pop-stack is empty\n";
            else topPtr--;
        }
        Type top() {return (stackPtr[topPtr]);}
        int empty() {return (topPtr == -1);}
}
```

Como em Ada, classes *template* em C++ são instanciadas em tempo de compilação. A diferença é que em C++ as instanciações são implícitas: uma nova instanciação é criada sempre que é criado um objeto que requeira uma versão da classe *template* até então não existente.

11.5.3 Java 5.0

Java 5.0 suporta uma forma de tipos de dados abstratos parametrizados na qual os parâmetros genéricos devem ser classes. Lembre que tais parâmetros são 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 armazenam objetos da classe `Object`, então podem armazenar quaisquer objetos (mas não tipos primitivos). Logo, os tipos de coleção sempre foram capazes de armazenar múltiplos tipos (desde que fossem classes). Existem três problemas acerca disso. Primeiro, sempre que um objeto é removido da coleção, ele deve ser convertido para o tipo apropriado. Segundo, não existe verificação de erros quando elementos são adicionados à coleção. Terceiro, os tipos de coleção não podem armazenar tipos primitivos. Então, para armazenar valores `int` em um `ArrayList`, o valor primeiro deve ser colocado em um objeto 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);
```

Em Java 5.0, as classes de coleção, cujas mais usadas são `List`, `ArrayList` e `Queue`, tornaram-se classes genéricas. Essas classes são instanciadas chmando-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 com as coleções anteriores a Java 5.0. Apenas objetos da classe `Integer` podem ser colocados na coleção `myArray`. Além disso, não existe a necessidade de converter um objeto sendo removido da coleção. Entretanto, ainda não é possível instanciar uma coleção genérica que armazena valores primitivos.

Lembre que no Capítulo 9 descrevemos que Java 5.0 suporta classes coringa. Por exemplo, `Collection<?>` é uma classe coringa para todas as classes de coleção. Isso permite que métodos sejam escritos 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 coringa. Por exemplo, como os componentes de um objeto particular 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 ilegal usar o método `add` para colocar algo nessa coleção a menos que seu tipo fosse `String`.

Os usuários podem definir classes genéricas em Java 5.0. Esse é um processo bastante direto e as classes resultantes se comportam exatamente como as classes genéricas pré-definidas.

11.5.4 C# 2005

Como no caso de Java, a primeira versão de C# definia classes de coleção que armazenavam objetos de qualquer classe. Elas eram `ArrayList`, `Stack` e `Queue`, que tinham os mesmos problemas que as classes de coleção de Java antes da versão 5.0.

As classes genéricas foram adicionadas em C# em sua versão 2005. As cinco coleções genéricas pré-definidas são `Array`, `List`, `Stack`, `Queue` e `Dictionary` (`Dictionary` implementa dispersões). Exatamente como em Java 5.0, essas classes eliminam os problemas de permitir tipos mistos em coleções e requerer 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 que C# 2005 não fornece são as classes coringa.

11.6 CONSTRUÇÕES DE ENCAPSULAMENTO

As primeiras cinco seções deste capítulo discutem tipos de dados abstratos, que são encapsulamentos mínimos³. 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 e subprogramas ou definições de tipos de dados abstratos não impõe um

³ No caso de Ada, o encapsulamento pacote pode ser usado para tipos individuais e também para múltiplos tipos.

nível adequado de organização no programa 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 caro. Para programas grandes, o custo da recompilação é significativo. Então, existe uma necessidade óbvia de encontrar maneiras para 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, cada uma podendo ser compilada sem a recompilação do resto do programa. Um **encapsulamento** é tal coleção.

Encapsulamentos são colocados em bibliotecas e disponibilizados para reuso em programas além daqueles para os quais eles foram escritos. As pessoas têm escrito programas com mais alguns milhares de linhas nos últimos 40 anos, então as técnicas para fornecer encapsulamentos têm evoluído.

11.6.2 Subprogramas aninhados

Em linguagens que permitem subprogramas aninhados, os programas podem ser organizados por definições de subprogramas aninhadas dentro de subprogramas maiores que os usam. Isso pode ser feito em Ada, Fortran 95, 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 uma construção de organização de encapsulamento primária.

11.6.3 Encapsulamento em C

C não fornece um suporte forte 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 esse arquivo, incluindo os dados, tipos e declarações de funções, é colocada em um separado chamado de **arquivo de cabeçalho**. Representações de tipo podem ser ocultas declarando-as no arquivo de cabeçalho como ponteiros para tipos `struct`. As definições completas para tais tipos `struct` precisam aparecer apenas no arquivo de implementação. Essa abordagem tem as mesmas desvantagens do uso de ponteiros como tipos de dados abstratos em pacotes Ada – ou seja, os problemas inerentes de ponteiros e a confusão em potencial com a atribuição e com as comparações de ponteiros.

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 usando 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 abordagem separa a especificação e a implementação de um encapsulamento.

Apesar de esses encapsulamentos funcionarem, eles criam algumasseguranças. 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 ele aparece. Entretanto, existem dois problemas com essa abordagem. Primeiro, a documentação de dependência entre o programa cliente e a biblioteca (e seu arquivo de cabeçalho) é perdida. Segundo, o autor da biblioteca pode modificar os arquivos de cabeçalho e os de implementação, mas o cliente poderia tentar usar o novo arquivo de implementação (sem se dar conta de que ele mudou), mas com o de cabeçalho antigo, o usuário havia copiado em seu programa cliente. 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` sendo um `int`, mas o código cliente foi compilado com `x` sendo um `float`. O ligador não detecta esse erro.

Então, é responsabilidade do usuário garantir que tanto o arquivo de cabeçalho quanto o de implementação estão atualizados. Em geral, isso é feito com um utilitário `make`.

11.6.4 Encapsulamento em C++

C++ fornece dois tipos de encapsulamento – arquivos de cabeçalho e de implementação podem ser definidos como em C, ou cabeçalhos de classes e definições podem ser definidas. Em função da complexa interação dos *templates* C++ e da compilação separada, os arquivos de cabeçalho das bibliotecas de *template* de C++ geralmente incluem a definição completa 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 contém *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 11.4.2.4. Isso separa claramente a interface da implementação.

Um problema de projeto de linguagem resultante de se ter classes, mas nenhuma construção generalizada de encapsulamento, é não ser sempre natural associar operações de objetos com objetos individuais. 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 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 { /** The class definition
    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.

11.6.5 Pacotes em Ada

Pacotes de especificação em Ada podem incluir qualquer número de declarações de dados ou de subprogramas em suas seções pública e privada. Logo, podem incluir interfaces para qualquer número de tipos de dados abstratos, assim como para quaisquer outros recursos de programas. Então, o pacote é uma construção de encapsulamento de múltiplos tipos.

Pacotes Ada podem ser compilados separadamente. As duas partes do pacote, especificação e corpo, também podem ser compiladas separadamente se o pacote de especificação for compilado primeiro. Um programa inteiro que use qualquer número de pacotes externos pode ser compilado separadamente, desde que as especificações de todos os pacotes usados tenham sido

compiladas. Os corpos dos pacotes usados podem ser compilados após o programa cliente.

Considere a situação descrita na Seção 11.6.4 dos tipos vetor e matriz e a necessidade para métodos com acesso às partes privadas de ambos, manipulada por meio de funções amigas em C++. Em Ada, tanto o tipo matriz quanto o tipo vetor podem ser definidos em um único pacote Ada, o que diminui a necessidade de funções amigas.

11.6.6 Montagens em C#

C# inclui uma construção de encapsulamento maior do que a classe. Nesse caso, a construção é aquela usada pelas linguagens de programação .NET: a **montagem (assembly)**, uma coleção de um ou mais arquivos que aparecem para os programas aplicativos como uma única biblioteca de ligação dinâmica ou um executável (EXE). Cada arquivo define um módulo, que pode ser desenvolvido separadamente. Uma **biblioteca de ligação dinâmica (DLL)** é uma coleção de classes e métodos individualmente ligados a um programa que está executando quando necessário durante a execução. Logo, apesar de um programa ter acesso a todos os recursos em uma DLL em particular, apenas as partes realmente usadas são carregadas e ligadas ao programa. As DLLs têm sido parte do ambiente de programação do Windows desde sua primeira aparição do sistema operacional. Além do código objeto para seus recursos, uma montagem .NET inclui um manifesto, com definições de topo para cada classe que ela contém, definições de outros recursos da montagem, uma lista de todas as montagens referenciadas na montagem e um número de versão da montagem.

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 ele aparece.

Como uma montagem é uma unidade executável autocontida, ela pode ter apenas um ponto de entrada.

11.7 NOMEANDO ENCAPSULAMENTOS

Temos considerado os encapsulamentos como contêineres sintáticos para recursos de software relacionados logicamente – em particular, tipos de dados abstratos. O propósito desses encapsulamentos é fornecer uma maneira de organizar programas em unidades lógicas para a compilação, permitindo partes de programas serem recompiladas após mudanças isoladas. Existe outro tipo de encapsulamento necessário para construir grandes programas: um encapsulamento de nomeação.

Um grande programa pode ser escrito por muitos desenvolvedores, trabalhando de maneira independente, talvez até em localizações geográficas diferentes. Isso requer que as unidades lógicas do programa sejam independentes, embora ainda assim seja possível trabalhar em conjunto. Isso também cria um problema de nomeação: como desenvolvedores trabalhando independentemente criam nomes para suas variáveis, métodos e classes sem acidentalmente usar nomes já em uso 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 necessitado de 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 alguma assistência linguística, isso é praticamente impossível, porque não existe uma maneira de o autor da biblioteca saber que 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 prevenir 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.

11.7.1 Espaços de nome em C++

C++ inclui uma especificação, `namespace`, que ajuda os programas a gerenciarem o problema de espaços de nome globais. Alguém pode 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 todas as declarações para a pilha em um bloco de espaço de nomes, como a seguir:

```
namespace MyStack {
    // 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

```
MyStack::topPtr
```

O arquivo de implementação também pode aparecer em uma especificação de bloco de espaço de nomes idêntica àquela usada no arquivo de cabeçalho, tornando todos os nomes declarados no arquivo de cabeçalho diretamente visíveis. Isso é definitivamente mais simples, mas levemente menos legível, porque é menos óbvio onde um nome específico no arquivo de implementação é declarado.

O código cliente pode ganhar acesso aos 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 `topPtr` poderia aparecer como

```
MyStack::topPtr
```

que é exatamente a maneira pela qual o código de implementação poderia referenciá-la.

As outras duas abordagens usam a diretiva `using`. Ela pode ser usada para qualificar nomes individuais de um espaço de nomes, como em

```
using MyStack::topPtr;
```

que torna `topPtr` visível, mas não quaisquer outros nomes do espaço de nomes `MyStack`.

A diretiva `using` pode também ser usada para qualificar todos os nomes de um espaço de nomes, como em:

```
using namespace MyStack;
```

Código que inclui essa diretiva pode acessar diretamente os nomes definidos no espaço de nomes, como em

```
p = topPtr;
```

Esteja ciente de que os espaços de nomes são um recurso complicado de C++, e que introduzimos apenas a parte mais simples da história aqui.

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 classe, e as classes em um pacote são amigas parciais umas das outras. *Parcial* aqui significa que as en-

tidades definidas em uma classe em um pacote públicas ou protegidas (veja o Capítulo 12) ou que não têm um especificador de acesso são visíveis para todas as outras classes no pacote. Um pacote pode ter apenas uma definição de classe pública*.

Entidades sem modificadores de acesso são ditas como tendo **escopo de pacote**, porque são visíveis ao longo do pacote. Java, dessa forma, tem menos necessidade para declarações amigas explícitas e não inclui as 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 myStack;
```

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 nomes definidos no pacote com nomes completamente qualificados. Por exemplo, se o pacote `myStack` define uma variável chamada `topPtr`, ela pode ser referenciada em um cliente de `myStack` como `myStack.topPtr`**. Como esse procedimento pode rapidamente se tornar desajeitado quando os pacotes são aninhados, Java fornece a declaração `import`, que permite referências mais curtas aos nomes definidos em um pacote. Por exemplo, suponha que o cliente inclua o seguinte:

```
import myStack.*;
```

Agora, a variável `topPtr`, assim como outros nomes definidos no pacote `myStack`, pode ser referenciada apenas por seus nomes. Para acessar apenas um nome do pacote, ele pode ser dado na declaração de importação, como em

```
import myStack.topPtr;
```

Note que o `import` de Java é apenas um mecanismo de abreviação. Nenhum outro recurso externo oculto é tornado 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`).

* N. de R. T.: Na verdade, uma unidade de compilação pode ter apenas uma definição de classe pública. Um pacote pode conter diversas definições de classes públicas.

** N. de R. T.: Na verdade, variáveis não podem ser referenciadas dessa forma, pois em Java não é possível definir variáveis globais. Normalmente, nomes de classes dentro dos pacotes são nomeados assim.

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 Pacotes em Ada

Pacotes em Ada, em geral usados para encapsular bibliotecas, são definidos em hierarquias, que correspondem às de arquivos nas quais eles são armazenados. Por exemplo, se `subPack` é um pacote definido como um filho do `pack`, o arquivo de código de `subPack` apareceria em um subdiretório do diretório que armazenou `pack`. As bibliotecas de classe padrão de Java também definidas em uma hierarquia de pacotes e são armazenadas em uma hierarquia de diretórios correspondente.

Como discutido na Seção 11.4.1, os pacotes também definem espaços de nomes. A visibilidade a um pacote a partir de uma unidade de programa é obtida com a cláusula `with`. Por exemplo,

```
with Ada.Text_IO;
```

torna os recursos e o espaço de nomes do pacote `Ada.Text_IO` disponível. Acessos aos nomes definidos no espaço de nomes de `Ada.Text_IO` devem ser qualificados. Por exemplo, o procedimento `Put` de `Ada.Text_IO` deve ser acessado como

```
Ada.Text_IO.Put
```

Para acessar os nomes em `Ada.Text_IO` sem qualificação, a cláusula `use` pode ser usada, como em

```
use Ada.Text_IO;
```

Com essa cláusula, o procedimento `Put` de `Ada.Text_IO` pode ser acessado simplesmente como `Put`. O `use` de Ada é similar ao `import` de Java.

11.7.4 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 de *módulo*, que normalmente define coleções de métodos e constantes. Então, os módulos 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 uma vez que não podem ser instanciados ou estendidos por herança e 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 um módulo:

```
module MyStuff
    PI = 3.14159265
    def MyStuff.mymethod1(p1)
        ...
    end
    def MyStuff.mymethod2(p2)
        ...
    end
end
```

Assumindo que o módulo `MyStuff` é armazenado em seu próprio arquivo, um programa que quer usar a constante e os métodos de `MyStuff` deve primeiro obter acesso ao módulo. Isso é feito com o método `require`, que recebe o nome do arquivo na forma de um literal da cadeia como parâmetro. Então, as constantes e métodos do módulo podem ser acessados pelo nome do módulo. 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 o ocultamento de informações. Uma linguagem pode suportar tipos de dados abstratos ou simulá-los com encapsulamentos mais gerais.

Ada fornece encapsulamentos chamados pacotes que podem ser usados para simular tipos de dados abstratos. Os pacotes normalmente têm duas partes: uma especificação, que apresenta a interface cliente, e um corpo, que fornece a implementação de um tipo de dados abstrato. Representações de tipos de dados podem aparecer no pacote de especificação, mas serem ocultas de clientes ao colocá-las na cláusula privada do pacote. O tipo abstrato propriamente dito é definido como privado na parte pública do pacote de especificação. Tipos privados têm operações pré-definidas para atribuição e comparação para igualdade e desigualdade.

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 apenas o protocolo dado na classe e a definição colocada em outro arquivo, que pode ser compilado separadamente. Classes em C++ têm três cláusulas, cada uma pré-fixada com um modificador de acesso: **private**, **public** ou **protected**. Tanto construtores quanto destrutores podem ser dados em definições de classes. Objetos alocados do monte devem ser explicitamente liberados com **delete**.

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. De modo contrário, 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 no sentido de que são dinâmicas – membros podem ser adicionados, apagados ou modificados durante a execução.

Ada, 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 de coleção.

Para suportar a construção de grandes programas, algumas linguagens contemporâneas incluem construções de encapsulamento de múltiplos tipos, que podem conter uma coleção de tipos relacionados logicamente. Um encapsulamento pode fornecer controle de acesso às suas entidades e um método de organizar programas facilitando a recompilação.

C++, C#, Java, Ada 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 por causa da disponibilidade dos pacotes, Java não tem funções ou classes amigas. Em Ada, os pacotes podem ser usados como encapsulamentos de nomeação.

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. Explique como o ocultamento de informações é fornecido por um pacote Ada.
7. Para quem a parte privada de um pacote de especificação em Ada é visível?
8. Qual é a diferença entre os tipos **private** e **limited private** em Ada?
9. O que é um pacote de especificação em Ada? E o que é um pacote de corpo?
10. Qual é o uso da cláusula **with** em Ada?

11. Qual é o uso da cláusula `use` em Ada?
12. Qual é a diferença fundamental entre uma classe C++ e um pacote em Ada?
13. De onde os objetos em C++ são alocados?
14. Em que diferentes locais a definição de uma função membro em C++ pode aparecer?
15. Qual é o propósito de um construtor em C++?
16. Quais são os tipos legais de retorno de um construtor?
17. Onde todos os métodos Java são definidos?
18. Como os objetos de classe em C++ são criados?
19. De onde os objetos de classe Java são alocados?
20. Por que Java não tem destrutores?
21. Onde as classes Java são alocadas?
22. O que é uma função amiga? O que é uma classe amiga?
23. Cite uma razão pela qual Java não tem funções amigas ou classes amigas.
24. Descreva as diferenças fundamentais entre structs e classes em C#.
25. Como um objeto struct é criado em C#?
26. Explique as três razões pelas quais os métodos de acesso para tipos privados são melhores do que tornar os tipos públicos.
27. Quais são as diferenças entre structs em C++ e structs em C#?
28. Porque Java não precisa de uma cláusula `use`, como em Ada?
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 genéricas de Ada são criadas?
32. Como as instâncias das classes *template* de C++ são criadas?
33. Descreva os dois problemas que aparecem na construção de grandes programas que levaram ao desenvolvimento de construções de encapsulamento.
34. Que problemas podem ocorrer ao usar C para definir tipos de dados abstratos?
35. O que é um espaço de nomes em C++ e qual é o seu propósito?
36. O que é um pacote Java e qual o seu propósito?
37. Descreva uma montagem no .NET.
38. Que elementos podem aparecer em um módulo Ruby?

CONJUNTO DE 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. Quais são as desvantagens de projetar um tipo de dados abstrato como um ponteiro?

5. Por que a estrutura de tipos de dados abstratos não ponteiros precisa ser dada nos pacotes de especificação de Ada?
6. Que perigos são evitados em Java e em C# ao ter coleta de lixo implícita, em relação a C++?
7. Discuta as vantagens das propriedades em C#, em relação a escrever métodos de acesso em C++ ou Java.
8. Explique os perigos da abordagem de C para encapsulamento.
9. Por que C++ não eliminou os problemas discutidos no Problema 8?
10. Por que os destrutores não são necessários em Java, mas 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 grandes programas.
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? Defenda sua resposta.
16. Quais são as vantagens e desvantagens da habilidade de modificar objetos em Ruby?
17. Compare os pacotes de Java com os módulos de Ruby.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Projete o exemplo do tipo abstrato pilha da Seção 11.2.3 em Fortran usando um único subprograma com múltiplas entradas para a definição de tipo e as operações.
2. Como a implementação do Exercício de Programação 1 em Fortran pode ser comparada à implementação de Ada deste capítulo em termos de confiabilidade e flexibilidade?
3. 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.
4. 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.
5. Modifique a classe C++ para o tipo abstrato mostrado na Seção 12.4.2 para usar uma representação de lista encadeada e teste-a com o mesmo código que aparece neste capítulo.
6. Modifique a classe Java para o tipo abstrato mostrado na Seção 12.4.3 para usar uma representação de lista encadeada e teste-a com o mesmo código que aparece neste capítulo.
7. 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 a construção de um número complexo a partir

de duas constantes, variáveis ou expressões de ponto flutuante. Use Ada, C++, Java, C# ou Ruby.

8. 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 Ada, C++, Java, C# ou Ruby.
9. Escreva um tipo de dados abstrato para uma fila cujos elementos possam ser de qualquer tipo primitivo. Use Java 5.0, C# 2005, C++ ou Ada.
10. Escreva um tipo de dados abstrato para uma fila cujos elementos incluem 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.
11. Um deque é uma fila de duas extremidades, com operações de adição e remoção de elementos em ambas. Modifique a solução do Problema 9 para implementar um deque.

Capítulo 12

Suporte para a Programação Orientada a Objetos

12.1 Introdução

12.2 Programação orientada a objetos

12.3 Questões de projeto para programação orientada a objetos

12.4 Suporte para programação orientada a objetos em Smalltalk

12.5 Suporte para programação orientada a objetos em C++

12.6 Suporte para programação orientada a objetos em Java

12.7 Suporte para programação orientada a objetos em C#

12.8 Suporte para programação orientada a objetos em Ada 95

12.9 Suporte para programação orientada a objetos em Ruby

12.10 Implementação de construções orientadas a objetos

Este capítulo começa com uma breve introdução à programação orientada a objetos, seguida por uma discussão sobre as questões de projeto primárias para herança e vinculação dinâmica. A seguir, o suporte para programação orientada a objetos em Smalltalk, C++, Java, C#, Ada 95 e Ruby são discutidos. O capítulo é concluído com uma visão geral sobre a implementação de vinculações dinâmicas de chamadas a métodos em linguagens orientadas a objetos.

12.1 INTRODUÇÃO

Linguagens que suportam programação orientada a objetos são agora bastante usadas. De COBOL até LISP, incluindo praticamente todas as linguagens entre elas, dialetos que suportam programação orientada a objetos apareceram. C++ e Ada suportam programação procedural e orientada a dados, além da programação orientada a objetos. CLOS, uma versão orientada a objetos de LISP (Bobrow et al., 1988) 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. Dentro dessas, estão Java e C#. Ruby é um tanto desafiadora para ser categorizada: 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 um tanto não convencional: Smalltalk, 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 tópico primário deste capítulo.

Em certo sentido, este capítulo é uma continuação do anterior. Esse relacionamento reflete a realidade que programação orientada a objetos é, em essência, uma aplicação do princípio de 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.

12.2 PROGRAMAÇÃO ORIENTADA A OBJETOS

12.2.1 Introdução

O conceito de **programação orientada a objetos** tem suas raízes no SIMULA 67, mas não foi completamente desenvolvido até a evolução do Smalltalk que resultou no 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 em vinculação dinâmica.

12.2.2 Herança

Há muito tempo existe uma pressão sobre os desenvolvedores de software para aumentarem sua produtividade. Essa pressão tem sido intensificada pela contínua redução no custo de hardware de computadores. No final dos anos 1980, tornou-se aparente para muitos desenvolvedores que uma das oportunidades mais promissoras para um aumento de produtividade em sua profissão era o reúso de software. Tipos abstratos de dados, com encapsulamento e controles de acesso, eram obviamente candidatos para reúso. O problema com o 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 requer ao menos algumas modificações, que podem ser difíceis, por exigirem que a pessoa responsável por elas entenda parte do código existente (ou todo ele). Além disso, em muitos casos as mudanças requerem modificações em todos os programas clientes.

Um segundo problema com a programação com tipos de dados abstratos é que as definições de tipos são independentes e no mesmo nível. Esse projeto normalmente torna impossível organizar um programa para casar com o espaço do problema que está sendo 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 reúso de tipos abstratos de dados quanto para o de organização de programas. Se um novo tipo abstrato de dados pode herdar os dados e funcionalidades de algum tipo existente, e também é permitido que ele modifique algumas das entidades e adicione novas, o reúso é amplamente facilitado sem requerer mudanças ao tipo abstrato de dados reutilizado. Os programadores podem começar com um tipo abstrato de dados 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.

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 é definida por meio de herança de outra classe é chamada de **classe derivada** ou **subclasse**. Uma classe da qual a nova

é derivada é sua **classe pai** ou **superclasse**. Os subprogramas que definem as operações em objetos de uma classe são chamados de **métodos**. As chamadas a métodos são algumas vezes chamadas de **mensagens**. A coleção completa de métodos de um objeto é chamada de **protocolo de mensagens** ou **interface de mensagens**. Computações em um programa orientado a objetos são especificadas por mensagens enviadas de objetos para outros ou, em alguns casos, para classes.

Existem diversas maneiras pelas quais uma classe derivada pode diferir de sua classe pai¹. Estas são as diferenças mais comuns entre uma classe pai e suas subclasses:

1. A classe pai pode definir alguns de seus membros como tendo acesso privado, fazendo com que esses métodos não sejam visíveis na subclasse.
2. A subclasse pode adicionar membros àqueles herdados da classe pai.
3. 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. É dito que o novo método **sobrescreve** o método herdado, chamado então de **método sobrescrito**. O propósito mais comum de um método sobrescrever outro é para fornecer uma operação específica para objetos da classe derivada, mas não é apropriado para objetos da classe pai.

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 instância². 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. **Métodos de classe** podem realizar operações na classe e também em objetos 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.

Uma desvantagem da herança como forma de aumentar a possibilidade de reuso é que ela cria dependências entre classes em uma hierarquia. Isso vai contra uma das vantagens dos tipos abstratos de dados, a independência de

¹ Se uma subclasse não difere de sua classe pai, ela obviamente não serve a propósito algum.

² Isso não é verdade em Ruby, que permite diferentes objetos da mesma classe serem diferentes de outras maneiras.

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 reusabilidade 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.

12.2.3 Vinculação dinâmica

A terceira característica (além dos tipos de dados abstratos e da herança) das linguagens de programação orientadas a objetos é um tipo de polimorfismo fornecido pela vinculação dinâmica de mensagens às definições de métodos. Considere a seguinte situação: existe uma classe base A, que define um método que realiza uma operação em objetos da classe base. Uma segunda classe, B, é definida como uma subclasse de A. Objetos dessa nova classe precisam de uma operação parecida com a fornecida por A, mas um pouco diferente porque eles são levemente diferentes. Então, a subclasse sobrescreve o método herdado. Se um cliente de A e B tem uma referência ou um ponteiro para objetos da classe A, essa referência ou ponteiro também pode apontar para objetos da classe B, tornando-a uma referência ou um ponteiro **polimórfico**. Se o método, definido em ambas as classes, é chamado pela referência ou pelo ponteiro polimórfico, 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 o tipo do objeto atualmente referenciado pelo ponteiro ou pela referência). Um exemplo de vinculação dinâmica em C++ é dado na Seção 12.5.3.

Um propósito para essa vinculação dinâmica é permitir que os sistemas de software sejam melhor estendidos durante o desenvolvimento e a manutenção. Por exemplo, suponha que uma classe defina objetos que representam pássaros e suas subclasses definam pássaros específicos. Além disso, que cada subclasse redefina um método herdado de sua classe base que mostre seu pássaro específico. Esses métodos podem ser todos chamados por uma referência ou um ponteiro para a classe base que representa pássaros. Então, se o código cliente criasse um vetor de ponteiros da classe base que referenciasse objetos das subclasses de pássaros específicos e precisasse mostrar cada pássaro referenciado no vetor, o método *mostrar* para cada pássaro seria chamado com a mesma chamada (em um laço) por meio dos ponteiros da classe base no vetor. Uma vantagem disso para a manutenção é que adicionar uma nova subclasse (para um pássaro novo no sistema) não requereria uma mudança no código que chamou os métodos para mostrar informações sobre os pássaros.

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 clas-

se de construção (chamada `building`) e uma coleção de subclasses para tipos específicos de construções, como francesa gótica – `French_Gothic`. Provavelmente, não faria sentido ter um método de desenho (`draw`) implementado em `building`. Mas, como todas as suas classes descendentes devem ter tal método implementado, o protocolo (mas não o corpo) dele é incluído em `building`. Esse método é chamado de **método abstrato** (*método puramente virtual* em C++). Uma classe que inclui ao menos um método abstrato é chamada de **classe abstrata** (*classe base abstrata* em C++), que normalmente não pode ser instanciada, porque alguns de seus métodos são declarados, mas não são definidos (eles não têm corpos). Qualquer subclass de uma classe abstrata a ser instanciada deve fornecer implementações (definições) para todos os métodos abstratos herdados.

12.3 QUESTÕES DE PROJETO PARA PROGRAMAÇÃO ORIENTADA A OBJETOS

Diversas questões devem ser consideradas quando estamos projetando 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 absorve todos os outros conceitos de tipo. Tudo, desde o menor inteiro até um sistema de software completo, é um objeto em seu ponto de vista. 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, que em geral as torna mais lentas do que operações similares em um modelo imperativo, no qual instruções únicas de máquina implementam tais operações simples. Por exemplo, em Smalltalk, a adição de 7 a uma variável chamada `x` ocorre pelo envio do objeto 7 como um parâmetro para o método `+` do objeto `x`. Nesse modelo mais puro de computação orientada a objetos, todos os tipos são classes. Não existe distinção entre classes pré-definidas 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 adicionada é manter um modelo de tipos completamente imperativo e simplesmente adicionar o modelo de objetos. Essa abordagem resulta em uma linguagem maior, cuja estrutura de tipos é confusa para todos os usuários, exceto para os especialistas na 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 a velocidade de operações em valores primitivos comparável àquelas esperadas no modelo imperativo. Infelizmente, essa alternativa também leva a complicações na linguagem. Invariavelmente, valores que não são objetos precisam ser misturados com eles. Isso cria uma necessidade para as classes chamadas de *classes wrapper* para os tipos não objeto, de forma que algumas operações comumente necessárias podem ser enviadas a objetos com valores de tipos não objeto. Na seção 12.6.1, discutimos um exemplo de seu uso em Java. Esse projeto é uma troca de uniformidade de linguagem e pureza por eficiência.

12.3.2 As subclasses são subtipos?

A questão aqui é relativamente simples: um relacionamento “é-um(a)” se mantém entre uma classe derivada e sua classe pai? De um ponto de vista puramente semântico, se uma classe derivada é – um(a) classe pai, os objetos da classe derivada devem expor todos os membros expostos por objetos da classe pai. Em um nível menos abstrato, um relacionamento é-um(a) garante que em um cliente uma variável do tipo da classe derivada pode aparecer em qualquer lugar onde uma variável do tipo da classe pai seria legal, sem causar um erro de tipos. E mais, os objetos da classe derivada devem ter comportamento equivalente ao dos objetos da classe pai.

Os subtipos de Ada são exemplos dessa forma simples de herança para dados. Por exemplo,

```
subtype Small_Int is Integer range -100..100;
```

Variáveis do tipo `Small_Int` têm todas as operações de variáveis do tipo `Integer`, mas podem armazenar apenas um subconjunto dos valores possíveis em `Integer`. Além disso, toda variável `Small_Int` pode ser usada em qualquer lugar onde uma variável `Integer` pode ser usada. Ou seja, toda variável `Small_Int` é, em um certo sentido, uma variável `Integer`.

Existe uma ampla variedade de maneiras pelas quais uma subclasse pode diferir de sua classe pai ou classe base. Por exemplo, a subclasse pode ter métodos adicionais, menos métodos, tipos de alguns dos parâmetros diferentes em um ou mais métodos, tipo de retorno de algum método diferente, número de parâmetros de algum método diferente ou corpo de um ou mais métodos diferente. A maioria das linguagens de programação restringe severamente as maneiras pelas quais uma subclasse pode diferir de sua classe base. Na maioria dos casos, as regras de linguagem restringem a subclasse a ser um subtipo de sua classe pai.

Conforme mencionado, uma classe derivada é chamada de um **subtipo** se tem um relacionamento é-um(a) com sua classe pai. As características de

uma subclasse para garantir que ela é um subtipo são: os métodos da subclasse que sobrescrevem métodos da classe pai devem ser compatíveis em relação ao tipo com seus métodos sobrescritos correspondentes. *Compatível* aqui significa que uma chamada a um método sobrescrevedor pode substituir qualquer chamada ao método sobrescrito em qualquer aparição no programa cliente sem causar erros de tipo. Isso significa que todo método sobrescrevedor deve ter o mesmo número de parâmetros do método sobrescrito e do tipo dos parâmetros, e o tipo de retorno deve ser compatível com o da classe pai. Ter um número idêntico de parâmetros, de tipos de parâmetros e de tipo de retorno garantiria a conformidade de um método. Restrições menos severas são possíveis, entretanto, dependendo das regras de compatibilidade de tipos da linguagem.

Nossa definição de subtipo claramente proíbe haver entidades públicas na classe pai que não são públicas também na subclasse. Então, o processo de derivação para subtipos deve requerer que entidades públicas da classe pai sejam herdadas como entidades públicas na subclasse.

Pode parecer que relacionamentos de subtipos e de herança são praticamente idênticos. Entretanto, essa conjectura está longe de ser correta. Uma explicação dessa suposição incorreta, com um exemplo em C++, é dada na Seção 12.5.2.

12.3.3 Verificação de tipos e polimorfismo

Na Seção 12.2, o polimorfismo no mundo da orientação a objetos é definido como o uso de um ponteiro ou referência polimórfica para acessar um método cujo nome é sobrescrito na hierarquia de classes que define o objeto para ao qual o ponteiro ou referência apontam. A variável polimórfica é o tipo da classe base, e esta define ao menos o protocolo de um método que é sobrescrito pelas classes derivadas. A variável polimórfica pode referenciar objetos da classe base e de classes descendentes, então a classe do objeto para o qual ela aponta nem sempre pode ser determinada estaticamente. A vinculação de mensagens a métodos enviadas por variáveis polimórficas deve ser dinâmica. A questão aqui é quando ocorre a verificação de tipos dessa vinculação.

Essa questão é importante, visto que está alinhada à natureza fundamental da linguagem de programação. Seria melhor se essa verificação de tipos pudesse ser feita estaticamente, porque a verificação de tipos dinâmica custa tempo de execução e posterga a detecção de erros de tipo. Requerer verificação estática de tipos força a algumas restrições importantes no relacionamento entre mensagens polimórficas e métodos.

Existem duas formas de verificação de tipos que devem ser feitas entre uma mensagem e um método em uma linguagem fortemente tipada: os tipos dos parâmetros da mensagem devem ser verificados em relação aos parâmetros formais do método, e o tipo de retorno do método deve ser verificado em relação ao tipo esperado da mensagem. Se esses tipos devem

casar exatamente, um método sobrescrevedor deve ter o mesmo número de parâmetros, e os mesmos tipos de parâmetros e de retorno que o método sobrescrito. Um relaxamento dessa regra pode ser permitir compatibilidade de atribuição entre parâmetros reais e formais e entre o retornado e o esperado pela mensagem.

A alternativa óbvia à verificação de tipos estática é prorrogar a verificação de tipos até que a variável polimórfica seja usada para chamar um método.

12.3.4 Herança simples e múltipla

Outra questão simples é: a linguagem permite herança múltipla (além da simples)? Talvez não seja tão simples. 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, porque 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 definem um método herdável chamado `display`. Se `C` precisar referenciar ambas as versões de `display`, como isso pode ser feito? Esse problema de ambiguidade é ainda mais complicado quando as duas classes pais definem métodos nomeados de forma idêntica e um deles ou ambos precisam ser sobrescritos 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 pai. 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 herdável 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. A Seção 12.10 inclui um breve relance na implementação dessas situações. A herança diamante é mostrada na Figura 12.1.

A questão da eficiência pode ser mais aparente do que real. Em C++, por exemplo, suportar herança múltipla requer apenas mais um acesso a ma-

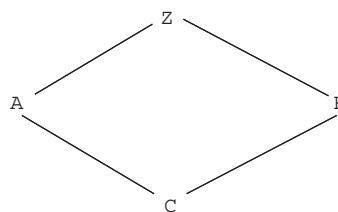


Figura 12.1 Um exemplo de herança diamante.

triz 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. A manutenção de sistemas que usam herança múltipla pode ser um problema mais sério, já que a herança múltipla leva a dependências mais complexas entre as classes. Não está claro para alguns que os benefícios da herança múltipla valem a pena o esforço adicional de projetar e manter um sistema que a usa.

12.3.5 Alocação e liberação de objetos

Existem duas questões de projeto relacionadas à alocação e à liberação de objetos. A primeira é o local onde os objetos são alocados. Se eles se comportam como tipos de dados abstratos, talvez possam 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 ter 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 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, são variáveis de valor nas quais o valor do objeto deve ser copiado para o espaço do objeto alvo. Se `B` adiciona um atributo 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.

A segunda questão se preocupa com casos nos quais os objetos são alocados do monte. A questão é se 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, lança a questão acerca da possibilidade ou não da criação de ponteiros soltos ou referências soltas.

12.3.6 Vinculação estática e dinâmica

Como já discutimos, a vinculação dinâmica de mensagens a métodos em uma hierarquia de herança 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?

12.3.7 Classes aninhadas

Uma das motivações primárias para classes aninhadas é o ocultamento de informação. Se uma nova classe é necessária para apenas uma classe, não existe uma razão para defini-la de forma que seja vista por outras. Nessa situação, a nova classe pode ser aninhada dentro da classe que a usa. 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 com aninhamento de classes são relacionadas à visibilidade. Especificamente, uma questão é: quais dos recursos da classe aninhadora são visíveis para a classe aninhada? A outra questão principal é o inverso: quais dos recursos da classe aninhada são visíveis para a classe aninhadora?

12.3.8 Inicialização de objetos

A questão da inicialização diz respeito a se e como os objetos são inicializados para valores ao serem criados. Isso é mais complicado do que pode parecer. A primeira 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 SMALLTALK

Muitos veem Smalltalk como a linguagem de programação orientada a objetos definitiva. Ela foi a primeira a incluir suporte completo para esse paradigma. Logo, é natural começar um *survey* de suporte linguístico a programação orientada a objetos com Smalltalk.

12.4.1 Características gerais

Um programa em Smalltalk consiste inteiramente em objetos, então, o conceito de um objeto é universal. Praticamente tudo, desde itens simples como a constante inteira 2 até um sistema complexo de manipulação de arquivos, são objetos – e 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.

Mensagens podem ser parametrizadas com variáveis que referenciam objetos. Respostas a mensagens têm a forma de objetos e são usadas para retornar informações solicitadas ou apenas para confirmar que o serviço solicitado foi completado.

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. Toda a 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.

Diferentemente de linguagens híbridas como C++ e Ada 95, Smalltalk foi projetada para apenas um paradigma de desenvolvimento de software – orientado a objetos. Além disso, ela não adita nenhuma das aparências das linguagens imperativas. Sua pureza de propósito é refletida em sua elegância simples e uniformidade de projeto.

12.4.2 Verificação de tipos e polimorfismo

A vinculação dinâmica de mensagens a métodos em Smalltalk opera da seguinte forma: uma mensagem para um objeto inicia uma busca na classe à qual o objeto pertence para achar um método correspondente. Se a busca falha, ela continua na superclasse dessa classe, 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 diferente de verificação de tipos do 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 o 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 para o qual a variável está vinculada.

O principal ponto dessa 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.

12.4.3 Herança

Uma subclasse Smalltalk herda todas as variáveis de instância, métodos de instância e métodos de classe 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 ao 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 as entidades em uma classe pai não podem ser ocultadas a partir das subclasses, todas as subclasses são subtipos. Um relacionamento é-um(a) se mantém entre todos os objetos da subclasse e um objeto de sua classe pai.

Smalltalk suporta herança simples, mas não herança múltipla.

12.4.4 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 ela for construída em torno de um conceito simples, mas poderoso. No caso de Smalltalk, o conceito é que toda a programação pode ser feita se empregando apenas uma hierarquia de classes construída usando herança, objetos e passagem de mensagens.

Em comparação com 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, a eficiência será uma questão a ser tratada na maioria das discussões acerca da aplicabilidade prática de Smalltalk.

A vinculação dinâmica de Smalltalk permite que erros de tipo passem sem ser detectados até a execução. Um programa pode ser escrito e compilado incluindo mensagens a métodos inexistentes, causando muito mais erros e reparos posteriormente no desenvolvimento do que ocorreria em uma linguagem estaticamente tipada.

De um modo geral, o projeto de Smalltalk piora ao optar pela elegância da linguagem e a aderência estrita aos princípios do suporte a programação orientada a objetos, geralmente sem levar em consideração questões práticas, em particular a eficiência de execução. Isso é mais óbvio no uso exclusivo de objetos e variáveis sem tipos.

A interface com o usuário de Smalltalk tem tido 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, atualmente a metodologia de projeto e codificação mais amplamente usada.

12.5 SUPORTE PARA PROGRAMAÇÃO ORIENTADA A OBJETOS EM C++

O Capítulo 2 descreve como C++ evoluiu de C e SIMULA 67, com o objetivo de projeto de suportar a programação orientada a objetos. Classes C++, e como elas são usadas para suportar tipos de dados abstratos, são discutidas no Capítulo 11. O suporte a 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 grande e complexa. Esta seção foca apenas nos tópicos mais importantes, especificamente aqueles 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 dentre as mais populares. Então, naturalmente, é a aquela com a qual as outras linguagens são geralmente comparadas. Por essas razões, nossa cobertura de C++ é mais detalhada do que as outras linguagens de exemplo neste capítulo.

12.5.1 Características gerais

Como uma das considerações de projeto primárias de C++ era de que deveria ser compatível com C, ela retém o sistema de tipos de C e adiciona classes a ele. Logo, C++ tem tanto tipos tradicionais de linguagens imperativas quanto as estruturas de classes de uma orientada a objetos. C++ também suporta tanto métodos quanto funções não relacionadas a classes específicas. Isso a torna uma linguagem híbrida, suportando programação procedural e programação orientada a objetos.

Os objetos de C++ podem ser estáticos, dinâmicos da pilha ou dinâmicos do monte. A liberação explícita usando o 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.

12.5.2 Herança

Uma classe C++ pode ser derivada de uma classe existente — sua classe pai ou classe base. Diferentemente de Smalltalk, uma classe C++ também pode ser autocontida, sem uma superclasse.

Lembre que os dados em uma definição de classe são chamados de *membros de dados*, e as funções são chamadas de *funções membro* (funções membros em outras linguagens são geralmente chamadas de métodos). Alguns ou todos os membros de dados e as funções membro da classe base podem ser herdados pela classe derivada, que pode também adicionar novos membros de dados e funções membro 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 qualquer um dos membros de dados são ponteiros para dados alocados no monte, o construtor aloca o armazenamento.

Se uma classe tem um pai, os membros de dados herdados devem ser inicializados quando o objeto da subclasse é criado. Para tanto, o construtor pai é chamado implicitamente. Quando os dados de inicialização devem ser fornecidos para o construtor pai, esses dados são passados na chamada para o construtor do objeto da subclasse. Em geral, isso é feito por meio da seguinte construção:

```
subclasse(parâmetros da subclasse): classe_pai(parâmetros da superclasse){  
    ...  
}
```

Se nenhum construtor é incluído em uma definição de classe, o compilador inclui um construtor trivial. Esse construtor padrão chama o construtor da classe pai, se ela existir.

Lembre, do Capítulo 11, que membros de classe podem ser privados, protegidos ou públicos. Membros privados são acessíveis apenas por funções membros e por amigos da classe. Tanto funções quanto classes podem ser declaradas como amigas de uma classe e terem acesso aos membros privados de tal classe. Membros públicos são visíveis em todos os lugares. Membros protegidos são parecidos com 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 de uma classe derivada é

```
class nome_classe_derivada: modo_de_derivação nome_classe_base  
{declarações de membros de dados e de funções membro};
```



Sobre paradigmas e uma programação melhor

BJARNE STROUSTRUP

Bjarne Stroustrup é o projetista e implementador original de C++ e autor de *The C++ Programming Language* e *The Design and Evolution of C++*. Seus interesses em pesquisa incluem sistemas distribuídos, simulação, projeto, programação e linguagens de programação. Dr. Stroustrup é professor de Ciência da Computação na Faculdade de Engenharia da Universidade do Texas A&M e está ativamente envolvido na padronização ANSI/ISO do C++. Após mais de duas décadas na AT&T, mantém uma ligação com os laboratórios da companhia, pesquisando como membro do Laboratório de Pesquisa em Informação e Sistemas de Software. Bjarne é ACM Fellow, AT&T Bell Laboratories Fellow e AT&T Fellow. Em 1993, recebeu o Prêmio Grace Murray Hopper da ACM "por seu trabalho pioneiro que levou às bases da linguagem de programação C++". Através dessas bases e dos esforços contínuos do Dr. Stroustrup, C++ tornou-se uma das linguagens de programação mais influentes na história da computação."

PARADIGMAS DE PROGRAMAÇÃO

Quai as vantagens e desvantagens do paradigma orientado a objetos? 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 hierarquias de classes que fornecem comportamento polimórfico por meio de algum equivalente aproximado às funções virtuais. Para 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 naturalmente 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 dependendo de polimorfismo paramétrico (em tempo de compilação) é uma abordagem 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 essas. A combinação desses paradigmas define a programação multiparadigma,

que tem como característica o suporte a mais de um estilo de programação (“paradigma”) e suas combinações.

Você tem um minrexemplo 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 container genérico `vector` com o tipo polimórfico `Shape`. Ele fornece a segurança de tipagem estática e desempenho em tempo de execução ótimo. O tipo `Shape` fornece a habilidade de manipular uma forma (ou seja, qualquer objeto de uma classe derivada a partir de `Shape`) sem recomilação.

Podemos facilmente generalizar esse procedimento para qualquer container 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);
```

ESCOLHENDO A LINGUAGEM “CERTA” PARA A TAREFA

É útil ter experiência com vários paradigmas ou seria melhor investir tempo em conhecer profundamente linguagens OO apenas? 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. Entretanto, não é recomendável conhecer apenas C++, muito menos saber apenas uma linguagem de único paradigma. Seria como ter cegueira para cores ou ser monoglotá: você dificilmente saberia o que está 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 educação sólida e ampla, e não para pessoas com um “treinamento” restrito e rápido.

O modo de derivação pode ser **public** ou **private**³. (Não confunda a 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, tanto membros públicos quanto protegidos da classe base são privados. Portanto, em uma hierarquia de classes, uma classe privadamente derivada corta o acesso a todos os membros de todas as classes ancestrais e todas as classes sucessoras, e os membros protegidos podem ou não ser acessíveis às classes subsequentes. Membros privados de uma classe base são herdados por uma classe derivada, mas não são visíveis aos membros de tal classe derivada e logo não tem utilidade alguma lá. 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 { ... };
```

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 públicos, 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 ilegais no objeto da subclasse. O relacionamento é-um(a) seria quebrado. 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;
    ...
}
```

³ Ele pode ser também **protected**, mas essa opção não é discutida aqui.

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, que especifica a classe onde a entidade seguinte é definida.

O exemplo nos parágrafos 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 à classe aninhadora, `single_linked_list`. Se fossem privados, `node` precisaria declarar a classe aninhadora como a amiga para torná-los visíveis na classe aninhadora. Note que classes aninhadas não têm acesso especial aos membros da classe aninhadora. Apenas membros de dados estáticos da classe aninhadora são visíveis aos métodos da classe aninhada⁴.

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, que simplesmente configura `head` para ter o valor do ponteiro nulo. As quatro funções membro permitem que nós sejam inseridos em qualquer uma das extremidades de um objeto lista e removidos de um das extremidades da lista, e que as listas sejam testadas para ver se estão vazias.

As seguintes definições fornecem classes para pilha e fila, ambas baseadas na classe `single_linked_list`:

⁴ Uma classe também pode ser definida em um método de uma classe aninhadora. As regras de escopo de tais classes são as mesmas das regras para classes aninhadas diretamente em outras classes, mesmo para as variáveis locais declaradas no qual elas são definidas.

```
class stack : public single_linked_list {
public:
    stack() {}
    void push(int value) {
        single_linked_list :: insert_at_head(value);
    }
    int pop() {
        return single_linked_list :: remove_at_head();
    }
};

class queue : public single_linked_list {
public:
    queue() {}
    void enqueue(int value) {
        single_linked_list :: insert_at_tail(value);
    }
    int dequeue() {
        single_linked_list :: 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 das subclasses é criado, o construtor adequado na subclasse é implicitamente chamado, ou seja, o mesmo acontece com qualquer construtor aplicável na classe base. Em nosso exemplo, quando um objeto do tipo `stack` é criado, o construtor `stack` é chamado não fazendo nada. Logo, o construtor em `single_linked_list` é chamado, e faz a inicialização necessária.

As classes `stack` e `queue` sofrem do mesmo problema sério: clientes de ambas podem acessar todos os membros públicos da classe pai, `single_linked_list`. Um cliente de um objeto poderia chamar `insert_at_tail`, destruindo a integridade 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 alguém quer que a subclasse herde a interface inteira da classe base. A alternativa é permitir derivação na qual a subclasse herde apenas a implementação da classe base. Nos nossos dois exemplos, as classes derivadas podem ser escritas para torná-las não subtipos de sua classe pai usando derivação **privada** em vez de **pública**⁵. Então, ambas também precisarão reexportar `empty`, porque ela se tornará oculta para suas instâncias. Essa situação ilustra a necessidade para a opção de derivação pri-

⁵ 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.

vada. 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();
};
```

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. Entretanto, nenhuma delas é uma subclasse da classe lista encadeada. Em vez disso, a lista encadeada é na verdade um subtipo das duas.

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. Por exemplo, suponha que um programa use uma classe para vetores e outra para matrizes, e um subprograma precise multiplicar um objeto vetor por um objeto matriz. Em C++, a função de multiplicação simplesmente é tornada amiga de ambas as classes.

C++ fornece herança múltipla, permitindo mais de uma classe ser chamada de pai de uma nova classe. Por exemplo,

```
class A { ... };
class B { ... };
class C : public A, public B { ... };
```

A classe `C` herda todos os membros de `A` e de `B`. Se tanto `A` quanto `B` incluem membros com o mesmo nome, eles podem ser referenciados de ma-

neira não ambígua em objetos da classe C usando o operador de resolução de escopo (::). Alguns problemas com a implementação de C++ de herança múltipla são discutidos na Seção 12.10.

Métodos que sobrescrevem em C++ devem ter exatamente o mesmo perfil de parâmetros que o método sobreescrito. Se existir qualquer diferença nos perfis de parâmetros, o método na subclasse é considerado um novo método não relacionado àquele com o mesmo nome na classe ancestral. O tipo de retorno do que sobreescrava deve ser o mesmo do sobreescrito ou ser um tipo derivado publicamente do tipo de retorno do sobreescrito.

12.5.3 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 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 é necessária. Por outro lado, uma variável ponteiro que tem o tipo de uma classe base pode ser usada para apontar para quaisquer objetos dinâmicos do munte de qualquer classe publicamente derivada a partir dessa classe base, tornando-a uma variável polimórfica. Subclasses derivadas privadamente não 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 sobreescrita em uma das classes derivadas, a chamada deve ser dinamicamente vinculada à definição de função membro correta. Funções membro a serem dinamicamente vinculadas devem ser declaradas como funções virtuais precedendo seus cabeçalhos com a palavra reservada **virtual**, podendo 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 para a situação de exemplo descrita:

```
public class shape {
    public:
        virtual void draw() = 0;
    ...
}
```

```

public class circle : public shape {
    public:
        void draw() { ... }
        ...
}
public class rectangle : public shape {
    public:
        void draw() { ... }
        ...
}
public 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;           // Now ptr_shape points to a
                         // square object
ptr_shape->draw();      // Dynamically bound to the draw
                         // in the square class
rect->draw();           // Statically bound to the draw
                         // in the rectangle class

```

Note que a função `draw` na definição da classe base `shape` é configurada para 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 se chamam essa função. O propósito de uma função virtual pura é fornecer 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. Tal situação foi discutida na Seção 12.2.3.

Qualquer classe que inclua uma função virtual pura é uma **classe abstrata**. É ilegal instanciar uma classe abstrata. Em um sentido estrito, uma classe abstrata é uma 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 e a subclasse é também 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 de todas ou de qualquer uma das versões de `draw` serem escritas. Novas classes derivadas podem ser adicionadas anos depois, sem requerer mudança no código que usa tais membros vinculados dinamicamente. Esse é um recurso altamente 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 ao referenciado por `rect`, mas `rect` ainda assim referencia 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 o `rect` apontar para o objeto `square`, e uma chamada a `draw` por meio de `rect` seria vinculada dinamicamente à `draw` no objeto `square`.

12.5.4 Avaliação

É natural comparar os recursos orientados a objetos de C++ com aqueles de Smalltalk. A herança de C++ é mais intrincada do que a de Smalltalk em termos de controle de acesso. Ao usar tanto os controles de acesso dentro da definição de classe quanto os controles de acesso 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. Além disso, apesar de existir algum debate acerca de seu real valor, C++ fornece herança múltipla, enquanto Smalltalk permite apenas herança simples.

Em C++, o programador pode especificar se a vinculação estática ou dinâmica deve ser usada. Como a vinculação estática é mais rápida, isso é uma vantagem para aquelas situações nas quais a vinculação dinâmica não é necessária. Além disso, mesmo 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 requerem apenas cinco referências de memória a mais do 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 podem ter de ser trocadas posteriormente.

A verificação de tipos estática de C++ é uma vantagem sobre Smalltalk, onde toda a verificação de tipos é dinâmica. Um programa Smalltalk pode ser compilado com mensagens para métodos inexistentes, os quais não são descobertos até que o programa seja executado. Um compilador C++ acha tais erros, que são menos caros de reparar do que aqueles achados durante testes.

Smalltalk é essencialmente uma linguagem desprovida de tipos, ou seja, todo o código é efetivamente genérico. Isso fornece uma boa dose de flexibilidade, mas com um sacrifício 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 vantagem primária de Smalltalk está na elegância e na simplicidade da linguagem, que resulta da filosofia única de seu projeto. Ela é pura e completamente devotada ao paradigma orientado a objetos. C++, por outro lado, é uma linguagem grande e complexa sem uma filosofia única como base, exceto o suporte à programação orientada a objetos e a inclusão da base de usuários C. Um de seus objetivos mais significativos foi preservar a eficiência e o sabor de C enquanto fornece as vantagens da programação orientada a objetos. Algumas pessoas acham que os recursos dessa linguagem nem sempre se encaixam bem juntos e que ao menos parte de sua complexidade é desnecessária.

De acordo com Chambers e Ungar (1991), Smalltalk rodou um conjunto particular de pequenos *benchmarks* no estilo de C em apenas 10% da velocidade de C otimizado. Os programas em C++ requerem apenas um pouco mais de tempo do 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 do que o de Smalltalk. É claro, existem outros fatores nessa diferença, mas a eficiência é um forte argumento a favor de C++.

12.6 SUPORTE PARA PROGRAMAÇÃO ORIENTADA A OBJETOS EM JAVA

Como o projeto de classes, herança e métodos em Java é similar ao de C++, nesta seção focamos apenas nas áreas nas quais Java difere de C++.

12.6.1 Características gerais

Como em 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 ter não objetos é a eficiência. No entanto, conforme discutido na Seção 12.3.1, ter dois sistemas de tipos leva a algumas situações difíceis de tratar. Uma delas, em Java, é que as classes contêm pre-definidas, como `ArrayList`, podem conter apenas objetos. Para colocar um valor primitivo em um `ArrayList`, o valor deve primeiro ser colocado em um objeto. Em versões de Java anteriores à 5.0, isso pode ser feito por meio da criação de um novo objeto da classe *wrapper* para o tipo primitivo. Tal classe tem uma variável de instância do tipo primitivo e um construtor que recebe um parâmetro do tipo primitivo e o atribui para sua variável de instância. Por exemplo, para colocar 10 no objeto `ArrayList` referenciado pela variável `myArray`, a seguinte sentença poderia ser usada:

```
myArray.add(new Integer(10));
```

onde `add` é um método de `ArrayList` que insere um novo elemento e `Integer` é a classe *wrapper* para `int`. Além disso, quando um valor é removido de `myArray` e atribuído à uma variável `int`, ele deve ser convertido de volta para o tipo `int`.

Essa situação é aliviada em Java 5.0 pela coerção implícita de valores primitivos quando são colocados no contexto de objetos, sendo enviados como um parâmetro para o método `add` de `ArrayList`. Essa coerção converte o valor primitivo para 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*). Por exemplo, em Java 5.0, o trecho a seguir é permitido

```
myArray.add(10);
```

O compilador fabrica o encaixotamento do valor `int` para um objeto `Integer`. Além do encaixotamento, quando um elemento é removido de `myArray` e atribuído para uma variável `int`, ele é implicitamente convertido para `int`.

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 de `Object`. Uma razão para ter uma única classe raiz é que existem algumas operações universalmente necessárias. Dentre elas, está um método para comparar objetos em relação à sua igualdade.

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 coleção de lixo é usada para recuperação de armazenamento. Como muitos outros recursos de linguagem, apesar da 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 *lock* em um recurso compartilhado, o coletor de lixo não recupera esses recursos. Para tais situações, Java permite a inclusão de um método especial, **finalize**, relacionado com 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 com **finalize** é que o momento no qual ele será executado não pode ser forçado, nem previsto. A alternativa ao uso de **finalize** para recuperar recursos usados por um objeto que está por ser coletado como lixo é incluir um método que faz a recuperação. O único problema com isso é que todos os clientes dos objetos devem estar cientes desse método e lembrarem de chamarem-no.

12.6.2 Herança

Em Java, um método pode ser definido como **final**, ou seja, não pode ser sobreescrito em nenhuma classe descendente. Quando a palavra reservada **final** é especificada em uma definição de classe, significa que ela não pode ser pai de nenhuma subclasse. Também significa que as vinculações de chamadas a métodos da subclasse podem ser estaticamente vinculadas.

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, tal construtor 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 suporta diretamente apenas herança simples. Entretanto, ela inclui um tipo de classe abstrata, chamada de **interface**, que fornece suporte parcial para herança múltipla. 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 ou métodos não abstratos. Então, uma interface nada mais é 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 ou mais). Uma classe não herda de uma interface; ela a implementa. Na verdade, uma classe pode implementar qualquer número de interfaces. Para implementar uma interface, a classe deve implementar todos os métodos cujas especificações (mas não os corpos) aparecem na definição desta.

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. Isso é chamado algumas vezes 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 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 com herança múltipla ocorre quando uma classe é derivada de duas classes pai e ambas definem um método público com o mesmo nome e protocolo. Esse problema é evitado com interfaces, porque uma classe que implementa uma interface deve fornecer definições para *todos* os métodos especificados nela. Então, se tanto a classe pai quanto a interface incluírem métodos com o mesmo nome e protocolo, a subclasse deve reimplementar esse método. Os conflitos de nomes que podem ocorrer com herança múltipla não ocorrem com herança simples e interfaces.

Uma interface não é um substituto para herança múltipla, porque na herança múltipla existe reúso de código, enquanto as interfaces não fornecem essa opção. Existe uma diferença importante, já que o reúso de código é um dos benefícios primários da herança.

Como um exemplo de uma interface, considere o método `sort` da classe padrão de Java, `Arrays`. Qualquer classe que usa esse método deve fornecer uma implementação de um método para comparar os elementos a serem ordenados. A interface `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 é menor do que o objeto passado como parâmetro, zero se forem iguais e um inteiro positivo se o parâmetro for menor do 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.

O Capítulo 14 ilustra o uso de interfaces na manipulação de eventos em Java.

12.6.3 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 sobreescrito e todas as vinculações são estáticas. A vinculação estática é usada também se o método for estático (`static`) ou privado (`private`), no qual ambos os modificadores não permitem sobreescrita.

12.6.4 Classes aninhadas

Java tem diversas variedades de classes aninhadas, todas com a vantagem de serem ocultas de todas as classes em seus pacotes, exceto para a aninhadora. Classes não estáticas aninhadas diretamente em outra, chamadas de **classes internas** (*inner classes*), têm um ponteiro implícito para a aninhadora que dá aos métodos da aninhada acesso a todos os membros da primeira, 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++.

Uma instância de uma classe aninhada pode existir apenas dentro de uma instância de sua aninhadora. Classes aninhadas podem ser também anônimas, contendo uma sintaxe complexa, mas são uma forma abreviada de definir uma classe usada de apenas um lugar. Um exemplo de uma classe 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.

12.6.5 Avaliação

O projeto de Java para suporte à programação orientada a objetos é similar ao de C++, mas ela emprega uma aderência mais consistente aos princípios de orientação a objetos. Devido à sua falta de funções, Java não suporta programação procedural. Além disso, não permite classes sem pais e usa vinculação dinâmica como a maneira “normal” de vincular chamadas a métodos às definições de métodos. 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 fornecer uma forma simples de suporte para herança múltipla, enquanto C++ inclui suporte completo, mas complexo, para tal.

12.7 SUPORTE PARA PROGRAMAÇÃO ORIENTADA A OBJETOS EM C#

O suporte de C# para programação orientada a objetos é similar ao de Java.

12.7.1 Características gerais

Conforme discutido no Capítulo 11, C# inclui tanto classes quanto estruturas, com classes bastante similares às de Java e estruturas como construções dinâmicas da pilha menos poderosas.

12.7.2 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 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 prefixando a chamada com **base**. Por exemplo,

```
base.Draw();
```

O suporte de C# para interfaces é o mesmo de Java.

12.7.3 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 como **virtual**, como em C++. Para evitar sobreescrita acidental, os métodos correspondentes nas classes derivadas devem ser marcados com **override**. A inclusão de **override** torna claro que o método é uma nova versão de um método herdado. Por exemplo, a versão C# da classe C++ **shape** que aparece na Seção 12.5.3 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 que são especificados com sintaxe diferente. Por exemplo, o seguinte método é abstrato em C#:

```
abstract public void Draw();
```

Uma classe que inclui ao menos um método abstrato é abstrata, e toda classe abstrata deve ser marcada como **abstract**, não podendo ser instanciada, ou seja, quaisquer subclasses de uma abstrata que serão instanciadas devem implementar todos os métodos abstratos que ela 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#.

12.7.4 Classes aninhadas

Uma classe C# diretamente aninhada em uma classe aninhadora se comporta como uma classe aninhada estática em Java (como uma classe aninhada em C++). C# não suporta classes aninhadas que se comportam como as aninhadas não estáticas de Java.

12.7.5 Avaliação

Como C# é a linguagem orientada a objetos baseada em C projetada mais recentemente, deve-se esperar que seus projetistas tenham aprendido com seus antecessores, duplicado os sucessos do passado e remediado alguns dos problemas. Um resultado disso, com os poucos problemas com Java, é que as diferenças entre o suporte de C# para programação orientada a objetos e o de Java são relativamente pequenas.

12.8 SUPORTE PARA PROGRAMAÇÃO ORIENTADA A OBJETOS EM ADA 95

Ada 95 foi derivada de Ada 83, com algumas extensões significativas. Esta seção apresenta uma breve visão sobre as extensões projetadas para suportar a programação orientada a objetos. Como Ada 83 já incluía construções para criar tipos de dados abstratos, discutidos no Capítulo 11, os recursos adicionais necessários para Ada 95 foram aqueles para suportar herança e vinculação dinâmica. Os objetivos de projeto de Ada 95 eram requerer mudanças mínimas aos tipos e às estruturas de pacote de Ada 83 e reter o máximo de verificação de tipos possível.

12.8.1 Características gerais

As classes de Ada 95 formam uma nova categoria de tipos chamados de **tipos etiquetados** (*tagged types*), que podem ser ou registros ou tipos privados. Os tipos etiquetados são definidos em pacotes, que permitem que sejam compilados separadamente. Tipos etiquetados são chamados dessa forma porque cada objeto implicitamente inclui uma etiqueta mantida pelo sistema que indica seu tipo. Os subprogramas que definem as operações em um tipo etiquetado aparecem na mesma lista de declaração que a declaração de tipo. Considere o seguinte exemplo:

```
package Person_Pkg is
    type Person is tagged private;
    procedure Display(P : in Person);
    private
        type Person is tagged
            record
                Name : String(1..30);
                AddressS : String(1..30);
                Age : Integer;
            end record;
    end Person_Pkg;
```

Esse pacote define o tipo `Person`, que é útil por si só e pode também servir como classe pai de classes derivadas.

Diferentemente de C++, não existe chamada implícita de subprogramas construtores ou destrutores em Ada 95. Esses subprogramas podem ser escritos, mas devem ser explicitamente chamados pelo programador.

12.8.2 Herança

Ada 83 suporta apenas uma forma restrita de herança com seus tipos derivados e subtipos. Em ambos, um novo tipo pode ser definido com base em um existente. A única modificação permitida é restringir a faixa de valores do novo tipo. Esse não é o tipo de herança completa requerida para programação orientada a objetos, suportada por Ada 95.

Tipos derivados em Ada 95 são baseados em tipos etiquetados. Novas entidades são adicionadas às entidades herdadas colocando-as em uma definição de registro. Considere o exemplo:

```
with Person_Pkg; use Person_Pkg;
package Student_Pkg is
    type Student is new Person with
        record
            Grade_Point_Average : Float;
            Grade_Level : Integer;
        end record;
```

```
procedure Display(St : in Student);
end Student_Pkg;
```

Nele, o tipo derivado `Student` é definido para ter as entidades de sua classe pai, `Person`, com as novas entidades `Grade_Point_Average` e `Grade_Level`. Ele também redefine o procedimento `Display`. Essa nova classe é definida em um pacote separado para permitir que seja modificada sem requerer recompilação do pacote contendo a definição do tipo pai.

Esse mecanismo de herança não permite que alguém previna as entidades da classe pai de serem incluídas na classe derivada. Consequentemente, as classes derivadas podem apenas estender as classes pai – e, dessa forma, são subtipos. Entretanto, pacotes de biblioteca filhos, discutidos brevemente a seguir, podem ser usados para definir subclasses que não são subtipos.

Suponha que tivéssemos as definições:

```
P1 : Person;
S1 : Student;
Fred : Person := ("Fred", "321 Mulberry Lane", 35);
Freddie : Student :=
    ("Freddie", "725 Main St.", 20, 3.25, 3);
```

Como `Student` é um subtipo de `Person`, a atribuição

```
P1 := Freddie;
```

seria permitida, e é. As entidades `Grade_Point_Average` e `Grade_Level` de `Freddie` são simplesmente ignoradas na coerção necessária.

A questão óbvia agora é se uma atribuição na direção oposta é permitida; ou seja, podemos atribuir uma pessoa (`Person`) a um estudante (`Student`)? Em Ada 95, essa ação é permitida em uma forma que inclui as entidades na subclasse. Em nosso exemplo, o seguinte código é permitido:

```
S1 := (Fred, 3.05, 2);
```

Para derivar uma classe que não inclua todas as entidades da classe pai, os pacotes de biblioteca filhos são usados. Esses pacotes têm seu nome pré-fixado com o do pacote pai e podem também ser usados no lugar de definições amigas em C++. Por exemplo, se um subprograma deve ser escrito de forma que possa acessar os membros de duas classes diferentes, o pacote pai pode definir uma das classes e o pacote filho pode definir a outra. Então, um subprograma no pacote filho pode acessar os membros de ambos.

Ada 95 não fornece herança múltipla. Apesar de classes genéricas e de herança múltipla serem conceitos relacionados de forma distante, existe uma maneira de atingir um efeito similar à herança múltipla usando tipos genéricos. Ela não é, entretanto, tão elegante quanto a abordagem de C++ e não é discutida aqui.

12.8.3 Vinculação dinâmica

Ada 95 fornece vinculação tanto estática quanto dinâmica de chamadas a procedimentos para definições de procedimentos em tipos etiquetados. A vinculação dinâmica é迫使ada por meio do uso de um tipo polimórfico, que representa todos os tipos em uma hierarquia de classes cuja raiz é de um tipo em particular. Cada tipo etiquetado tem um tipo polimórfico. Para um tipo etiquetado T, o polimórfico é especificado com T' **class**. Se T é um tipo etiquetado, uma variável de T' **class** pode armazenar um objeto do tipo T ou de qualquer tipo derivado de T.

Considere novamente as classes Person e Student definidas na Seção 12.8.2. Suponha que tivéssemos uma variável do tipo Person' **class**, Pcw, que em algumas vezes referencia um objeto Person e em outras referencia um objeto Student. A seguir, suponha que quiséssemos mostrar o objeto referenciado por Pcw, independentemente de estar sendo referenciado um Person ou um Student. Esse resultado requer que a chamada a Display seja dinamicamente vinculada à versão correta de Display. Poderíamos usar um novo procedimento que recebesse um parâmetro do tipo Person e o enviasse para Display. A seguir, está tal procedimento:

```
procedure Display_Any_Person(P: in Person) is
begin
    Display(P);
end Display_Any_Person;
```

Esse procedimento pode ser chamado com as chamadas:

```
with Person_Pkg; use Person_Pkg;
with Student_Pkg; use Student_Pkg;
P : Person;
S : Student;
Pcw : Person' class;
...
Pcw := P;
Display_Any_Person(Pcw); -- chama Display de Person
Pcw := S;
Display_Any_Person(Pcw); -- chama Display de Student
```

Ada 95 também suporta ponteiros polimórficos. Eles são definidos como tendo o tipo polimórfico, como em

```
type Any_Person_Ptr is access Person' class;
```

Tipos base puramente abstratos podem ser definidos em Ada 95 com a inclusão da palavra reservada **abstract** nas definições de tipos e nas definições de subprogramas. Além disso, as definições de subprogramas não podem ter corpos. Considere esse exemplo:

```
package Base_Pkg is
type T is abstract tagged null record;
```

```
procedure Do_It (A : T) is abstract;
end Base_Pkg;
```

12.8.4 Pacotes filhos

Pacotes podem ser aninhados diretamente em outros, nesse caso chamados de **pacotes filhos**. O problema com esse projeto é que se um pacote tem um número significativo de pacotes filho e eles são grandes, o aninhador se torna muito grande para ser uma unidade de compilação eficaz. A solução é relativamente simples: é permitido que os pacotes filhos sejam unidades compiláveis separadamente. O nome de um pacote filho é o nome do aninhador com o nome individual do filho anexado com um ponto. Por exemplo, se o nome do pacote aninhador é `Binary_Tree` e o nome individual do filho é `Traversals`, o nome completo do filho é `Binary_Tree.Traversals`.

Pacotes filhos podem ser públicos (padrão) ou privados. A posição lógica de um filho público é no final das declarações no pacote de especificação do aninhador. Logo, todas as entidades declaradas no pacote de especificação do aninhador são visíveis para o filho. Entretanto, quaisquer declarações que apareçam no corpo do pacote aninhador são ocultadas do filho. Se existir mais de um pacote filho, eles não estão logicamente em ordem sequencial. Logo, as declarações de um não são visíveis para outro pacote filho a menos que ele inclua uma cláusula `with` com o nome do primeiro.

Um pacote filho é declarado como privado ao precedermos a palavra reservada `package` com a palavra reservada `private`. A posição lógica de um pacote filho privado é no início das declarações do pacote de especificação do aninhador. As declarações do pacote filho privado não são visíveis para o corpo do pacote aninhador, a menos que este inclua uma cláusula `with` com o nome do filho.

12.8.5 Avaliação

Ada oferece suporte completo para programação orientada a objetos, apesar de os usuários de outras linguagens orientadas a objetos poderem achar esse suporte fraco. Apesar de os pacotes poderem ser usados para construir tipos de dados abstratos, eles são construções de encapsulamento mais generalizadas. A menos que pacotes de biblioteca filhos sejam usados, não existe uma forma de restringir a herança, logo, nesse caso, todas as subclasses são subtipos. Essa forma de restrição de acesso é limitada em comparação com a oferecida por C++, Java e C#.

C++ oferece uma forma melhor de herança múltipla do que Ada 95. Entretanto, o uso de unidades de biblioteca filhas para controlar o acesso das entidades da classe pai parece ser uma solução mais limpa do que as funções e classes amigas de C++. Por exemplo, se a necessidade de um amigo não é conhecida quando uma classe é definida, ela precisará ser modificada e recompilada quando tal necessidade for descoberta. Em Ada 95, novas classes em novos pacotes filhos são definidas sem perturbar o pai, porque cada nome definido no pai é visível no filho.

A inclusão de construtores e destrutores em C++ para a inicialização de objetos é boa, mas Ada 95 não inclui tais capacidades.

Outra diferença entre essas duas linguagens é que o projetista de uma classe raiz em C++ deve decidir se uma função membro em particular será vinculada estaticamente ou dinamicamente. Se a escolha é feita em favor da vinculação estática, mas uma mudança posterior no sistema requerer vinculação dinâmica, a classe raiz deve ser modificada. Em Ada 95, essa decisão de projeto não precisa ser feita no da classe raiz. Cada chamada pode por si só especificar se será vinculada estaticamente ou dinamicamente, independentemente do projeto da classe raiz.

Uma diferença mais sutil é que a vinculação dinâmica nas linguagens orientadas a objetos baseadas em C é restrita aos ponteiros e/ou às referências a objetos, em vez de ser para os objetos propriamente ditos. Ada 95 não tem tal restrição, então é mais ortogonal.

12.9 SUPORTE PARA PROGRAMAÇÃO ORIENTADA A OBJETOS EM 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 a mesma aparência que as expressões em linguagens como Java, essas expressões são na verdade avaliadas por meio de passagem de mensagens. Quando alguém escreve `a + b`, a expressão é executada como o envio da mensagem `+` para o objeto referenciado por `a`, passando uma referência ao objeto `b` como parâmetro.

12.9.1 Características gerais

Lembre-se, do Capítulo 11, de que 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 incluem 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 as duas definições nas cláusulas `senão` e `então` de uma construção de seleção.

Todas as variáveis em Ruby são referencias 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. Se o acesso externo a uma variável de instância é requerido, 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 ambos. Se alguém quer 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 a função, usando `:one` e `:two` como os parâmetros reais. Preceder uma variável com dois pontos (`:`) faz com que o nome da variável seja usado, em vez de desreferenciá-lo para o objeto a qual ela se refere.

A função que cria métodos de escrita de maneira similar é chamada de `attr_writer`. Essa função tem o mesmo perfil de parâmetros de `attr_reader`.

As funções para criar métodos de leitura e de escrita são chamadas assim porque fornecem o protocolo para objetos da classe, que em Ruby são cha-

madas de **atributos**. Então, os atributos de uma classe formam a interface de dados (os dados públicos) para os objetos da classe.

Lembre-se, do Capítulo 11, de que 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 por meio de uma chamada a `super` com os valores iniciais como parâmetros reais. `super` chama o método na classe pai com o mesmo nome do método no qual a chamada a `super` aparece.

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 ele 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`, `protected` e `public`. Uma maneira é chamar a função apropriada sem parâmetros. Isso reinicia o acesso padrão para métodos definidos na sequência na classe. Por exemplo,

```
class MyClass
  def meth1
    ...
  end
  ...
  private
  def meth7
    ...
  end
  ...
  protected
  def meth11
    ...
  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 equivalente semanticamente à definição de classe anterior:

```
class MyClass
  def meth1
    ...
  end
  ...
  def meth7
    ...
  end
  ...
  def meth11
```

```

...
end
...
private :meth7, ...
protected :meth11, ...
end # da classe MyClass

```

Variáveis de classe, especificadas precedendo-se seus nomes com 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 usadas.

12.9.2 Herança

As subclasses são definidas em Ruby com o símbolo menor que (<), em vez de com os dois pontos de C++. Por exemplo,

```
class MySubClass < BaseClass
```

Algo distinto acerca dos controles de acesso a métodos de Ruby é que eles podem ser modificados em uma subclasse, simplesmente chamando as 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 privatamente acessível na subclasse. Tal subclasse obviamente não pode ser um subtipo.

Lembre-se de que os módulos Ruby foram discutidos no Capítulo 11. Eles fornecem um encapsulamento de nomeação em geral usado para definir bibliotecas de funções. Talvez o aspecto mais interessante dos módulos, entretanto, é que suas funções podem ser acessadas diretamente a partir das classes. O acesso ao módulo em uma classe é especificado com uma sentença `include`, como

```
include Math
```

O efeito de incluir um módulo é que a classe ganha um ponteiro para o módulo e efetivamente herda as funções definidas nele. Na verdade, quando um módulo é incluído em uma classe, ele se torna uma superclasse *proxy* da classe. Tal módulo é chamado de um **mixin**, porque suas funções se misturam com os métodos definidos na classe. Os *mixins* fornecem uma maneira de incluir a funcionalidade de um módulo em qualquer módulo em qualquer classe que precisar. E, é claro, a classe ainda tem uma superclasse normal a partir da qual herda membros. Então, os *mixins* fornecem os benefícios da herança múltipla, sem as colisões de nomes que podem ocorrer se os módulos não requerem nomes de módulos em suas funções.

12.9.3 Vinculação dinâmica

O suporte para vinculação dinâmica em Ruby é o mesmo que o 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 propriamente ditos são dinâmicas.

12.9.4 Avaliação

Como Ruby é uma linguagem de programação orientada a objetos no sentido mais puro, seu suporte a programação orientada a objetos é adequado. Entretanto, o controle de acesso para os membros de classe é mais fraco do que o de C++. Além disso, a herança múltipla não é suportada. Por fim, Ruby não suporta classes abstratas ou interfaces, apesar de os *mixins* serem fortemente relacionados às interfaces.

12.10 IMPLEMENTAÇÃO DE CONSTRUÇÕES ORIENTADAS A OBJETOS

Existem ao menos duas partes 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, damos uma breve olhada nessas duas.

12.10.1 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. Chamamos a forma dessa estrutura de um **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 feita como nos registros, usando deslocamentos constantes a partir do início da instância do RIC. Isso torna esses acessos tão eficientes como os dos campos dos registros.

12.10.2 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 poderiam simplesmente ter um ponteiro para o código do método, configurado no momento

de 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 da instância.

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, por isso, a lista de tais métodos deve 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 uma **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 12.2. Observe que o ponteiro para o método *area* na *vtable* de B aponta para o código para o método *area* de A. A razão é que B não sobrescreve o método *area* de A, então se um cliente de B chama *area*, é 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 sobreescrito em B e *sift* é definido como uma adição em B.

Suponha que uma classe chamada C crie objetos de A e B, com referências chamadas *aObj* e *bObj*, respectivamente. Suponha também que existe uma sentença de atribuição em C

```
aObj = bObj;
```

Agora, se existe uma chamada

```
aObj.draw();
```

e a sentença de atribuição apenas copia o ponteiro de *bObj* para *aObj*, o método *draw* de A seria chamado, o que seria incorreto. Esse exemplo ilustra que a

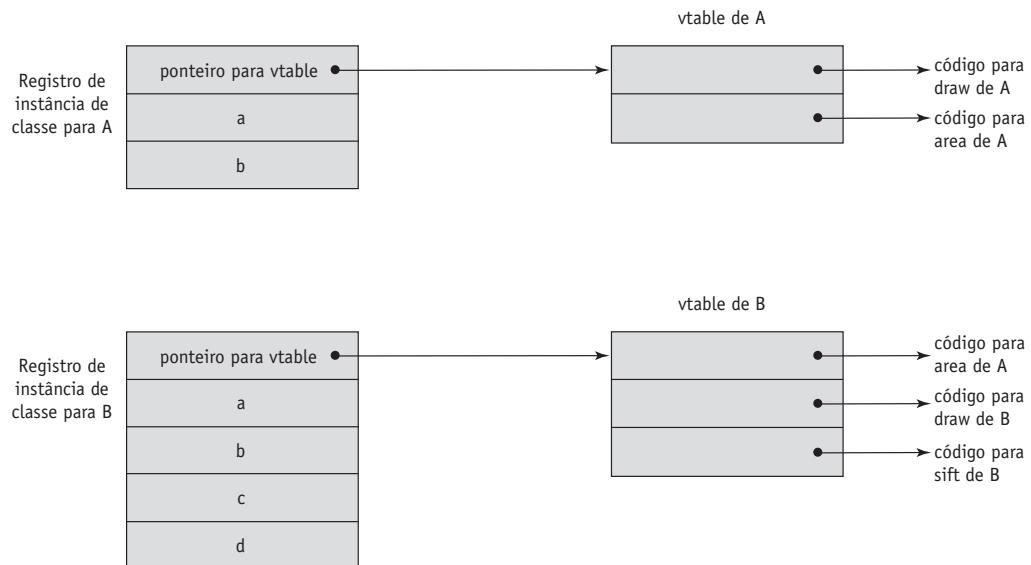


Figura 12.2 Um exemplo das RICs com herança simples.

atribuição da referência *bObj* para *aObj* deve ter o efeito colateral de trocar o ponteiro da *vtable* de *aObj* para apontar para a *vtable* da classe *B*.

A herança múltipla complica a implementação da vinculação dinâmica. Considere três definições de classe em C++:

```
class A {
    public:
        int a;
        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 tem 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`. Segundo isso, no RIC de `C` está a parte da visão de `B`, a qual inicia com um ponteiro na *vtable* para os métodos virtuais de `B`, seguido pelos dados herdados de `B` e os dados definidos em `C`. O RIC para `C` é mostrado na Figura 12.3.

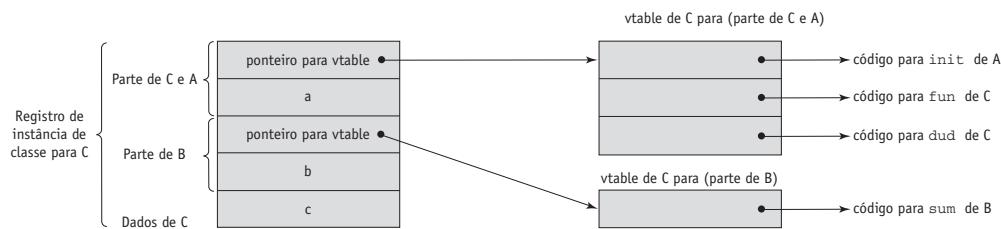


Figura 12.3 Um exemplo de um RIC de uma subclasse com múltiplos pais.

RESUMO

A programação orientada a objetos envolve 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 das 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 um objeto e todas as computações são realizadas por passagem de mensagens. Em Smalltalk, todas as subclasses são subtipos. Toda a verificação de tipos e a toda a vinculação de mensagens a métodos é dinâmica 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. Embora a vinculação dinâmica de Smalltalk forneça um pouco mais de flexibilidade de programação do que a linguagem

híbrida C++, é muito menos eficiente. C++ fornece herança múltipla e liberação explícita de objetos. C++ inclui uma variedade de controles de acesso para as entidades em classes, alguns dos quais previnem que as subclasses sejam subtipos. Tanto métodos construtores quanto destrutores podem ser incluídos nas classes; ambos são implicitamente chamados.

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 a herança simples é diretamente suportada, apesar de um tipo de herança múltipla ser possível usando interfaces. Toda a vinculação de mensagens a métodos é dinâmica, exceto no caso de métodos que não podem ser sobreescritos. Além das classes, Java inclui pacotes como uma segunda construção de encapsulamento.

Ada 95 fornece suporte para programação orientada a objetos por meio de tipos etiquetados, que podem suportar herança. A vinculação dinâmica é possível usando tipos ponteiros polimórficos. Tipos derivados são extensões aos tipos pais, a menos que sejam definidos em pacotes de biblioteca filhos, que no caso das entidades do tipo pai podem ser eliminadas no tipo derivado. Fora os pacotes de biblioteca filhos, todas as subclasses são subtipos.

C#, baseado 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 sobreescritos devem ser marcados usando **virtual**, e os métodos que sobreescrivem devem ser marcados usando **override**. Todas as classes (e todos os tipos primitivos) são derivados 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 que são executáveis e podem ser modificadas em qualquer momento. Ruby suporta apenas herança simples, e subclasses não são necessariamente subtipos.

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 sobrecededor?
6. Descreva uma situação na qual a vinculação dinâmica tem uma grande vantagem sobre sua ausência.

7. O que é um método virtual?
8. O que é um método abstrato? O que é uma classe abstrata?
9. Descreva brevemente as oito 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 que Smalltalk tem tido na computação?
17. Em essência, 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?
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**) em C++?
25. O que é uma função virtual pura em C++?
26. Como os parâmetros são enviados para o construtor de uma superclasse?
27. Qual é a diferença prática mais importante entre Smalltalk e C++?
28. Como o sistema de tipos de Java é diferente do de C++?
29. De onde os objetos em Java podem ser alocados?
30. O que é encaixotamento (*boxing*)?
31. Como os objetos em Java são liberados?
32. Todas as subclasses em Java são subtipos?
33. Como os construtores da superclasse são chamados em Java?
34. Sob quais circunstâncias uma chamada a método em Java é estaticamente vinculada a um método?
35. De que maneira métodos sobrecarregados em C# diferem sintaticamente de seus correspondentes em C++?
36. Como a versão pai de um método herdado sobreescrito em uma subclasse pode ser chamada em tal subclasse em C#?
37. Todas as subclasses em Ada 95 são subtipos?
38. Como uma chamada a um subprograma em Ada 95 é especificada como dinamicamente vinculada a uma definição de subprograma? Quando essa decisão é tomada?
39. Como Ruby implementa tipos primitivos, como aqueles para dados inteiros ou de ponto flutuante?
40. Como os métodos de leitura são definidos em uma classe Ruby?

41. Que controles de acesso Ruby suporta para variáveis de instância?
42. Que controles de acesso Ruby suporta para métodos?
43. Todas as subclasses em Ruby são subtipos?
44. Ruby suporta herança múltipla?

CONJUNTO DE PROBLEMAS

1. Que parte importante do suporte para programação orientada a objetos faltava em SIMULA 67?
2. De que maneiras o relacionamento entre um método sobreescrito com o método que o sobrecreve pode ser definido como “compatível”?
3. Compare a vinculação dinâmica de C++ com a de Java.
4. Compare os controles de acesso de entidades de classe de C++ e Java.
5. Compare os controles de acesso de entidades de classe de C++ e Ada 95.
6. Compare a herança múltipla de C++ com a fornecida por interfaces em Java.
7. Compare as capacidades de tipos genéricos de Java 5.0 com as de C++.
8. Compare as capacidades de tipos genéricos de C# 2005 com as de C++.
9. Cite uma situação de programação na qual a herança múltipla tenha uma vantagem significativa sobre as interfaces.
10. Explique os dois problemas com os tipos de dados abstratos aliviados por herança.
11. Descreva as categorias de mudanças que uma subclasse pode realizar em sua classe pai.
12. Explique uma desvantagem da herança.
13. Explique as vantagens e as desvantagens de ter todos os valores em uma linguagem como objetos.
14. O que exatamente significa para uma subclasse ter um relacionamento é-um(a) com sua classe pai?
15. Descreva a questão de o quanto fortemente os parâmetros de um método sobrecedor devem casar com os parâmetros do método que ele sobrecreve.
16. Explique a verificação de tipos de Smalltalk.
17. Os projetistas de Java obviamente pensaram que não valia a pena o ganho adicional de eficiência ao ser permitido que qualquer método fosse estaticamente vinculado, como no caso de C++. Quais são os argumentos a favor e contra o projeto de Java?
18. Qual é a razão primária pela qual todos os objetos Java têm um ancestral comum?
19. Qual é o propósito da cláusula **finalize** em Java?
20. O que seria ganho se Java permitisse objetos dinâmicos da pilha, assim como objetos dinâmicos do monte? Qual seria a desvantagem de ter ambos?
21. Compare a maneira pela qual Ada 95 fornece polimorfismo com aquela de C++, em termos de conveniência de programação.
22. Quais são as diferenças entre uma classe abstrata C++ e uma interface Java?
23. Explique por que permitir que uma classe implemente múltiplas interfaces em Java e em C# não cria os mesmos problemas que a herança múltipla em C++ cria.
24. Estude e explique as questões de 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.

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 Ada 95.
3. Repita o Exercício de Programação 1 usando Ruby.
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 instancia A, B e C e que inclua três chamadas ao método. Uma das chamadas deve ser estaticamente vinculada ao método de A, outra deve ser dinamicamente vinculada ao método de B e a terceira deve ser dinamicamente vinculada ao método de C. Todas as chamadas a métodos devem ser por meio de um ponteiro para a classe A.
5. Escreva um programa em C++ que chame tanto um método dinamicamente vinculado quanto um método estaticamente vinculado um grande número de vezes, cronometrando o tempo das chamadas para os dois métodos. Compare os tempos e calcule a diferença necessária para ambos. Explique os resultados.
6. Repita o Exercício 5 usando Java, forçando a vinculação estática com `final`.

Capítulo 13

Concorrência

13.1 Introdução

13.2 Introdução à concorrência no 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 no nível de sentença

Este capítulo começa com introduções aos vários tipos de concorrência no nível de subprogramas ou de unidades, e também no nível de sentenças. É incluída uma breve descrição dos tipos comuns de arquiteturas multiprocessadas de computadores. A seguir, é apresentada uma extensa discussão sobre concorrência no 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 no nível de unidade, especificamente a sincronização de competição e a de cooperação. A seguir, são descritas as questões de projeto para fornecer suporte linguístico para concorrência. Posteriormente, segue uma discussão detalhada, incluindo exemplos de código, das três principais abordagens para o suporte linguístico para concorrência: semáforos, monitores e passagem de mensagens. Um programa exemplo em pseudocódigo é usado para demonstrar como os semáforos podem ser usados. Ada e Java são usadas para ilustrar monitores; para passagem de mensagens, Ada é usada. 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) e a passagem de mensagens assíncronas também são discutidos. O suporte para concorrência no nível de unidade em Java e C# é então discutido. A última seção do capítulo é uma breve discussão da concorrência no 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: no nível de instrução (executando duas ou mais instruções de máquina simultaneamente), no nível de sentença (executando duas ou mais sentenças na linguagem fonte simultaneamente), no nível de unidade (executando duas ou mais unidades de subprograma simultaneamente) e no nível de programa (executando dois ou mais programas simultaneamente). Como nenhuma questão de projeto de linguagem está envolvida com elas, a concorrência no nível de instrução e no nível de programa não são discutidas neste capítulo. A concorrência tanto nos níveis de subprograma e de sentença é discutida, com a maior parte do foco no nível de subprograma.

A execução concorrente de subprogramas pode ocorrer ou fisicamente, em processadores separados, ou logicamente, compartilhando-se um único processador. À primeira vista, a concorrência pode parecer um conceito simples, mas ela apresenta um desafio significativo para o projetista de linguagens de programação.

Mecanismos de controle de concorrência aumentam a flexibilidade da programação. Eles foram originalmente inventados para serem usados em problemas particulares oriundos dos sistemas operacionais, mas eles são requeridos por uma variedade de outras aplicações de programação. Alguns dos programas mais usados atualmente são os navegadores Web, cujo projeto é fortemente baseado em concorrência. Os navegadores devem realizar muitas funções diferentes ao mesmo tempo, dentre 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. Outro exemplo são os sistemas de software

projetados para simular sistemas físicos reais que consistem em múltiplos subsistemas concorrentes. Para esses tipos de aplicações, a linguagem de programação deve suportar concorrência no nível de unidade.

A concorrência no nível de sentença é de certa forma diferente. Do ponto de vista de um projetista de linguagem, a concorrência no nível de sentença é uma questão de especificar como os dados devem ser distribuídos sobre múltiplas memórias e quais sentenças podem ser executadas concurrentemente. 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 aumenta com cada nova geração 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 em particular – ou seja, devem rodar eficientemente em máquinas com diferentes arquiteturas.

A intenção deste capítulo é discutir os aspectos de concorrência mais relevantes para questões de projeto de linguagem, em vez de apresentar um estudo definitivo de todas as questões de concorrência, que seria inadequado para um livro sobre linguagens de programação.

13.1.1 Arquiteturas multiprocessadas

Diversas arquiteturas de computadores tem mais de um processador e podem suportar alguma forma de execução concorrente. Antes de iniciarmos a discussão acerca da 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 a esses computadores, surgidos no final dos anos 1950, executar um programa enquanto realizavam entrada ou saída para outros programas.

No início dos anos 1960, existiam máquinas com processadores múltiplos completos, usados pelo escalonador de processos do sistema operacional, que distribuía processos separados de uma final de processos em lote para os processadores separados. Sistemas com essa estrutura suportavam a concorrência no nível de programa.

Em meados dos anos 1960, apareceram máquinas com diversos processadores parciais idênticos alimentados com certas instruções a partir de um único fluxo de instruções. 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 concurrentemente e precisavam agendar essas instruções de forma correta. Sistemas com essa estrutura suportavam concorrência no 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 múltiplos fluxos de dados eram definidas como segue: computadores com múltiplos processadores que executam a mesma instrução simultaneamente, cada um com dados diferentes, são chamados de arquiteturas de computadores SIMD – Single-Instruction Multiple Data (Única Instrução Múltiplos Dados). Em um computador SIMD, cada processador tem sua própria memória local. Um processador controla a operação dos outros 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. Os tipos de programas que podem se beneficiar mais dessa arquitetura são comuns na computação científica, uma área geralmente alvo de máquinas multiprocessadas. Até pouco tempo, a maioria dos 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 aparecer em 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 chassi quanto distribuídas, talvez em uma grande área. As máquinas MIMD de memória compartilhada obviamente devem fornecer alguma forma de sincronização para prevenir colisões de acesso à memória. Mesmo máquinas MIMD distribuídas requerem sincronização para operarem juntas em programas únicos. Computadores MIMD, mais caros e mais gerais do que computadores SIMD, suportam concorrência no nível de unidade.

Com o advento de chips únicos de computadores poderosos, mas de baixo custo, tornou-se possível ter um grande número desses microprocessadores conectados em pequenas redes dentro de um chassi. Esses tipos de computadores, que usam microprocessadores de prateleira, apareceram de diversos fabricantes.

Uma razão importante pela qual os aplicativos de software não evoluíram para usar as máquinas concorrentes é o poder dos processadores ter aumentado continuamente. Uma das motivações mais fortes para usar máquinas concorrentes é aumentar a velocidade de computação. Entretanto, dois fatores de hardware foram combinados para fornecer computação mais rápida, sem requerer qualquer mudança na arquitetura dos sistemas de software. Primeiro, as taxas de *clock* dos processadores têm se tornado cada vez mais rápida com cada geração de processadores (mais ou menos a cada 18 meses).

Segundo, diversos tipos de concorrência vêm pré-definidos nas arquiteturas dos processadores. Dentre eles, estão o *pipelining* de instruções e dados da memória para o processador (as instruções são obtidas e decodificadas enquanto a instrução atual está sendo 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 sem requerer 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. Com a aparição de processadores múltiplos em um único *chip*, como os Intel Core Duo, há mais pressão para desenvolvedores de software usarem mais os processadores múltiplos disponíveis nas máquinas. Se eles não fizerem, 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 uma na qual, assumindo a disponibilidade de mais de um processador, diversas unidades do mesmo programa são executadas simultaneamente. Essa é a **concorrência física**. Um leve relaxamento desse conceito de concorrência permite ao programador e ao aplicativo de software assumirem a existência de múltiplos processadores fornecendo concorrência real, quando a execução real dos programas está ocorrendo de maneira intercalada em um processador. Essa é a **concorrência lógica**. É similar à ilusão de execução simultânea fornecida para diferentes usuários de um sistema de computação de multiprogramação de tempo compartilhado. Do ponto de vista do programador e do projetista de linguagem, a concorrência lógica é o mesmo que a concorrência física. É tarefa do implementador da linguagem, usando as capacidades do sistema operacional, mapear a concorrência lógica para o 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 uma execução em particular 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 altamente complexo

impossível de seguir visualmente. Formalmente, uma **linha de execução de controle** em um programa é a sequência de pontos alcançada à medida que o controle flui pelo programa.

Programas com corrotinas (veja o Capítulo 9), entretanto, 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. Um programa projetado para ter mais de uma linha de execução de controle é dito como **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. Ele se torna, nesse caso, um programa praticamente *multithreaded*.

A concorrência no nível de sentença é um conceito relativamente simples. Laços com sentenças inclusas que operam em elementos de matrizes são separados de forma que possa distribuir o processamento sobre múltiplos processadores. Por exemplo, um laço que executa 500 repetições e inclui uma sentença que opera em um dos 500 elementos de um vetor pode ser separado de forma que cada um dos 10 processadores possam simultaneamente processar 50 elementos do vetor.

13.1.3 Motivações para o uso de concorrência

Existem ao menos duas razões para projetar sistemas de software concorrentes. A primeira é a velocidade de execução dos programas. Hardware concorrente fornece uma maneira efetiva de aumentar a velocidade de execução de programa, desde que os programas sejam projetados para usar a concorrência em hardware. Existe agora um grande número de computadores instalados com múltiplos processadores, incluindo muitos dos computadores pessoais vendidos nos últimos anos. É uma perda não usar essa capacidade de hardware.

A segunda 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 a solução de 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 uma área de controle, estações de relés em uma rede de comunicações e as várias máquinas em uma indústria de manufatura. Os aplicativos de software que usam concorrência devem ser utilizados para simular tais sistemas de forma precisa.

13.2 INTRODUÇÃO À CONCORRÊNCIA NO NÍVEL DE SUBPROGRAMA

Antes de o suporte de linguagem para concorrência poder ser considerado, deve-se entender os conceitos subjacentes de concorrência e os requisitos para que tal suporte seja útil. Esses tópicos são cobertos 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, às vezes chamada de **processo**, pode suportar uma linha de execução de controle.

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, ela não precisa esperar que a tarefa complete sua execução antes de continuar sua própria. Por fim, quando a execução de uma tarefa estiver completa, o controle pode ou não retornar à unidade que iniciou sua execução.

As tarefas caem em duas categorias gerais: pesadas (*heavyweight*) e leves (*light-weight*). Simplificando, cada **tarefa pesada** executa em seu próprio espaço de endereçamento. **Tarefas leves** rodam todas no mesmo espaço de endereçamento. É mais fácil de 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 variáveis não locais compartilhadas, passagem de mensagens ou parâmetros. Se uma tarefa não se comunica com ou não afeta a execução de qualquer outra no programa, é chamada **disjunta**. Como as tarefas geralmente trabalham juntas para criar simulações ou resolver problemas – e, dessa forma, são não disjuntas –, elas 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 complete alguma tarefa específica antes que A possa começar ou continuar sua execução. A **sincronização de competição** é necessária entre duas tarefas quando ambas requerem o uso de algum recurso que não pode ser simultaneamente usado. 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 complete seu processamento de x, independentemente de que processamento é esse. Então, para a sincronização de cooperação, as tarefas podem precisar esperar pelo término do processamento específico do qual sua operação correta depende, enquanto para a sincronização de competição, as tarefas podem precisar esperar pelo término de qualquer outro

processamento 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**, originado no desenvolvimento de sistemas operacionais, no qual uma unidade de programa produz algum valor de dado ou recurso e outra unidade o usa. Os dados produzidos são normalmente colocados em um *buffer* de armazenamento pela unidade produtora e removido 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 problema é chamado de **problema da sincronização de cooperação** porque os usuários da estrutura de dados compartilhada têm de cooperar para que *buffer* seja usado corretamente.

A sincronização de competição previne 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, o acesso mutuamente exclusivo aos dados compartilhados deve ser garantido.

Para clarear o problema da competição, considere o seguinte cenário: suponha que a tarefa A deva adicionar 1 à variável inteira compartilhada TOTAL, que tem um valor inicial de 3. A seguir, suponha que a tarefa B deva multiplicar o valor de TOTAL por 2. As tarefas A e B poderiam tentar modificar TOTAL ao mesmo tempo.

Cada tarefa realiza sua operação em TOTAL com o seguinte processo:

1. Obtém o valor de TOTAL.
2. Realiza a operação aritmética.
3. Coloca o novo valor de volta em TOTAL.

Sem sincronização de competição, quatro diferentes valores poderiam resultar dessas operações, dependendo de sua ordem. 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 e 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 é algumas vezes 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 chegar primeiro. A importância da sincronização de competição deve estar clara agora.

Um método geral para fornecer acesso mutuamente exclusivo (para suportar a sincronização de competição) a um recurso compartilhado é considerar o recurso como algo que uma tarefa pode possuir e permitir apenas que uma única tarefa possa possuí-lo em um dado momento. Para obter a posse

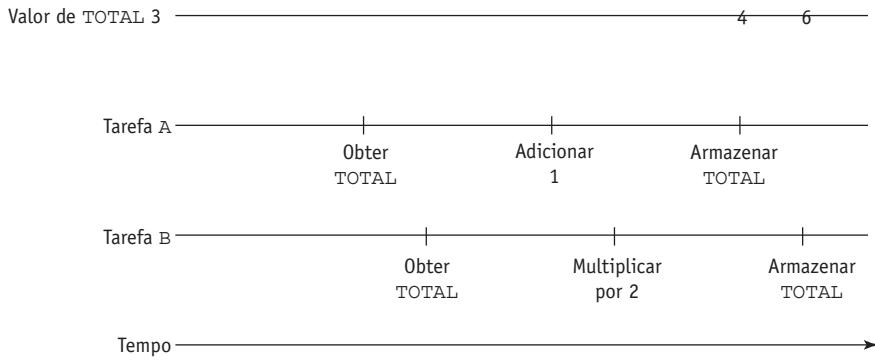


Figura 13.1 A necessidade por sincronização de competição.

de um recurso compartilhado, uma tarefa deve requisitá-lo. A posse será dada apenas se nenhuma outra tarefa estiver com ela. Quando uma tarefa estiver terminado o uso de um recurso seu, ela deve liberá-lo para ele ficar disponível a outras tarefas.

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 do que processadores. Um programa chamado de **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 em um processador por essa quantidade de tempo. É claro, existem diversos eventos complicadores, atrasos para sincronização e espera por operações de entrada e saída.

As tarefas podem estar em diversos estados:

1. *Nova*: Quando ela 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 previamente, mas foi bloqueada em uma das maneiras descritas no Parágrafo 4 desta subseção. As tarefas que estão 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 rodando, mas foi interrompida por um dentre diversos eventos – o mais comum é uma operação de entrada ou de saída. Como operações de entrada e de saída são muito mais lentas do que a execução de um programa, uma tarefa que inicia uma operação de entrada ou de saída é bloqueada de usar o processador enquanto espera que a operação esteja completa. Além desses três tipos de bloqueios, 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 completa ou se ela for morta explicitamente pelo programa.

Uma questão importante na execução de tarefas é: 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? Diversos algoritmos diferentes têm sido usados para essa escolha, alguns baseados em níveis especificáveis de prioridade. O algoritmo que faz a escolha é implementado no escalonador.

Associado com a execução concorrente de tarefas e com o uso de recursos compartilhados está o conceito de vivacidade (*liveness*). No ambiente de programas sequenciais, um programa tem a característica de vivacidade se ele continua a executar, eventualmente sendo completado. 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 não pode continuar e, dessa forma, nunca terminará.

Por exemplo, suponha que tanto a tarefa A quanto a tarefa B precisam dos recursos compartilhados X e Y para completar seu trabalho. A seguir, suponha que A ganha a posse de X e B ganha 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 um 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

NOTA HISTÓRICA

PL/I foi a primeira linguagem de programação a incluir tarefas concorrentes. Ela permitia que os programas de usuário executassem qualquer programa concorrentemente com a unidade que o chamou. O mecanismo para sincronização dessas execuções concorrentes era, entretanto, completamente inadequado. Ele consistia apenas de semáforos binários, chamados de eventos, e da habilidade de detectar quando uma tarefa havia completado sua execução. O ALGOL 68, que permitia concorrência composta no nível de sentenças, incluía um tipo de dados chamado *sema*.

a execução do programa nunca será completada normalmente. Esse tipo de perda de vivacidade é chamado de **impasse** (*deadlock*), uma ameaça séria para a confiabilidade de um programa – dessa forma, evitá-lo demanda sérias considerações tanto no projeto de linguagens quanto de programas.

Agora, estamos prontos para discutir alguns dos mecanismos linguísticos para fornecer controle de unidades concorrentes.

13.2.2 Projeto de linguagem para concorrência

Diversas linguagens têm sido projetadas para suportar a concorrência, começando com PL/I em meados dos anos 1960 e incluindo as linguagens contemporâneas Ada 95, Java, C#, Python e Ruby.

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 dessas, existem diversas questões de importância secundária. Proeminente dentre essas está como controlar 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.

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 respostas 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. 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 fornecer sincronização de cooperação.

Um **semáforo** é uma estrutura de dados que 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. O conceito de um semáforo é que, para fornecer acesso limitado a uma estrutura de dados, guardas são colocadas ao redor do código que acessa a estrutura. Uma **guarda** é um dispositivo lin-

guístico que permite ao código guardado ser executado apenas quando uma condição específica é verdadeira. Uma guarda pode ser usada para permitir que apenas uma tarefa acesse uma estrutura de dados compartilhada em um dado momento. Um semáforo é uma implementação de uma guarda. Uma parte integral de um mecanismo de guarda é um procedimento para garantir que todas as execuções tentadas do código de guarda ocorram em algum momento do tempo. O procedimento típico é fazer com que requisições para o acesso ocorridas quando ele não pode ser dado 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 nomeadas P e V por Dijkstra, em referência às duas palavras holandesas *passeren* (passar) e *vrygeren* (liberar) (Andrews e Schneider, 1983). Referiremo-nos a essas operações como *wait* e *release* no restante desta seção.

O processo pelo qual os semáforos fornecem guardas é descrito em termos das aplicações mais comuns na subseção seguinte.

13.3.2 Sincronização de cooperação

Em boa parte deste capítulo, usamos o exemplo de um *buffer* compartilhado para ilustrar diferentes abordagens 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 no *buffer* (para prevenir *transbordamentos (overflows)* e *transbordamentos negativos (underflows)* do *buffer*). O componente de contagem de uma variável de semáforo pode ser usado para esse propósito. Uma variável de semáforo – por exemplo, `emptyspots` – para armazenar o número de posições vazias em um *buffer* compartilhado, e outra – digamos, `fullspots` – para armazenar o número de posições preenchidas no *buffer*. As filas desses semáforos armazenam os descritores de tarefas forçadas a esperar pelo acesso ao *buffer*. A fila `emptyspots` armazenará tarefas à espera por posições disponíveis no *buffer*; a fila `fullspots` armazenará tarefas 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 os dados 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`, que deve incluir o decremento do 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 completado 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 em 1. Se o `buffer` estiver vazio, o processo chamador é colocado na fila de `fullspots` para esperar até que um item apareça. Quando `FETCH` terminar, 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 `wait` é usado para testar o contador de uma variável semáforo. Se o valor for maior do que zero, o chamador pode continuar sua operação. Nesse caso, o valor do contador da variável semáforo é decrementado para a existência agora de 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 deve ser dada ao processador alguma outra tarefa pronta.

O subprograma `release` é usado por uma tarefa para permitir a alguma 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, 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 de pseudocódigo de `wait` e `release`:

```
wait (umSemaforo)
  if contador de umSemaforo > 0 then
    decrementar o contador de umSemaforo
  else
    colocar o chamador na fila de umSemaforo
    tentar transferir o controle para alguma outra tarefa
    (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 múltiplas 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. Assuma que o *buffer* tem tamanho BUflen, e as rotinas que na verdade o manipulam já existem 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 *fullspots.count*.

```

semaphore fullspots, emptyspots;
fullspots.count = 0;
emptyspots.count = BUflen;
task producer;
    loop
        -- Produz VALUE --
        wait(emptyspots);      { espera por um espaço }
        DEPOSIT(VALUE);
        release(fullspots);   { aumenta os espaços preenchidos }
    end loop;
end producer;

task consumer;
    loop
        wait(fullspots);      { certifica-se de que não está vazio }
        FETCH(VALUE);
        release(emptyspots); { aumenta os espaços vazios }
        -- Consome 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 está sendo acessado. Se o contador do semáforo tem um valor de 0, existe um acesso atual ocorrendo, e a tarefa é colocada na fila do semáforo. Note que o contador do semáforo deve ser inicializado para 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*. Note mais uma vez 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);      { esperar por um espaço }
wait(access);          { esperar por acesso }
DEPOSIT(VALUE);
release(access);        { liberar acesso }
release(fullspots);    { aumentar espaços preenchidos }
end loop;
end producer;

task consumer;
loop
wait(fullspots);       { garantir que não esteja vazio }
wait(access);          { esperar por acesso }
FETCH(VALUE);
release(access);        { liberar acesso }
release(emptyspots);   { aumentar espaços vagos }
-- consumir VALUE --
end loop;
end consumer;
```

Uma breve olhada nesse exemplo pode levar alguém a acreditar que existe um problema com ele. 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 apenas um aspecto crucial dos semáforos até agora não discutido. Lembre-se da descrição inicial do problema da sincronização de competição: operações em dados compartilhados não podem ser sobrepostas. Se uma segunda operação puder iniciar enquanto uma operação anterior ainda estiver em progresso, os dados compartilhados podem ser corrompidos. Um semáforo é um objeto de dados compartilhado, logo as operações nos semáforos

também são suscetíveis a problema. É essencial que as operações de semáforo não possam ser interrompidas. Muitos computadores têm instruções não interrompíveis 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 existe uma maneira de verificar estaticamente a corretude de seu uso, que depende da semântica do programa nos quais eles aparecem. No exemplo do *buffer*, deixar a sentença `wait (emptyspots)` de fora da tarefa produzir resultaria em um transbordamento do *buffer*. Deixar `wait (fullspots)` de fora da tarefa `consumer` resultaria em um transbordamento negativo do *buffer*. Deixar de fora qualquer uma das liberações resultaria em um impasse. Essas são as falhas de sincronização de cooperação.

Os problemas de confiabilidade que os semáforos causam ao fornecer sincronização de cooperação também surgem quando são usados para sincronização de competição. Deixar de fora a sentença `wait (access)` em qualquer uma das tarefas pode causar acessos inseguros ao *buffer*. Deixar de fora a sentença `release (access)` em qualquer das tarefas resultaria em um impasse. 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 as estruturas de dados compartilhadas serem tipos de dados abstratos. Essa solução pode fornecer sincronização de competição sem semáforos ao transferir a responsabilidade de sincronização para o sistema de tempo de execução.

13.4.1 Introdução

Quando os conceitos de abstração de dados estavam sendo 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 o Pascal Concorrente (Brinch Hansen, 1975). Modula (Wirth, 1977), CSP/k (Holt et al., 1978) e Mesa (Mitchell et al., 1979) também fornecem monitores. Dentre as linguagens contemporâneas, eles são suportados por Ada, Java e C#, todas discutidas neste capítulo.

13.4.2 Sincronização de competição

Um dos recursos mais importantes dos monitores é que os dados compartilhados são residentes no monitor, em vez de em uma das unidades clientes. O programador não sincroniza o acesso mutuamente exclusivo aos dados compartilhados pelo uso de semáforos ou de 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 enfileiradas se ele estiver ocupado no momento da chamada.

13.4.3 Sincronização de cooperação

Apesar de o acesso mutuamente exclusivo aos dados compartilhados ser intrínseco com um monitor, a cooperação entre processos ainda é tarefa do programador. Em particular, o programador deve garantir que um *buffer* compartilhado não sofra de transbordamentos positivos ou negativos. Diferentes linguagens fornecem diferentes maneiras de programar a sincronização de cooperação, onde todas são relacionadas aos semáforos.

Um programa contendo quatro tarefas e um monitor que fornece acesso sincronizado a um *buffer* compartilhado concorrentemente é mostrado na Figura 13.2. Nessa figura, a interface para o monitor é mostrada como duas caixas rotuladas inserir e remover (para a inserção e remoção de dados).

13.4.4 Avaliação

Monitores são uma forma melhor de fornecer sincronização de competição do que os semáforos, principalmente por causa dos problemas dos semáforos, conforme discutidos na Seção 13.3. A sincronização de cooperação ainda é um problema com os 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 monitores podem ser usados para implementar semáforos.

Ada fornece duas maneiras para implementar monitores. Ada 83 inclui um modelo geral de tarefas podendo usado para suportar monitores. Ada 95 adicionou uma maneira mais limpa e eficiente de construir monitores, chamada de *objetos protegidos*. Ambas as abordagens usam passagem de

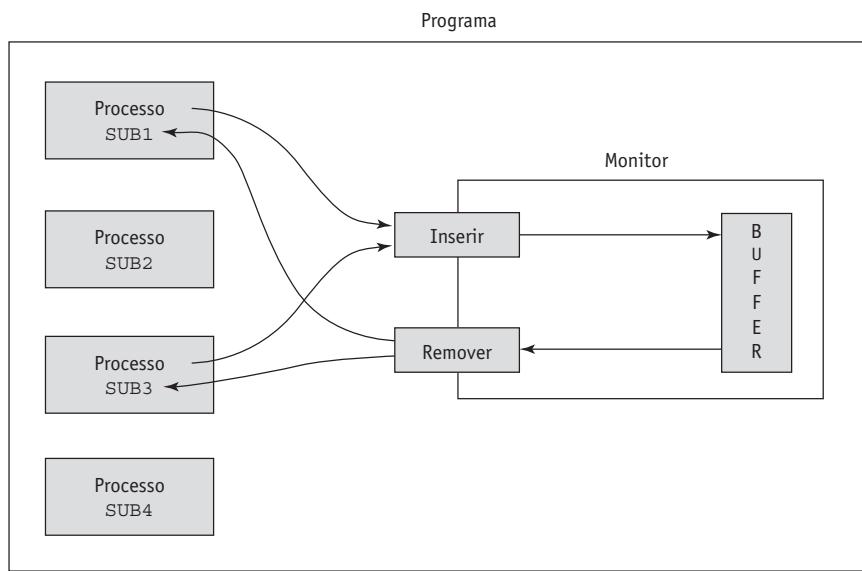


Figura 13.2 Um programa usando um monitor para controlar o acesso a um *buffer* compartilhado.

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.

13.5 PASSAGEM DE MENSAGENS

Esta seção introduz o conceito fundamental de passagem de mensagens.

13.5.1 Introdução

Os primeiros esforços para projetar linguagens que fornecem a capacidade para passagem de mensagens entre tarefas concorrentes foram os de Brich Hansen (1978) e Hoare (1978). Eles também desenvolveram uma técnica para tratar do problema de o que fazer quando múltiplas requisições simultâneas

eram feitas por outras tarefas para se comunicarem com uma dada tarefa. Foi decidido que alguma forma de não determinismo era necessária para fornecer justiça na escolha de qual requisição seria atendida primeiro. Essa justiça pode ser definida de várias maneiras, mas em geral significa que é dada uma chance igual para todos os requisitantes de comunicar com uma tarefa (assumindo que todos os requisitantes têm a mesma prioridade). Construções não determinísticas para controle no nível de sentenças, chamadas de *comandos protegidos*, foram introduzidas por Dijkstra (1975) e são a base para a construção projetada para controlar a passagem de mensagens (comandos protegidos são discutidos no Capítulo 8).

13.5.2 O conceito de passagem síncrona de mensagens

A passagem de mensagens pode ser síncrona ou assíncrona. A passagem assíncrona de mensagens de Ada 95 é descrita na Seção 13.6.8. Aqui, descrevemos a passagem síncrona. O conceito básico da passagem síncrona de mensagens é 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, impedindo o processamento atual de B. Além disso, as mensagens normalmente causam processamento associado no receptor, o que não seria desejado se outro processamento está 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 abordagem é de alguma forma parecida com a de um executivo que instrui sua secretaria para colocar em espera todas as chamadas até que outra atividade, talvez uma conversa importante, estiver terminada. Posteriormente, quando a conversação atual estiver completa, o executivo diz à secretaria que está disposto a conversar com uma das pessoas colocadas em espera.

Uma tarefa pode ser projetada de forma a suspender sua execução em um determinado momento, seja porque está desocupada ou porque precisa de informações de outra unidade antes de poder continuar, assim como uma pessoa esperando por uma chamada importante. Em alguns casos, não existe nada a fazer, além de se 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 um *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 oferece suporte apenas para a passagem síncrona de mensagens; Ada 95 adiciona suporte para a passagem assíncrona.

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 estão enviando 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 do que monitores. Monitores são entidades passivas que fornecem serviços de gerenciamento para os dados compartilhados que armazenam. Eles fornecem seus serviços, apesar de o fazermos apenas quando são solicitados. Quando usadas para gerenciar dados compartilhados, as tarefas Ada podem ser vistas como gerentes que podem residir com os recursos que gerenciam. Elas têm diversos mecanismos, alguns determinísticos e outros não determinísticos, que permitem a elas escolherem entre requisições que competem pelo acesso aos seus recursos.

O formato de tarefas Ada é similar ao dos pacotes Ada. Existem duas partes – 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 fazem parte de 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 também 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 uma 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 **accept**, as quais são introduzidas pela palavra reservada **accept**. Uma **cláusula accept** é definida como a faixa de sentenças que começa com a palavra reservada **accept** e termina com a palavra reservada correspondente **end**. Cláusulas **accept** são relativamente simples, mas outras construções nas quais elas podem ser embutidas podem tornar sua semântica complexa. Uma cláusula **accept** simples tem a forma

```
accept nome_entrada (parâmetros formais) do
...
end nome_entrada;
```

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**. Essas sentenças juntas são chamadas de **corpo da cláusula accept**. Durante o **rendezvous** real, a tarefa enviadora é suspensa.

Sempre que uma cláusula **accept** receber uma mensagem que ela não está disposta a aceitar, por qualquer razão, a tarefa enviadora deve ser suspensa até que a cláusula **accept** na tarefa receptora estiver pronta para aceitar a mensagem. É claro, a cláusula **accept** deve também lembrar as tarefas que enviaram mensagens não aceitas. Para esse propósito, cada cláusula **accept** em uma tarefa possui uma fila associada a ela para armazenar uma lista de outras tarefas que tentaram sem sucesso se comunicarem com ela.

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** estiver suspensa em seu **accept**, um **rendezvous** ocorre e o corpo da cláusula **accept** é executado. Então, por causa do 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.

Um **rendezvous** pode ocorrer em duas maneiras básicas nesse exemplo simples. Primeiro, a tarefa recebedora, **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 recebedora pode estar ocupada com um *rendezvous*, ou com algum outro processamento não associado com um *rendezvous*, quando outra tarefa tenta enviar uma mensagem para a mesma entrada. Nesse caso, o enviaor é suspenso até que o receptor esteja livre para aceitar essa mensagem em um *rendezvous*. Se diversas mensagens chegarem quando o receptor estiver ocupado, os enviadores são enfileirados para esperar sua vez por um *rendezvous*.

Os dois *rendezvous* descritos agora há pouco são ilustrados com os diagramas de linha do tempo na Figura 13.3.

As tarefas que não têm pontos de entrada são chamadas de **tarefas atrizes** (*actor tasks*), pois não precisam esperar por um *rendezvous* para realizar seu trabalho. Elas podem realizar *rendezvous* com outras tarefas pelo envio de mensagens a elas. Em contraste, uma tarefa pode ter cláusulas **accept**, mas não algum 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. Entretanto, 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 conhe-

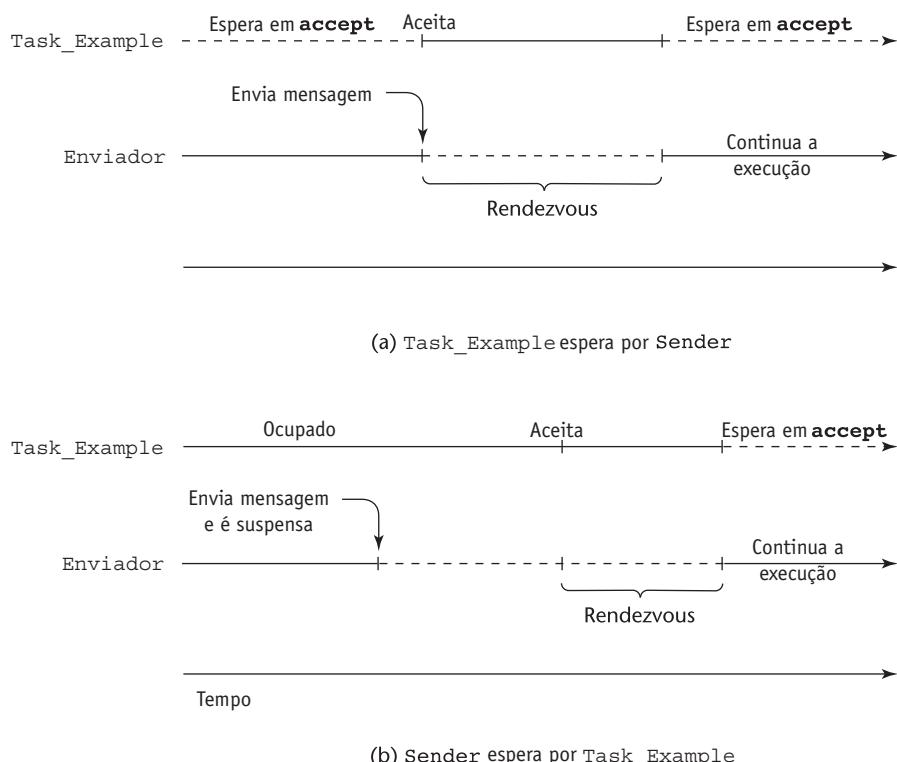


Figura 13.3 Duas maneiras pela qual um *rendezvous* com **Task_Example** pode ocorrer.

cida como CSP, ou Processos Sequenciais Comunicantes (Communicating Sequential Processes) (Hoare, 1978). Em CSP, 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 bibliotecas de tarefas não podem ser construídas 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 13.4.

As tarefas são declaradas na parte de declaração de um pacote, subprograma ou bloco. Tarefas criadas estaticamente¹ começam sua execução no mesmo momento das sentenças no código com o 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 pela qual as cláusulas `accept` associadas aparecem na tarefa dita a ordem pela 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 envolver as entradas. Por exemplo,

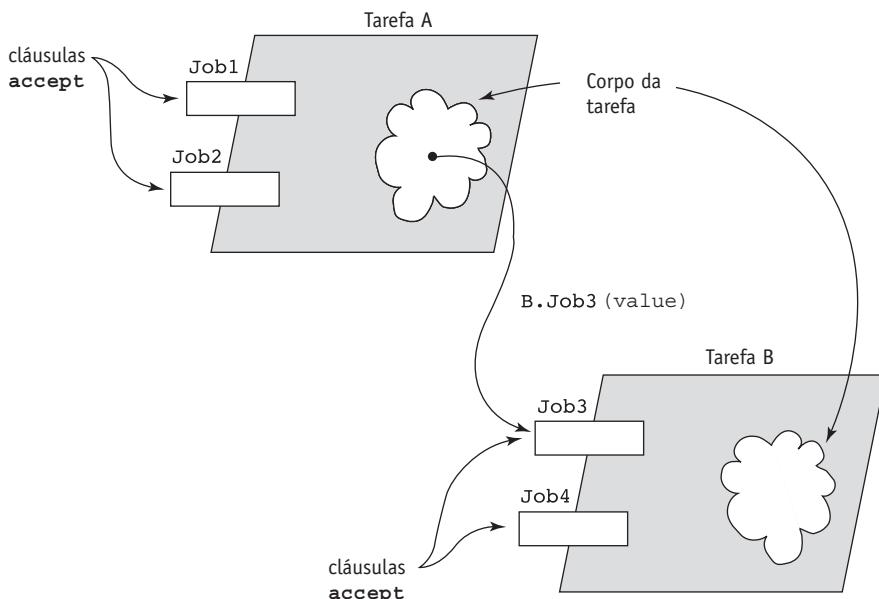


Figura 13.4 Representação gráfica de um *rendezvous* causado por uma mensagem enviada da tarefa A para a tarefa B.

¹ As tarefas também podem ser criadas dinamicamente, mas essas não são cobertas aqui.

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 contiver ao menos uma mensagem esperando (cliente), a cláusula **accept** associada com a mensagem (ou mensagens) que está esperando 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 das cláusulas **accept** é escolhida não deterministicamente para ter um *rendezvous* com um de seus chamadores. O laço força a sentença **select** para 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**. A cláusula **accept** estendida é executada apenas após a cláusula **accept** associada (que a precede imediatamente) ser executada. Essa execução da cláusula **accept** estendida não é parte do *rendezvous* e pode ocorrer concorrentemente com a execução da tarefa chamadora. O enviador é suspenso durante o *rendezvous*, mas é reiniciado (colocado 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` está 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 aberta está disponível para *rendezvous*; uma fechada não pode realizar um *rendezvous*.

Suponha que existam diversas cláusulas `accept` guardadas em uma cláusula `select`. `select` é normalmente colocada em um laço infinito, fazendo ela ser executada repetidamente, com cada cláusula `when` avaliada em cada repetição. Cada repetição faz ser construída uma lista de cláusulas `accept` abertas. Se exatamente 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` possuírem filas não vazias, uma das filas é escolhida não deterministicamente, uma mensagem é retirada dessa fila, 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, ocorre um erro ou exceção em tempo de execução. Essa possibilidade pode ser evitada se certificando de que uma das cláusulas `when` está sempre verdadeira ou adicionando 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 atingido declarando 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 um 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, ambas as cláusulas **accept** são estendidas, assim, 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

```

```
-- produce 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 Término de tarefas

A execução de uma tarefa está **completa** se o controle alcançou o final de seu corpo de código. Isso pode ocorrer porque uma exceção foi lançada e não existe nenhum manipulador (o tratamento de exceções em Ada é descrito no Capítulo 14). Se uma tarefa não criou outras, chamadas de *dependentes*, ela é terminada quando sua execução estiver completa. Uma tarefa que criou tarefas dependentes é terminada quando a execução de seu código estiver completa e todas as suas dependentes estiverem terminadas. Uma tarefa pode terminar sua execução esperando em uma cláusula **terminate** aberta. Nesse caso, a tarefa é terminada apenas quando sua tarefa (ou bloco, ou subprograma) mestre (que a criou) e todas as que dependam desse mestre tiverem ou sido completadas ou estiverem esperando em uma cláusula **terminate** aberta. Nesse caso, todas essas tarefas são terminadas simultaneamente. Um bloco ou subprograma não é deixado até que todas as suas tarefas dependentes estejam terminadas.

13.6.5 Prioridades

Tanto as tarefas nomeadas quanto as anônimas podem ter prioridades atribuídas a elas. Isso é feito com um *pragma*², como em

```
pragma Priority(static expression);
```

A expressão estática é normalmente um literal inteiro ou uma constante pré-definida. O valor da expressão especifica a prioridade relativa para a tarefa ou definição de tipo de tarefa na qual ele aparece. A faixa possível de valores de prioridade é dependente da implementação. A prioridade mais alta possível pode ser especificada com o atributo `Last`, e o tipo `priority`, definido em `System` (um pacote pré-definido). Por exemplo, as linhas a seguir especificam a prioridade mais alta em qualquer implementação:

```
pragma Priority(System.Priority'Last);
```

² Lembre-se de que um *pragma* é uma instrução para o compilador.

Quando são atribuídas prioridades às tarefas, elas são usadas pelo escalonador de tarefas para determinar qual escolher da fila de tarefas prontas quando a que está sendo atualmente executada for bloqueada, alcançar o final de seu tempo alocado ou completar sua execução. Além disso, se uma tarefa com uma prioridade mais alta do que a da atualmente em execução entra na fila de tarefas prontas, aquela com prioridade mais baixa é suspensa e a de prioridade mais alta inicia sua execução (ou continua, se já esteve em execução). Uma tarefa suspensa perde o processador e é colocada na fila de tarefas prontas.

13.6.6 Semáforos binários

Se o acesso à estrutura de dados deve ser controlado e ela não é encapsulada em uma tarefa, deve ser usada outra forma de fornecer acesso mutuamente exclusivo. Uma maneira é construir uma tarefa como um semáforo binário para usar com a que referencia a estrutura de dados. A tarefa como um semáforo binário poderia ser definida como segue:

```
task Binary_Semaphore is
    entry Wait;
    entry Release;
end Binary_Semaphore;

task body Binary_Semaphore is
begin
    loop
        accept Wait;
        accept Release;
    end loop;
end Binary_Semaphore;
```

O propósito dessa tarefa é garantir que as operações `Wait` e `Release` ocorram de maneira alternada.

A tarefa `Binary_Semaphore` ilustra as simplificações possíveis quando mensagens Ada são passadas apenas para sincronização, em vez de também para passar dados. Especificamente, note a forma simples das cláusulas `accept` que não precisam de corpos.

O uso da tarefa `Binary_Semaphore` para fornecer acesso mutuamente exclusivo para uma estrutura de dados compartilhada ocorreria exatamente como com o uso de semáforos no programa de exemplo da Seção 13.3. É claro, esse uso de semáforos sofre de todos os problemas lá discutidos.

Como os semáforos, os monitores podem ser simulados com o modelo de tarefas de Ada. As tarefas fornecem acesso mutuamente exclusivo implícito, exatamente como os monitores. Logo, o modelo de tarefas de Ada suporta tanto semáforos quanto monitores.

13.6.7 Objetos protegidos

Como vimos, o acesso a dados compartilhados pode ser controlado envolvendo os dados em uma tarefa e permitindo o acesso apenas por meio de entradas de tarefas, as quais fornecem implicitamente sincronização de competição. Um problema com esse método é a dificuldade de implementar o mecanismo de *rendezvous* de forma eficiente. Os objetos protegidos de Ada 95 dão 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 tanto por subprogramas protegidos quanto por entradas sintaticamente similares às cláusulas `accept` em tarefas³. Os subprogramas protegidos podem ser tanto procedimentos protegidos, os quais fornecem acesso de leitura e escrita mutuamente exclusivo para os dados do objeto protegido, quanto funções protegidas, as quais fornecem acesso concorrente somente leitura para esses dados. 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 usando o 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);
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;
```

³ Entradas em corpos de objetos protegidos usam a palavra reservada `entry`, em vez do `accept` usado no corpo das tarefas.

```

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;

```

13.6.8 Passagem de mensagens assíncronas

O mecanismo de *rendezvous* descrito até agora nesta seção é estritamente síncrono; tanto o enviador quanto o receptor devem estar prontos para comunicação antes de se comunicarem realmente pelo *rendezvous*.

Uma tarefa pode ter uma cláusula **select** especial, chamada de **select assíncrono**, capaz de reagir imediatamente a mensagens de outras tarefas. Tal cláusula pode ter uma de duas alternativas de disparo diferentes: uma chamada de entrada ou uma sentença **delay**. Além da parte do disparo, a cláusula de seleção assíncrona tem uma parte abortável, podendo conter qualquer sequência de sentenças Ada. A semântica de uma cláusula de seleção assíncrona é executar apenas uma de suas duas partes. Se o evento de disparo ocorrer (seja pela recepção de uma chamada de entrada (**entry**) ou se o tempo de espera (**delay**) terminar), ela executa essa parte. Caso contrário, executa a cláusula abortável. Os dois exemplos a seguir das cláusulas **select** assíncronas aparecem no manual de referência de Ada 95 (ARM, 1995). No primeiro segmento de código, a cláusula abortável é executada repetidamente (por causa do laço) até a chamada a `Terminal.Wait_For_Interrupt` ser recebida. No segundo segmento de código, a função chamada na cláusula abortável executa por ao menos 5 segundos. Se ela não finalizar nesse tempo, o **select** é deixado.

```

-- Laço de comando principal para um interpretador de comandos
loop
  select
    Terminal.Wait_For_Interrupt;
    Put_Line("Interrupted");
  then abort
    -- Isso será abandonado quando ocorrer uma interrupção no
    -- terminal
    Put_Line("-> ");
    Get_Line(Command, Last);
    Process_Command(Command (1..Last));
  end select;
end loop;

-- Um cálculo com tempo limitado
select
  delay 5.0;
  Put_Line("Calculation does not converge");
then abort
  -- Esse cálculo deve terminar em 5 segundos;

```

```
-- se não, assume-se que ele diverge.  
Horribly_Complicated_Recursive_Function(X, Y);  
end select;
```

13.6.9 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 dados compartilhados sincronizados.

Na ausência de processadores distribuídos com memórias independentes, a escolha entre monitores e tarefas com passagem de mensagens como uma forma de implementar dados compartilhados em um ambiente concorrente é uma questão de gosto. Entretanto, no caso de Ada, os objetos protegidos são claramente melhores do que as tarefas para suportar acesso concorrente a dados compartilhados. Não só o código é mais simples; ele também é 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 rodam no mesmo espaço de endereçamento. Elas são diferentes das tarefas em Ada, as quais são linhas de execução pesadas (elas rodam em seus próprios espaços de endereçamento). Um resultado importante dessa diferença é que as linhas de execução de Java requerem muito menos sobrecarga do que as tarefas de Ada.

Existem duas maneiras de definir uma classe com um método `run`. Uma é definir uma subclasse da classe pré-definida `Thread` e sobreescrivar 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, uma subclasse que herda de seu pai natural e implementa a interface `Runnable` é definida. `Runnable` fornece o protocolo do método `run`, então qualquer classe que implementa `Runnable` deve definir `run`. Um objeto da classe que implementa `Runnable` é passado para o construtor de `Thread`. Essa abordagem ainda assim requer um objeto `Thread`, como veremos no exemplo da Seção 13.7.4.

As linhas de execução de Java podem ser usadas para implementar monitores, conforme discutiremos na Seção 13.7.3.

As linhas de execução de Java são um tópico complexo – esta seção fornece apenas uma introdução às partes 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 o programador criar programas concorrentes em Java. Conforme mencionado previamente, a Seção 13.7.4 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 sobreescrito por subclasses de `Thread`. O método `start` de `Thread` inicia sua linha de execução como uma unidade concorrente chamando seu método `run`⁴. A chamada a `start` não é usual no sentido de que o controle retorna imediatamente ao chamador, que continua sua execução, em paralelo com o método `run` recém-criado.

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();
```

Quando um programa aplicativo Java (diferentemente de um *applet*) começa sua execução, uma nova linha de execução é criada (na qual o método `main` rodará) e `main` será chamado. Logo, todos os programas Java rodam em linhas de execução.

Quando um programa tem múltiplas linhas de execução, um escalonador deve determinar qual linha ou quais linhas rodarão em um dado momento. Na maioria dos casos, existe apenas um processador, então apenas uma linha de execução roda realmente nesse momento. É difícil de dar uma descrição precisa de como o escalonador Java funciona, porque as diferentes implementações (Solaris, Windows, e assim por diante) não necessariamente escalonam as linhas de execução exatamente da mesma forma. Entretanto, o escalonador normalmente dá fatias de tempo de tamanho igual a cada linha de execução pronta de maneira similar a um algoritmo *round-robin*, assumindo 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 re-

⁴ Chamar o método `run` diretamente nem sempre funciona, porque a inicialização algumas vezes necessária é incluída no método `start`.

quisição da linha de execução que está rodando para voluntariamente desistir do processador⁵.

A linha de execução é imediatamente colocada na fila de tarefas prontas, tornando-a apta a rodar. O escalonador escolhe a linha de execução com a prioridade mais alta da fila de tarefas prontas. Se não existirem outras linhas de execução prontas com prioridade mais alta aquela que desistiu do processador, ela pode também ser a próxima linha de execução a obter o processador.

O método `sleep` tem um único parâmetro, isto é, o número inteiro de milissegundos que 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 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 que a linha de execução *não* estará em execução. O método `sleep` pode lançar uma `InterruptedException`, devendo 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 se completado. O método `join` é usado quando o processamento de um método não pode continuar até que o trabalho de outra linha de execução esteja completo. Por exemplo, poderíamos ter o seguinte método `run`:

```
public void run() {  
    ...  
    Thread myTh = new Thread();  
    myTh.start();  
    // fazer parte da computação dessa linha de execução  
    myTh.join(); // Esperar por myTh ser completada  
    // fazer o resto da computação dessa linha de execução  
}
```

O método `join` coloca a linha de execução que o chama no estado bloqueado, podendo ser finalizado apenas pelo término da linha de execução na qual `join` foi chamado. Se 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 a linha de execução ser chamada seja completa. Por exemplo,

```
myTh.join(2000);
```

fará a linha de execução chamadora esperar 2 segundos para que `myTh` seja completada. Se ela não tiver completado sua execução após 2 segundos, a linha de execução chamadora é colocada novamente na fila de tarefas prontas, ou seja, ela continuará sua execução tão cedo for possível (de acordo com o escalonamento).

⁵ O método `yield` é definido como uma “sugestão” para o escalonador, que pode ou não segui-la (embora normalmente o faça).

As versões iniciais de Java incluíam três outros métodos de `Thread`: `stop`, `suspend` e `resume`. Todos foram marcados como depreciados por problemas de segurança. O método `stop` às vezes é sobreescrito com um método simples que destrói a linha de execução ao configurar sua variável de referência para `null`.

A maneira normal pela qual um método `run` termina sua execução é ao alcançar o final de seu código. Entretanto, em muitos casos, as linhas de execução rodam 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 ela 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 verificado com o método predicable, `isInterrupted`. Essa não é uma solução completa, porque a linha de execução que alguém está tentando interromper pode estar dormindo ou esperando no momento em que o método `interrupt` é chamado, ou seja, ela não estará verificando para ver se foi interrompida. Para essas situações, o método `interrupt` também 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 ela pode terminar. A linha de execução não pode perder a interrupção, porque se ela estava dormindo ou esperando quando a interrupção ocorreu, será acordada pela interrupção. Na verdade, existem mais detalhes para as ações e os usos de `interrupt`, mas eles não são cobertos 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`, normalmente é 5. `Thread` define duas outras constantes de prioridade, `MAX_PRIORITY` e `MIN_PRIORITY`, cujos valores são 10 e 1, respectivamente⁶. A prioridade de uma linha de execução pode ser modificada com o método `setPriority`. A nova prioridade pode ser qualquer uma das constantes pré-definidas 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`.

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 terminada, ou sua fatia de tempo expira, o escalonador escolhe a linha de execução da fila de tarefas

⁶ O número de prioridades é dependente de implementação, então podem existir mais ou menos níveis que 10 em algumas implementações.

prontas com a prioridade mais alta. Uma linha de execução com uma baixa prioridade rodará apenas se uma de mais alta prioridade não estiver na fila de tarefas prontas quando surgir a oportunidade.

13.7.3 Sincronização de competição

Os métodos 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 rodar em tal objeto. A sincronização de competição em um objeto é implementada ao especificarmos os métodos que acessam dados compartilhados como sincronizados. O mecanismo de sincronização é implementado como segue: cada objeto Java tem um cadeado. Métodos sincronizados devem adquirir o cadeado do objeto antes de ser permitido a eles executarem, previnindo outros métodos sincronizados de executarem no objeto durante esse tempo. Um método sincronizado libera o cadeado do objeto sobre o qual ele está rodando quando completa sua execução. 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 (**synchronized**), previnindo um de interfirir no outro enquanto estiverem executando sobre o mesmo objeto, mesmo se eles forem 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.

Em alguns casos, o número de sentenças que tratam com a estrutura de dados compartilhada é significativamente menor do que o número de outras sentenças na qual ela reside. Nesses casos, é melhor sincronizar o segmento de código que acessa ou 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

```
synchronized (expressão) {  
    sentenças  
}
```

onde a expressão 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 para ele deve ter uma fila associada capaz de armazenar os métodos sincronizados que tentaram executar nele enquanto estava sendo operado por outro desses métodos. Essas filas são implicitamente fornecidas. Quando um método sincronizado termina sua execução em um objeto, um método esperando na fila de espera de um objeto, se existir, é colocado na fila de tarefas prontas.

13.7.4 Sincronização de cooperação

A sincronização de cooperação em Java é realizada com o uso dos 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` no objeto. O método `notify` é chamado para informar uma linha de execução em espera que o 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 causa disso, com o fato de que as linhas de execução em espera podem estar esperando 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, iniciando sua execução logo após sua chamada a `wait`.

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. Por causa do uso de `notifyAll`, alguma outra linha de execução pode ter modificado a condição para `false` desde a última vez em que foi testada.

O método `wait` pode lançar `InterruptedException`, uma descendente de `Exception` (o tratamento de exceções em Java é discutido no Capítulo 14). Logo, qualquer código que chama `wait` deve também capturar `InterruptedException`. Assumindo a condição esperada, chamada `theCondition`, a maneira convencional de usar `wait` é:

```
try {
    while (!theCondition)
        wait();
    -- Faça 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
// valores 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) {
        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() {
        int item = 0;
        try {
            while (filled == 0)
                wait();
            item = que [nextOut];
            nextOut = (nextOut % queSize) + 1;
            filled--;
            notifyAll();
        } //** end of try clause
        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

```
class Producer implements Runnable { ... }
```

Para criar e rodar 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 o objeto *Producer* é passado para o construtor de *Thread*.

As capacidades de Java para controlar a execução de linhas de execução foi aumentada significativamente com a disponibilização do pacote *java.util.concurrent* em 2004. Entretanto, essas novas facilidades estão além do escopo deste livro.

13.7.5 Avaliação

O suporte de Java para concorrência é relativamente simples, mas eficaz. Como são linhas de execução pesadas, as tarefas de Ada podem facilmente ser distribuídas para processadores distintos, em particular com memórias diferentes, que poderiam ser em computadores em lugares diversos. Tais tipos de sistemas não são possíveis com as linhas de execução leves 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 uma breve visão geral 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 rodar em sua própria linha de execução. Uma linha de execução C# é criada por meio de um objeto *Thread*. Ao construtor *Thread* deve ser enviada uma instanciação de uma classe representante (*delegate*) pré-definida, *ThreadStart*⁷, para a qual deve ser enviado o método que implementa as ações da linha de execução. Por exemplo, podemos ter

```
public void MyRun1() { ... }

...
Thread myThread = new Thread(new ThreadStart(MyRun1));
```

Como em Java, criar uma linha de execução não inicia sua execução corrente. Mais uma vez, a execução deve ser solicitada por meio de um método, nesse caso chamado *Start*, como em

```
myThread.Start();
```

⁷ Um representante em C# é uma versão orientada a objetos de um ponteiro para função. Nesse caso, ele aponta para o método que queremos rodar na nova linha de execução.

Como em Java, pode-se fazer uma linha de execução esperar pelo término da execução de outra linha de execução antes de continuar, usando o método similarmente chamado de `Join`.

Uma linha de execução pode ser suspensa por uma quantidade de tempo específica usando `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 terminada com o método `Abort`, apesar de ele não matar a linha de execução. Em vez disso, ele lança a exceção `ThreadAbortException`, que a linha de execução pode capturar. Quando a linha de execução captura essa exceção, ela normalmente libera qualquer recurso alocado, e então termina (ao chegar ao final de seu código).

13.8.2 Sincronizando linhas de execução

Existem três maneiras pelas quais as linhas de execução em C# podem ser sincronizadas: pela classe `Interlock`, pela sentença `lock` ou pela classe `Monitor`. Cada um desses mecanismos é projetado para uma necessidade específica. A classe `Interlock` é usada quando as únicas operações que precisam ser sincronizadas são o incremento e o decremento de um inteiro. Essas operações são feitas atomicamente com os dois métodos de `Interlock`, `Increment` e `Decrement`, os quais recebem uma referência a um inteiro como parâmetro. Por exemplo, para incrementar um inteiro chamado `counter` em uma linha de execução, alguém poderia 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 (expressão) {
    // A seção crítica
}
```

A expressão, parecida com um parâmetro para `lock`, é normalmente uma referência para o objeto no qual a linha de execução está rodando, `this`.

A classe `Monitor` define cinco métodos, `Enter`, `Wait`, `Pulse`, `PulseAll` e `Exit`, que podem ser usados para fornecer uma sincronização mais sofisticada de linhas de execução. O método `Enter`, que recebe uma referência a um objeto como seu parâmetro, marca o início da sincronização da linha de execução desse objeto. O método `Wait` suspende a execução da linha e instrui a 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 seu parâmetro, notifica uma linha de execução em

espera que agora tem uma chance de rodar 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.

13.8.3 Avaliação

As linhas de execução em C# fornecem uma leve 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. 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 para `null`). A sincronização da execução das linhas é mais sofisticada em C#, porque a linguagem tem diversos mecanismos, cada um para uma aplicação específica. Linhas de execução em C#, como aquelas de Java, são leves, não podendo ser tão versáteis quanto as tarefas de Ada.

13.9 CONCORRÊNCIA NO NÍVEL DE SENTENÇA

Nesta seção, damos uma breve olhada no projeto de linguagem para concorrência no nível de sentença. Do ponto de vista de projeto de linguagem, o objetivo é fornecer um mecanismo para que o programador possa informar o compilador de maneiras pelas quais ele possa mapear o programa em uma arquitetura multiprocessada⁸.

Nesta seção, é discutida apenas uma coleção de construções linguísticas de uma linguagem para concorrência no nível de sentença. A seguir, essas construções e seus objetivos serão descritos em termos da arquitetura de máquinas SIMD (veja Seção 13.1.1), apesar de as construções terem sido projetadas para serem úteis a uma variedade de configurações arquiteturais.

O problema tratado pelas construções de linguagens discutidas é o de minimizar a comunicação requerida entre processadores e as memórias de outros processadores. A premissa é ser mais rápido para um processador acessar dados em sua própria memória do que de outro processador. Compiladores bem projetados podem ajudar bastante nesse processo, mas muito mais pode ser feito se o programador for capaz de fornecer informações ao compilador sobre a possível concorrência que poderia ser empregada.

13.9.1 Fortran de alto desempenho

O Fortran de Alto Desempenho (HPF – High-Performance Fortran) (HPF; ACM, 1993b) é uma coleção de extensões ao Fortran 90 feitas para permitir que os programadores especifiquem informações ao compilador para ajudá-lo a otimizar a execução de programas em computadores multiprocessados. O HPF

⁸ Apesar de o ALGOL 68 incluir um tipo semáforo desenvolvido para lidar com concorrência no nível de sentença, não discutimos esta aplicação de semáforos aqui.

inclui tanto novas sentenças de especificação quanto subprogramas intrínsecos, pré-definidos. Esta seção discute apenas algumas das sentenças do HPF.

As principais sentenças de especificação do HPF são para especificar 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. As sentenças de especificação do HPF aparecem como comentários especiais em um programa Fortran. Cada uma é introduzida pelo prefixo !HPF\$, onde ! é o caractere usado para iniciar linhas de comentários em Fortran 90. Esse prefixo as torna invisíveis para os compiladores Fortran 90, mas fáceis para os compiladores HPF as reconhecerem.

A especificação PROCESSORS tem a 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 com os processadores.

A sentença DISTRIBUTE especifica quais dados serão distribuídos e o tipo de distribuição a ser usada. Sua forma é

```
!HPF$ DISTRIBUTE (tipo) ONTO procs :: lista_de_identificadores
```

Nessa sentença, o tipo pode ser ou bloco (BLOCK) ou cíclica (CYCLIC). A lista de identificadores são os nomes das variáveis matrizes que serão distribuídas. Uma variável especificada para ser distribuída em bloco (BLOCK) é dividida em n grupos iguais, onde cada um consiste em coleções contíguas de elementos de matriz igualmente distribuídos nas memórias de todos os processadores. Por exemplo, se uma matriz com 500 elementos chamada LIST é distribuída em bloco em cinco processadores, os primeiros 100 elementos de LIST 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, o primeiro elemento de LIST 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 informação 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 em particular da máquina alvo.

A sentença `FORALL` especifica uma coleção de sentenças que podem ser executadas concorrentemente. Por exemplo,

```
FORALL (index = 1:1000) list_1 (index) = list_2 (index)
```

especifica a atribuição dos elementos de `list_2` aos elementos correspondentes de `list_1`. Conceitualmente, ela especifica que o lado direito de todas as 1.000 atribuições podem ser avaliados primeiro, antes de quaisquer atribuições ocorrerem. Isso permite a execução concorrente de todas as sentenças de atribuição. A sentença HPF `FORALL` é incluída no Fortran 95 e no Fortran 2003.

Discutimos brevemente apenas uma pequena parte das capacidades do HPF. Entretanto, deve ser o suficiente para fornecer ao leitor a ideia dos tipos de extensões de linguagem úteis para programar computadores com muitos processadores.

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 – SIMD ou MIMD. Os computadores MIMD podem ser distribuídos.

Duas das facilidades primárias que as linguagens que suportam concorrência no nível de subprogramas devem fornecer são o acesso mutuamente exclusivo às estruturas de dados compartilhadas (sincronização de competição) e a cooperação entre tarefas (sincronização de cooperação).

As tarefas podem estar em um de cinco estados diferentes: nova, pronta, executando, bloqueada ou morta.

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 fácil usar semáforos incorretamente, resultando em erros que não podem ser detectados pelo compilador, ligador ou sistema de tempo de execução.

Os monitores são abstrações de dados que fornecem uma maneira natural de dar acesso mutuamente exclusivo a estruturas de dados compartilhadas entre tarefas. Eles são suportados por diversas linguagens de programação, dentre elas Ada, Java e C#. A sincronização de cooperação em linguagens com monitores deve ser fornecida com alguma forma de semáforos.

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, a passagem síncrona de mensagens. Um *rendezvous* é a ação de uma tarefa aceitando uma mensagem enviada por outra. Ada inclui tanto métodos simples quanto complicados de controlar a ocorrência de *rendezvous* entre as tarefas.

Ada 95 inclui capacidades adicionais para o suporte à concorrência, primariamente objetos protegidos, e passagem assíncrona de mensagens. Ada 95 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 herda de Thread ou implementa Runnable pode sobreescriver 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 definindo métodos que acessam os dados compartilhados a serem sincronizados. Pequenas seções de código também podem ser 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.

O suporte de C# para concorrência é baseado no de Java, mas é levemente mais sofisticado. Qualquer método pode ser executado em uma linha de execução. Três tipos de sincronização de linhas de execução são suportados com as classes Interlock e Monitor e com a sentença lock.

O 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

O assunto geral de concorrência é discutido em grande extensão 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 do desenvolvimento do modelo de tarefas de Ada pode ser encontrada em Ichbiah et al. (1979). Ada 95 é descrita em detalhes em ARM (1995). O Fortran de Alto Desempenho é descrito 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 duas 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 de suporte linguístico para concorrência, o que é uma guarda?
18. Qual é o propósito de uma fila de tarefas prontas?
19. Quais são as questões de projeto primárias 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 problemas primários com o 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, tarefa atriz, tarefa servidora, cláusula accept estendida, cláusula accept aberta, cláusula accept fechada e tarefa completada*.
26. O que é mais geral, concorrência por monitores ou concorrência por passagem de mensagens?
27. As tarefas em Ada são criadas estaticamente ou dinamicamente?
28. Para que serve uma cláusula **accept** estendida?
29. Como a sincronização de cooperação é fornecida para tarefas Ada?
30. Qual é o propósito de uma cláusula **terminate** em Ada?
31. Qual é a vantagem dos objetos protegidos em Ada 95 em relação às tarefas para fornecer acesso a objetos de dados compartilhados?
32. Defina a cláusula **select** assíncrona em Ada 95.
33. Especificamente, que unidade de programa Java pode rodar concorrentemente com o método principal em um programa aplicativo?
34. As linhas de execução Java são tarefas leves ou pesadas?

35. O que o método Java `sleep` faz?
36. O que o método Java `yield` faz?
37. O que o método Java `join` faz?
38. O que o método Java `interrupt` faz?
39. Quais são as duas construções Java que podem ser sincronizadas?
40. Como a prioridade de uma linha de execução pode ser configurada?
41. Quando o método Java `stop` é sobreescrito, o que a nova versão faz na verdade?
42. Descreva as ações de três métodos Java usados para suportar sincronização de cooperação.
43. Que tipo de objeto Java é um monitor?
44. Explique por que Java inclui a interface `Runnable`.
45. Que tipos de métodos podem rodar em uma linha de execução C#?
46. O que é diferente no método `Sleep` de C#, em relação ao `sleep` de Java?
47. O que exatamente faz o método `Abort` de C#?
48. Qual é o propósito da classe `Interlock` de C#?
49. O que a sentença `lock` de C# faz?
50. Qual é o objetivo das sentenças de especificação do Fortran de Alto Desempenho?
51. Qual é o propósito da sentença `FORALL` do Fortran de Alto Desempenho?

CONJUNTO DE 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 tomar quando um impasse é detectado?
3. Espera ociosa, ou espera ocupada, é um método no qual uma tarefa espera por um evento ao continuamente verificar pela ocorrência desse evento. Qual é o principal problema com essa abordagem?
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 que 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 um dela. Assuma que tais operações aritméticas são feitas pelo processo de três passos de obter o valor atual, realizar a aritmética e colocar o novo valor de volta. Na ausência de sincronização de competição, que sequência de eventos são possíveis e quais são os valores resultantes dessas operações. Assuma 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 usando as cláusulas `when` de Ada em tarefas.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Escreva uma tarefa Ada para implementar semáforos gerais.
2. Escreva uma tarefa 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 do capítulo em C#.
6. O problema do leitor-escritor pode ser definido como segue: uma posição de memória compartilhada pode ser concorrentemente lida por qualquer número de tarefas, mas quando uma tarefa deve 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#.

Capítulo 14

Tratamento de Exceções e Tratamento de Eventos

- 14.1** Introdução ao tratamento de exceções
- 14.2** Tratamento de exceções em Ada
- 14.3** Tratamento de exceções em C++
- 14.4** Tratamento de exceções em Java
- 14.5** Introdução ao tratamento de eventos
- 14.6** Tratamento de eventos com Java

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 que não são pré-determinados, e ambos são mais bem tratados com construções e processos de linguagem especiais. Alguns desses – por exemplo, a propagação – são similares para o tratamento de exceções e o tratamento de eventos.

Primeiro, descrevemos os conceitos fundamentais do tratamento de exceções, incluindo exceções detectáveis por hardware e por software, tratadores de exceções e o levantamento de exceções. Então, as questões de projeto para tratamento de exceções são introduzidas e discutidas, incluindo desde a vinculação até os tratadores de exceções, continuação, tratadores padrão e desabilitação de exceções. Essa seção é seguida por uma descrição e uma avaliação dos recursos de tratamento de exceções de três linguagens de programação: Ada, C++ e Java.

A parte final deste capítulo fala sobre tratamento de eventos. Primeiro apresentamos uma introdução aos conceitos básicos do tratamento de eventos. Esse tópico é seguido por uma breve discussão da abordagem de tratamento de eventos de Java para seus componentes de interface com o usuário (GUI).

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 podia 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 transferido para o sistema operacional. A reação típica do sistema operacional para um erro em tempo de execução é mostrar uma mensagem diagnóstica, a qual pode ser significativa e útil ou altamente críptica. Após mostrar a mensagem, o programa é terminado.

No caso de operações de entrada e saída, entretanto, a situação é de certa forma diferente. Por exemplo, uma sentença `Read` do 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, é claro que a condição nem sempre é considerada um erro. Na maioria dos casos, não passa de um sinal de que um tipo de processamento está completo e outro tipo precisa iniciar. A despeito da diferença óbvia entre o término de um arquivo e eventos que são sempre erros, como um processo falho de entrada, o 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` espe-

cifica que o controle deve ser transferido para a sentença rotulada 999 se a operação de leitura encontra o final do arquivo. Então, o Fortran usa desvios simples tanto para erros de entrada quanto para um término de arquivo.

Existe uma categoria de erros sérios que não são detectáveis por hardware, mas que podem ser pelo código gerado pelo compilador. Por exemplo, os erros de índices de faixas de matrizes quase nunca são detectados por hardware¹, mas levam a sérios erros que geralmente não são notados até mais tarde na execução do programa.

A detecção de erros de faixas de índices é algumas vezes requerida pelo projeto da linguagem. Por exemplo, os compiladores Java normalmente geram código para verificar a corretude de cada uma das expressões de índices (eles não geram tal código quando pode ser determinado em tempo de compilação 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 acredita-se) que o custo da verificação não vale o benefício de detectar tais erros. Em alguns compiladores para algumas linguagens, a verificação de faixa de índices pode ser selecionada (se não for ligada por padrão) ou desabilitada (se for ligada 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 uma maneira padrão a certos erros em tempo de execução, assim como a outros eventos não usuais detectados pelos programas. 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. Tais mecanismos são coletivamente chamados de tratamento de exceções.

Talvez a razão mais plausível 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 o término de arquivos (o qual é também detectado por hardware), como exceções. Estendemos um pouco mais o conceito de uma 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 requerer um processamento especial.

O processamento especial que pode ser requerido quando uma exceção é detectada é chamado de **tratamento de exceção**. Esse processamento é feito por uma unidade de código ou por um segmento chamado de **tratador de exceção** (ou manipulador de exceção). Uma exceção é **levantada** (*raised*)

¹ Nos anos 1970, existiam alguns computadores que *detectavam* erros de faixas de índices em hardware.

quando seu evento associado ocorre. Em algumas linguagens baseadas em C, as exceções são ditas como lançadas (*thrown*), em vez de levantadas².

Diferentes tipos de exceções requerem diferentes tratadores. A detecção de fim de arquivo praticamente sempre requer alguma ação específica do programa. Mas, claramente, tal ação não seria também 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 um término organizado do programa.

Em outras situações, pode ser desejável ignorar certas exceções detectáveis por hardware – por exemplo, divisão por zero – por algum tempo. Essa ação pode ser feita ao desabilitarmos a exceção. Uma exceção desabilitada pode ser habilitada novamente.

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, ou invocador. Um projeto possível é enviar um parâmetro auxiliar, usado como uma variável de estado. É atribuído um valor à variável de estado no subprograma chamado de acordo com a corretude e a normalidade de sua computação. 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 abordagem. Os valores de retorno são usados como indicadores de erro.

Outra possibilidade é passar um rótulo como parâmetro para o subprograma. É claro, essa abordagem 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 um parâmetro para a unidade chamada. Nesse caso, o subprograma tratador é fornecido pelo chamador, mas a unidade chamada o chama quando uma exceção é levantada. Um problema com essa abordagem é ser necessário enviar um subprograma tratador a *todas* as chamadas a *todos* os subprogramas que recebem um subprograma tratador como um 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 for detectada, o tratador é incluído como um segmento de código nessa unidade.

² C++ foi a primeira linguagem baseada em C a incluir tratamento de exceções. A palavra *lançar* (*throw*) foi usada, em vez de levantar (*raise*), porque a biblioteca padrão de C inclui uma função chamada *raise*.

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 requeira a verificação de faixas de índice. Sem essa verificação pré-definida, cada uma dessas operações pode precisar ser precedida por código para detectar um possível erro de faixa e 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, que permite a exceção levantada em uma unidade de programa ser 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 reúso 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 como eles podem ser tratados. Essa abordagem é muito melhor do que não considerar tais possibilidades e simplesmente esperar que nada dê errado. Essa vantagem é relacionada a requerer que uma construção de seleção múltipla inclua ações para todos os valores possíveis da expressão de controle, como requerido por Ada.

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 desses programas se tornariam muito confusas sem tal tratamento.

14.1.2 Questões de projeto

Agora, exploramos algumas das questões de projeto para um sistema de tratamento de exceções quando ele é parte de uma linguagem de programação. Tal sistema deve permitir tanto exceções pré-definidas quanto exceções definidas pelo usuário, assim como tratadores de exceções. Note que

exceções pré-definidas 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) is zero\n");
    }
    ...
}
```

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 pela linguagem. A seguir, 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 pré-definidas. Desde que PL/I foi projetada, uma quantidade substancial de trabalho foi feita para projetar métodos alternativos de tratamento de exceções. Em particular CLU (Liskov et al., 1984), Mesa (Mitchell et al., 1979), Ada, COMMON LISP (Steele, 1984), ML (Milner et al., 1990), C++, Modula-3 (Cardelli et al., 1989), Eiffel, Java, e C# incluem recursos de tratamento de exceções.

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 uma ocorrência de uma exceção é vinculada a um tratador. Essa questão ocorre em dois níveis. No nível de unidade, existe a questão de como a mesma exceção sendo levantada em diferentes pontos em uma unidade pode ser vinculada a diferentes manipuladores dentro da unidade. Por exemplo, no subprograma de exemplo, existe um manipulador para uma exceção de divisão por zero que parece ser escrito para lidar com uma ocorrência de uma divisão por zero em uma sentença em particular (aquele mostrada). Mas suponha que a função inclua diversas outras expressões com operadores de divisão. Para tais operadores, esse manipulador provavelmente não seria apropriado. Então, deve ser possível vincular as exceções que podem ser levantadas por sentenças em particular a manipuladores em particular, mesmo que essa exceção possa ser levantada por muitas sentenças diferentes.

Em um nível mais alto, a questão de vinculação surge quando não existe um tratador de exceções local à unidade na qual a exceção é levantada. Nesse caso, o projetista deve decidir se propaga a exceção para alguma outra unidade, e se esse for o caso, para onde. Como essa propagação ocorre e o quanto longe ela vai têm um impacto importante na facilidade de escrita

de tratadores de exceções. Por exemplo, se os tratadores devem ser locais, muitos 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 requerer que o tratador seja mais geral do que o desejado.

Uma questão relacionada com a 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 essa 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 com 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 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 boa, 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 da exceção. 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 ela encontrar uma exceção. Em algumas situações, é necessário completar alguma computação independentemente de como a execução do subprograma termina. A habilidade 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: 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 elas podem ser levantadas. O escopo de uma exceção declarada é normalmente o escopo da unidade de programa que contém a declaração.

No caso em que uma linguagem fornece exceções pré-definidas, diversas outras questões de projeto aparecem. Por exemplo, o sistema de tempo de execução da linguagem deve fornecer tratadores padrão para as exceções pré-definidas, ou deve ser requerido que o usuário escreva tratadores para todas as exceções? Outra questão é se as exceções pré-definidas podem ser levantadas explicitamente pelo programa do usuário. Esse uso pode ser conveniente

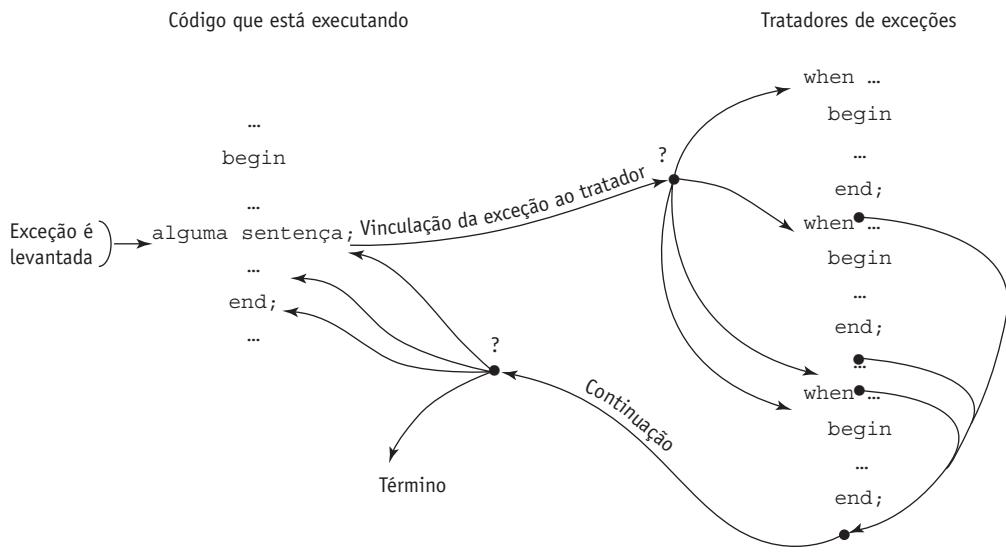


Figura 14.1 Fluxo de controle de tratamento de exceções.

se existirem situações detectáveis por software nas quais o usuário gostaria de usar um tratador pré-definido.

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 pré-definidas, que são implicitamente levantadas por hardware ou por software de sistema.

Por fim, existe a questão se as exceções, sejam elas pré-definidas ou definidas pelo usuário, podem ser temporariamente ou permanentemente desabilitadas. Essa questão é de certa forma filosófica, particularmente no caso de condições de erro pré-definidas. Por exemplo, suponha que uma linguagem tem uma exceção pré-definida lançada quando um erro de faixa de índices ocorre. Muitos acreditam que erros de faixa de índices sempre devem ser detectados, e logo não deve ser possível aos programas desabilitar a detecção desses erros. Outros argumentam que a verificação de faixa de índices é muito cara para software de produção, onde, presumidamente, o código é suficientemente livre de erros, de forma que os erros de faixa nunca devem ocorrer.

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 pré-definidas, devem existir tratadores de exceção padronizados para programas que não forneçam seus próprios?
- As exceções pré-definidas podem ser explicitamente levantadas?
- Os erros detectáveis por hardware são considerados exceções que devem ser tratadas?
- Existe alguma exceção pré-definida?
- Deve ser possível desabilitar exceções pré-definidas?

Estamos agora em uma posição que permite examinar os recursos de tratamento de exceções de três linguagens de programação contemporâneas.

14.2 TRATAMENTO DE EXCEÇÕES EM ADA

O tratamento de exceções em Ada é uma ferramenta poderosa para construir sistemas de software mais confiáveis. Ela é baseada nas boas partes do projeto de tratamento de exceções de duas linguagens anteriores – PL/I e CLU.

14.2.1 Tratadores de exceção

Os tratadores de exceção em Ada são geralmente locais ao código no qual a exceção pode ser levantada (apesar de eles poderem ser propagados para outras unidades de programa). Como isso fornece a eles o mesmo ambiente de referenciamento, não são necessários parâmetros para os tratadores e eles não são permitidos. Logo, se uma exceção é tratada em uma unidade diferente daquela que levantou a exceção, nenhuma informação sobre a exceção pode ser passada para o tratador³.

Os tratadores de exceção têm a seguinte forma geral, dada aqui em EBNF:

when escolha_da_exceção { | escolha_da_exceção} => sequência_de_sentenças

³ Isso não é bem verdade. É possível para o tratador obter o nome da exceção, uma breve descrição da exceção e a posição aproximada onde a exceção foi levantada.

Lembre-se de que as chaves são metassímbolos, ou seja, o conteúdo nelas pode ser deixado de fora ou repetido quantas vezes necessário. A `escolha_da_exceção` tem a forma

`nome_da_exceção | others`

O `nome_da_exceção` indica uma exceção em particular a que esse tratador foi criado para tratar. A sequência de sentenças é o corpo do manipulador. A palavra reservada `others` indica que o tratador foi criado para tratar quaisquer exceções não nomeadas em nenhum outro tratador local.

Os tratadores de exceção podem ser incluídos em blocos ou em corpos de subprogramas, pacotes ou tarefas. Independentemente do bloco ou da unidade na qual eles aparecem, os tratadores são unidos em uma cláusula `exception`, que deve ser colocada no final do bloco ou da unidade. Por exemplo, a forma usual de uma cláusula de exceção é mostrada a seguir:

```
begin
-- o bloco ou corpo de unidade --
exception
  when nome_da_exceção_1 =>
    -- primeiro tratador --
  when nome_da_exceção_2 =>
    -- segundo tratador --
    -- outros tratadores --
end;
```

Qualquer sentença que pode aparecer no bloco ou na unidade na qual o tratador aparece também é legal no tratador.

14.2.2 Vinculando exceções a tratadores

Quando o bloco ou unidade que levanta uma exceção inclui um tratador para ela, a exceção é estaticamente vinculada a esse tratador. Se uma exceção é levantada em um bloco ou unidade que não tem um tratador para ela, a exceção é propagada para outro bloco ou unidade. A maneira pela qual as exceções são propagadas depende da entidade de programa na qual elas ocorrem.

Quando uma exceção é levantada em um procedimento, seja na elaboração de suas declarações ou na execução de seu corpo, e o procedimento não tem um tratador para ela, a exceção é implicitamente propagada para a unidade de programa chamadora no ponto da chamada. Essa política é um reflexo da filosofia de projeto que diz que a propagação de exceção de subprogramas deve rastrear de volta pelo caminho de controle (ancestrais dinâmicos), não pelos ancestrais estáticos.

Se a unidade chamadora para a qual uma exceção foi propagada também não possui um tratador para a exceção, ela é novamente propagada para o chamador dessa unidade. Isso continua, se necessário, até o procedimento principal, ou seja, a raiz dinâmica de todos os programas Ada. Se uma exceção é propagada para o procedimento principal e um tratador não for encontrado, o programa é terminado.

No contexto de tratamento de exceções, um bloco em Ada é considerado um procedimento sem parâmetros “chamado” por seu bloco pai quando o controle da execução alcança a primeira sentença do bloco. Quando uma exceção é levantada em um bloco, seja em suas declarações ou em suas sentenças executáveis, e o bloco não tem um tratador para ela, a exceção é propagada até o próximo escopo estático maior que envolve o bloco, ou seja, o código que o “chamou”. O ponto para o qual a exceção é propagada é logo após o final do bloco no qual ela ocorreu, que é seu ponto de “retorno”.

Quando uma exceção é levantada em um corpo de pacote e tal corpo não possui um tratador para a exceção, ela é propagada para a seção de declaração da unidade contendo a declaração de pacote. Se o pacote é uma unidade de biblioteca (compilada separadamente), o programa é terminado.

Se uma exceção ocorrer no nível mais externo em um corpo de tarefa (não em um bloco aninhado) e a tarefa contiver um tratador para a exceção, esse tratador é executado e a tarefa é marcada como completa. Se a tarefa não tiver um tratador para a exceção, ela é simplesmente marcada como completa; a exceção não é propagada. O mecanismo de controle de uma tarefa é muito complexo para servir como uma resposta razoável e simples para a questão de onde as exceções não tratadas devem ser propagadas.

As exceções também podem ocorrer durante a elaboração das seções declarativas de subprogramas, blocos, pacotes e tarefas. Quando tais exceções são levantadas em procedimentos, pacotes e blocos, elas são propagadas exatamente como se a exceção fosse levantada na seção de código associada. No caso de uma tarefa, ela é marcada como estando completa, nenhuma elaboração adicional ocorre e a exceção pré-definida `Tasking_Error` é levantada no ponto de ativação para a tarefa.

14.2.3 Continuação

Em Ada, o bloco ou unidade que levanta uma exceção, com todas as unidades para as quais a exceção foi propagada, mas não a manipularam, é sempre terminado. O controle nunca retorna implicitamente para o bloco ou unidade que levantou a exceção após esta ser tratada. O controle simplesmente continua após a cláusula de exceção, sempre no final de um bloco ou de uma unidade. Isso gera um retorno imediato para um nível de controle mais alto.

Ao decidir onde a execução continuaria após o término da execução do tratador de exceção em uma unidade de programa, a equipe de projeto

de Ada tinha pouca escolha, porque a especificação de requisitos para Ada (Departamento de Defesa dos EUA, 1980a) dizia que as unidades de programa que lançam exceções não podem ser continuadas ou reiniciadas. Entretanto, no caso de um bloco, uma sentença pode ser retentada após ela lançar uma exceção e esta ser tratada. Por exemplo, suponha que tanto uma sentença capaz de lançar uma exceção quanto um tratador para essa exceção estejam contidos em um bloco, envolto por um laço. O seguinte segmento de código de exemplo, que obtém quatro valores inteiros na faixa desejada do teclado, ilustra esse tipo de estrutura:

```
...
type Age_Type is range 0..125;
type Age_List_Type is array (1..4) of Age_Type;
package Age_IO is new Integer_IO (Age_Type);
use Age_IO;
Age_List : Age_List_Type;
...
begin
for Age_Count in 1..4 loop
    loop -- laço para repetição quando a exceção ocorrer
        Except_Blk:
            begin -- sentença composta para encapsular o tratamento
                   de exceção
                Put_Line("Enter an integer in the range 0..125");
                Get(Age_List(Age_Count));
            exit;
        exception
            when Data_Error => -- A cadeia de entrada não é um
                                 número
                Put_Line("Illegal numeric value");
                Put_Line("Please try again");
            when Constraint_Error => -- A entrada é < 0 ou > 125
                Put_Line("Input number is out of range");
                Put_Line("Please try again");
            end Except_Blk;
        end loop; -- fim do laço infinito para repetir a entrada
                   -- onde existe uma exceção
    end loop; -- fim do for Age_Count in 1..4 loop
    ...

```

O controle permanece no laço interno, que contém apenas o bloco, até um número válido de entrada ser recebido.

14.2.4 Outras escolhas de projeto

Existem quatro exceções definidas no pacote padrão, Standard:

```
Constraint_Error
Program_Error
Storage_Error
Tasking_Error
```

Cada uma delas é uma categoria de exceções. Por exemplo, a exceção `Constraint_Error` é levantada quando um índice de matriz está fora da faixa, quando existe um erro de faixa em uma variável numérica que tem uma restrição de faixa, quando uma referência é feita a um campo de registro que não está presente em uma união discriminada e em muitas outras situações.

Além das exceções definidas em `Standard`, outros pacotes pré-definidos definem outras exceções. Por exemplo, `Ada.Text_IO` define a exceção `End_Error`.

Exceções definidas pelo usuário têm o seguinte formato de declaração

```
lista_de_nomes_das_exceções : exception
```

Tais exceções são tratadas exatamente como pré-definidas, exceto pelo fato de que elas precisam ser levantadas explicitamente. Existem tratadores padrão para as exceções pré-definidas, todos resultando no término do programa.

As exceções são explicitamente levantadas com a sentença `raise`, com a forma geral

```
raise [nome_da_exceção]
```

O único lugar onde uma sentença `raise` pode aparecer sem nomear uma exceção é dentro de um tratador. Nesse caso, ela levanta novamente a mesma exceção que causou a do tratador. Isso tem o efeito de propagar a exceção de acordo com as regras de propagação mencionadas anteriormente. Um `raise` em um tratador de exceções é útil quando alguém quer imprimir uma mensagem de erro na qual uma exceção é levantada, mas quer tratá-la em outro lugar.

Um `pragma` em Ada é uma diretiva para o compilador. Certas verificações em tempo de execução que são partes das exceções pré-definidas podem ser desabilitadas em programas Ada com o uso do `pragma Suppress`, cujo formato simples é

```
pragma Suppress(nome_da_verificação)
```

onde `nome_da_verificação` é o nome de uma verificação de exceção em particular. Exemplos de tais verificações são dados posteriormente neste capítulo.

O `pragma Suppress` pode aparecer apenas em seções de declaração. Quando ele aparece, a verificação especificada pode ser suspensa no bloco ou unidade de programa associado do qual a seção de declaração é uma parte. Levantamentos explícitos não são afetados por `Suppress`. Apesar de não ser obrigatório, a maioria dos compiladores Ada implementa o `pragma Suppress`.

Exemplos de verificações possíveis de serem suprimidas são: `Index_Check` e `Range_Check` especificam duas das verificações que normalmente feitas em um programa Ada; `Index_Check` se refere a verificação de faixas de índices de matrizes; `Range_Check` se refere a verificação de itens como a faixa de um valor sendo atribuído a uma variável de subtipo. Se `Index_Check` ou `Range_Check` são violadas, uma exceção `Constraint_Error` é levantada. `Division_Check` e `Overflow_Check` são verificações suprimíveis associadas

com Numeric_Error. O **pragma** a seguir desabilita a verificação de faixas de índices de matrizes:

```
pragma Suppress(Index_Check);
```

Existe uma opção de suppress que permite a verificação nomeada ser restrita a objetos, tipos, subtipos e unidades de programa em particular.

14.2.5 Um exemplo

O programa de exemplo a seguir ilustra alguns usos simples dos tratadores de exceção em Ada. O programa 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, que levanta uma exceção Constraint_Error porque as notas são do tipo Natural (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 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 As possíveis do que Bs ou Cs é uma evidência conclusiva da generosidade dos professores). A nota 100 é também tratada no mesmo tratador de exceções usado para dados de entrada inválidos.

```
-- 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.
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
procedure Grade_Distribution is
    Freq: array (1..10) of Integer := (others => 0);
    New_Grade : Natural;
    Index,
    Limit_1,
    Limit_2 : Integer;
begin
    Grade_Loop:
    loop
        begin -- Um bloco para a exceção de entrada negativa
            Get(New_Grade);
        exception
            when Constraint_Error => -- para entrada negativa
                exit Grade_Loop;
        end;
        if New_Grade < 0 then
            exit Grade_Loop;
        else
            Index := New_Grade;
            if Index > 10 then
                Index := 10;
            end if;
            if Index < 0 then
                Index := 0;
            end if;
            Freq(Index) := Freq(Index) + 1;
        end if;
    end loop;
    for I in 1..10 loop
        Put(Freq(I));
        Put(" ");
    end loop;
end Grade_Distribution;
```

```

end; -- end of negative input block
Index := New_Grade / 10 + 1;
begin -- Um bloco para o tratador de faixa de índices
Freq(Index) := Freq(Index) + 1;
exception
-- For index range errors
when Constraint_Error =>
  if New_Grade = 100 then
    Freq(10) := Freq(10) + 1;
  else
    Put("ERROR -- new grade: ");
    Put(New_Grade);
    Put(" is out of range");
    New_Line;
  end if;
end; -- fim do bloco de faixa de índices
end loop;
-- Produzir saída
Put("Limits Frequency");
New_Line; New_Line;
for Index in 0..9 loop
  Limit_1 := 10 * Index;
  Limit_2 := Limit_1 + 9;
  if Index = 9 then
    Limit_2 := 100;
  end if;
  Put(Limit_1);
  Put(Limit_2);
  Put(Freq(Index + 1));
  New_Line;
end loop; -- for Index in 0..9 ...
end Grade_Distribution;

```

Note que o código para tratar notas de entrada inválidas está em seu próprio bloco local, permitindo que o programa continue após tais exceções serem tratadas, como em nosso exemplo anterior que lia valores do teclado. O tratador para entrada negativa está também em seu próprio bloco. A razão para esse bloco é restringir o escopo do tratador para `Constraint_Error` quando ela for levantada por uma entrada negativa.

14.2.6 Avaliação

Como no caso de algumas outras construções de linguagem, o projeto de Ada para tratamento de exceções representa algo como um consenso, ao menos na época de seu projeto (final dos anos 1970 e início dos 1980), sobre as ideias acerca do tema. Por algum tempo, Ada foi a única linguagem amplamente usada que incluía tratamento de exceções.

Existem diversos problemas com o tratamento de exceções de Ada. Um é o modelo de propagação, permitindo que as exceções sejam propagadas para um escopo mais externo no qual a exceção não é visível. Além disso, nem sempre é possível determinar a origem das exceções propagadas.

Outro problema é a inadequação do tratamento de exceções para tarefas. Por exemplo, uma tarefa que levanta uma exceção, mas não a trata, simplesmente morre.

Por fim, quando o suporte para programação orientada a objetos foi adicionado em Ada 95, seu tratamento de exceções não foi estendido para lidar com as novas construções. Por exemplo, quando diversos objetos de uma classe são criados e usados em um bloco e um deles propaga uma exceção, é impossível determinar qual deles a levantou.

Os problemas do tratamento de exceções em Ada são discutidos em Romanovsky e Sandén (2001).

14.3 TRATAMENTO DE EXCEÇÕES EM C++

O tratamento de exceções de C++ foi aceito pelo comitê de padronização de C++ em 1990 e encontrou seu caminho nas implementações C++. O projeto é, em parte, baseado no tratamento de exceções de CLU, Ada e ML. Uma grande diferença entre o tratamento de exceções de C++ e o de Ada é a ausência de exceções pré-definidas (além daquelas em sua biblioteca padrão). Logo, em C++, as exceções são definidas pelo usuário ou em bibliotecas e explicitamente levantadas.

14.3.1 Tratadores de exceção

Na seção 14.2, vimos que Ada usa unidades de programa ou blocos para especificar o escopo para tratadores de exceções. C++ usa uma construção especial introduzida com a palavra `try` para esse propósito. 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

```
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 e pode ter apenas um parâmetro formal, similar a um parâmetro formal em uma definição de função em C++, incluindo a possibilidade de 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 precisarem 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++.

14.3.2 Vinculando exceções a tratadores

Exceções em C++ são levantadas apenas por meio da sentença explícita `throw`, cuja forma geral em EBNF é

`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. Quando aparecer lá, ele relança a exceção, então tratada em outro lugar. Esse efeito é exatamente o mesmo de Ada.

O tipo da expressão `throw` seleciona o tratador particular, que deve ter um parâmetro de tipo formal que “case” com ele. Nesse caso, o *casamento* significa que 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 construção `try` causa um término imediato da execução do código em tal construção. A busca por um tratador que casa com a expressão começa com os tratadores que seguem imediatamente a construção `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 exatamente, este não será usado. Logo, tratadores para exceções específicas são colocados no topo da lista, seguidos por tratadores mais genéricos. O último tratador é

geralmente um com um parâmetro formal 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 com essa cláusula, a exceção é propagada. Se a cláusula **try** estiver aninhada dentro de outra, a exceção será propagada para os tratadores associados com a cláusula mais externa. Se nenhuma das cláusulas **try** externas levar a um tratador que casa, 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 casa for encontrado no programa durante esse processo de propagação, o tratador padrão será chamado. Esse tratador é discutido em mais detalhes na Seção 14.3.4.

14.3.3 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, que faz a exceção ser propagada.

14.3.4 Outras escolhas de projeto

Em termos das 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, não esperado, 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**. Exceções não podem ser desabilitadas.

Uma função C++ pode listar os tipos das exceções (os das expressões **throw**) que pode levantar, anexando a palavra reservada **throw** ao cabeçalho da função, seguida de uma lista desses tipos entre parênteses. 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 especificar aos usuários da função que exceções podem ser levantadas por ela. A cláusula **throw** 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 não levanta

algumas exceções não listadas, o programa será terminado. 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 e isso significa que a função não levantará qualquer exceção. Se não existir uma especificação `throw` no cabeçalho, a função pode levantar qualquer exceção. A lista não é 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 do que a sobrescrita.

Apesar de C++ não ter exceções pré-definidas, 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.

14.3.5 Um exemplo

O seguinte exemplo tem a mesma intenção e uso de tratamento de exceções que o programa Ada mostrado na Seção 14.2.5. Ele produz uma distribuição de notas de entrada usando um vetor de contadores para 10 categorias. Notas ilegais 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 de 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]++;
    } /* end of inner try compound
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 of 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 sobre-carga do operador de indexação, que pode então levantar a exceção, em vez da detecção direta do operador de indexação com a construção de seleção usada em nosso exemplo.

14.3.6 Avaliação

De algumas maneiras, o tratamento de exceções de C++ é similar ao de Ada. Por exemplo, exceções não tratadas em funções são propagadas para o chamador da função. Mas, de outras maneiras, o projeto de C++ é um tanto diferente: não existem exceções detectáveis por hardware pré-definidas que possam ser tratadas pelo usuário, e as exceções não são nomeadas. Exceções são conectadas aos tratadores por um tipo de parâmetro no qual o parâme-

tro formal pode ser omitido. O tipo do parâmetro formal de um tratador determina as condições nas quais ele é chamado, mas pode não ter nada a ver com a natureza da exceção levantada. Logo, o uso de tipos pré-definidos de exceções certamente não promove a legibilidade. É muito melhor definir classes para exceções com nomes significativos em uma hierarquia significativa que pode ser usada para definir exceções. O parâmetro de exceção fornece uma maneira de passar informações acerca de uma exceção para o tratador da exceção.

14.4 TRATAMENTO DE EXCEÇÕES EM JAVA

No Capítulo 13, o programa de exemplo Java incluía o uso de tratamento de exceções com pouca explicação. Esta seção descreve os detalhes das capacidades de Java em termos de tratamento de exceções.

O tratamento de exceções de Java 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 pré-definidas que podem ser implicitamente levantadas pela Máquina Virtual Java (JVM).

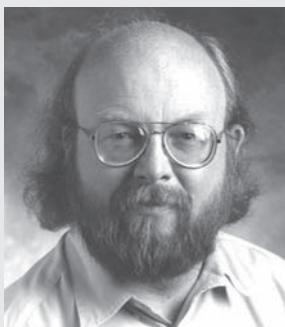
14.4.1 Classes de exceções

Todas as exceções em Java são objetos de classes descendentes da classe `Throwable`. O sistema Java inclui duas classes de exceção pré-definidas que são subclasses de `Throwable`: `Error` e `Exception`. A classe `Error` e suas descendentes estão relacionadas a erros lançados pela JVM, 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 sistema de `Exception`: `RuntimeException` e `IOException`. Como seu nome indica, `IOException` é lançada quando ocorre um erro em uma operação de entrada ou de saída, todas as quais são definidas como métodos nas várias classes definidas no pacote `java.io`.

Existem classes pré-definidas descendentes de `RuntimeException`. Na maioria dos casos, uma exceção do tipo `RuntimeException` é lançada (pela JVM⁴) 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`.

⁴ A especificação Java também requer que os compiladores JIT detectem essas exceções e lancem `RuntimeException` quando elas ocorrerem.



O nascimento de Java

JAMES GOSLING

James Gosling, *fellow* e vice-presidente na Sun Microsystems, é o criador da linguagem Java e um dos programadores mais reconhecidos da indústria da computação. Ele ganhou o Prêmio de Excelência em Programação da Software Development, desenvolveu o NeWS – sistema de janelas extensível a redes da Sun – e foi um dos responsáveis pelo projeto Andrew na Universidade de Carnegie-Mellon, na qual obteve Doutorado (PhD) em Ciência da Computação.

HISTÓRICO PESSOAL

Como você se envolveu com a computação? Quando eu tinha 14 anos, um colega de meu pai me levou às instalações de computação da Universidade de Calgary. Fui fisgado. Foi simplesmente amor à primeira vista. Aprendi a programar sozinho, arranjei um emprego no departamento de física quando estava no ensino médio e foi uma bola de neve a partir disso.

Qual foi o seu primeiro emprego em computação? Foi escrevendo software de terra para o satélite ISIS-II para o Departamento de Física da Universidade de Calgary.

Qual foi o seu emprego favorito? Meu primeiro emprego foi muito especial. Mas também gosto muito do meu atual: sou pesquisador dos laboratórios da Sun. Oficialmente, sou "VP e fellow".

JAVA: O NASCIMENTO DE UMA LINGUAGEM

Muitas linguagens nasceram como um produto secundário oriundo de um objetivo diferente. Ouvi que esse foi o caso de Java e do Projeto Green. Você pode nos contar a história? Estábamos investigando tendências futuras que poderiam ter impacto na Sun. Rapidamente, focamos na disseminação dos sistemas digitais em dispositivos que normalmente não eram vistos como computadores (telefones celulares, televisores, sistemas de controle etc.), com o crescimento das redes de computadores. Iniciamos com a construção de um protótipo que nos ajudou a aprender sobre o espaço. Foram encontrados problemas oriundos das ferramentas de programação básicas que estávamos usando. Minha parte no projeto foi resolver o problema de ferramentas. O resultado foi Java.

Por que criar uma linguagem? Nada mais funcionalava. O primeiro protótipo foi feito em C++, mas antes pensamos e descartamos muitos outros. Eles eram (todos) muito ligados a arquiteturas de CPU específicas (no nível binário), com pouca preocupação com a segurança. Além disso, eles tinham alguns problemas de confiabilidade.

Você considera SIMULA o antecessor de Java. Por que SIMULA? Ela foi a linguagem OO original. Eu já usei muito a SIMULA. Como vantagem em relação a C e C++, ela tinha herança simples e um modo de memória restrito.

Que recursos distinguiram Java de outras linguagens populares? Produtividade, confiabilidade, segurança, portabilidade.

O quanto esses recursos e as tendências atuais de desenvolvimento de hardware e de software ajudaram no sucesso imediato de Java? Muito.

JAVA: O JOGO DOS NOMES

Por que o projeto original se chamava Green? Por nenhuma razão específica. A Apple tinha um projeto de pesquisa chamado "Pink", – as cores estavam na moda. A inspiração para o nosso veio da porta do conjunto de escritórios em que trabalhávamos: ela era verde. Java era originalmente chamada "Oak" (carvalho). Eu tinha de dar um nome a, ao olhar pela janela do escritório, vi um carvalho. Entretanto, esse nome original possuía todos os tipos de conflitos de marcas registradas. Os advogados pediram que encontrássemos um nome livre dessas questões.

Você está cansado do nome Java? Não.

Se você pudesse renomear Java que nome escolheria? Essa é uma pergunta muito difícil. Escolher nomes é muito complicado.

JAVA: PASSADO, PRESENTE, FUTURO

Se você pudesse modificar dois recursos de Java, ou retrabalhá-los, quais seriam? Objetos leves e sentenças de seleção múltipla (*switch*).

Você pensa a respeito disso? Sim.

Os dispositivos que você tentou criar para o Projeto Green existem hoje? Você estava mirando o alvo certo? Sim. Os Palm Pilots e os celulares modernos repletos de recursos estão diretamente alinhados com que estávamos tentando fazer.

Se você fosse designado para um projeto cujo objetivo fosse similar ao do Projeto Green, em que funcionalidade para o usuário você focaria seus

“Java era originalmente chamada “Oak” (carvalho). Eu tinha de dar um nome e, ao olhar pela janela do escritório, vi um carvalho.”

esforços? Ou melhor, se a Internet e Java eram as ferramentas “uau” do início dos anos 1990, qual será a próxima ferramenta “uau”? A próxima ferramenta “uau” não mudou – ainda é a Internet. Mal começamos a explorá-la.

Avance no tempo 15 anos: que funcionalidade as linguagens de programação poderiam oferecer que ainda não oferecem? Racionalização/verificação.

Se você não estivesse fazendo isso, o que estaria fazendo? Algo que envolvesse construir coisas.

14.4.2 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 pré-definida `Throwable`.

A sintaxe da construção `try` em Java é exatamente igual à de C++, exceto pela cláusula `finally` descrita na Seção 14.4.6.

14.4.3 Vinculando exceções a tratadores

Lançar uma exceção é bastante 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

```
throw new MyException();
```

A criação da instância da exceção para `throw` pode ser feita separadamente da sentença `throw`, como em

```
MyException myExceptionObject = new MyException();  
...  
throw myExceptionObject;
```

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. Então 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 é executado.

Exceções podem ser tratadas e depois relançadas ao incluir uma sentença `throw` sem um operando no final do tratador. A nova exceção lançada não será

tratada no mesmo `try` em que ela foi originalmente lançada, assim, repetições infinitas não são uma preocupação. Esse relançamento é normalmente 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 pode também lançar alguma outra exceção, que não aquela que transferiu o controle para esse tratador.

Para garantir que exceções capazes de ser 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` simplesmente definindo o tratador com um parâmetro de tipo `Exception`, como em

```
catch (Exception genericObject) {
    ...
}
```

Como um nome de classe sempre casa consigo ou com uma classe ancestral, qualquer classe derivada de `Exception` casa com `Exception`. 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 ela termina quando um casamento é encontrado.

14.4.4 Outras escolhas de projeto

Durante a execução de programas, 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 com o objeto parâmetro, criada 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 a aparência e posicionamento (em um programa) similar à da especificação `throw` de C++. Entretanto, a semântica de `throws` é de certa forma 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 ele pode lançar `IOException`, significa que ele pode lançar um objeto de `IOException` ou um objeto de qualquer uma de suas classes descendentes, como `EOFException`, e que ele 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, o compilador garante que todas as exceções verificadas que um método pode lançar sejam ou listadas em sua cláusula **throws** ou tratadas no método. Note que verificar isso em tempo de compilação está em contraste com C++, em que isso é feito em tempo de execução. A razão pela qual as exceções das classes `Error` e `RuntimeException` e suas descendentes não são verificadas é que qualquer método pode lançá-las. Um programa pode capturar exceções não verificadas, mas ele 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 fazê-lo quaisquer métodos que o sobrescrevam. Um método pode lançar qualquer exceção listada em sua cláusula **throws**, com qualquer uma de suas classes descendentes.

Um método que não lança uma exceção em particular 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 exceção 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 **throws** superior (que envolve a atual), se existir uma, ou para o chamador do método, se não existir uma cláusula **throws** superior.

Não existem tratadores de exceção padronizados, e não é possível desabilitar exceções. Continuações em Java ocorrem exatamente como em C++.

14.4.5 Um exemplo

A seguir, temos o programa Java com as capacidades do programa C++ da Seção 14.3.5:

```
// 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.*;
// The exception definition to deal with the end of data
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 do try-catch interno
            catch(ArrayIndexOutOfBoundsException e) {
                if (newGrade == 100)
                    freq [9]++;
                else
                    System.out.println("Error - new grade: " +
                        newGrade + " is out of range");
            } //** fim do catch (ArrayIndex...
        } //** end of 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]);
        }
    }
}
```

```
        limit_2 + "      " + freq [index]) ;
    } /** fim do for (index = 0; ...
} /** fim of 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 pela JVM. 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 frequentemente não são úteis.

14.4.6 A Cláusula `finally`

Existem algumas situações nas quais um processo deve ser executado independentemente de uma cláusula `try` lançar uma exceção ou se uma exceção lançada é capturada em um método. Um exemplo de tal situação é um arquivo que deve ser fechado. Outra é se o método tem algum recurso externo que deve ser liberado no método independentemente de como a execução do método terminar. A cláusula `finally` foi projetada para esses tipos de necessidades. Ela é colocada no final da lista de tratadores logo 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: se a cláusula `try` não lançar uma exceção, a cláusula `finally` será executada antes de a execução continuar após a construção `try`. Se a cláusula `try` lançar uma exceção e ela for capturada por um tratador que a segue, a cláusula `finally` é 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` é 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 nes-

ses casos é o mesmo de quando ela é usada com tratamento de exceções. Por exemplo, considere:

```
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** aqui será executada, independentemente de o **return** terminar o laço ou ele terminar normalmente.

14.4.7 Aserções

Na discussão de Plankalkül no Capítulo 2, mencionamos que ela incluía asserções. Aserções foram adicionadas à Java na versão 1.4. Para usá-las, é necessário habilitá-las rodando o programa com a opção `enableassertions` (ou `ea`), como em

```
java -enableassertions MyProgram
```

As duas formas possíveis da sentença **assert** são:

```
assert condição;
assert condição : expressão;
```

No primeiro caso, a condição é testada quando a execução alcançar o **assert**. Se for avaliada como verdadeira, nada acontece. Se avaliada como falsa, a exceção `AssertionError` é 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 a computação do programa esteja pronta para produzir resultados corretos. Muitos programadores colocam tais verificações quando escrevem um programa, como uma ajuda para a depuração, mesmo que a linguagem usada não suporte asserções. Quando o programa é suficientemente testado, essas verificações são removidas. A vantagem das sentenças **assert**, que têm o mesmo propósito, é poderem ser desabilitadas sem removê-las do programa. Isso economiza o esforço de removê-las do programa e permite o seu uso durante subsequentes manutenções de programa.

14.4.8 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 a partir de todos os outros objetos (e não objetos) que habitam um programa. Que significância pode ser anexada 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 ele não trate. Logo, o leitor de um método Java sabe pelo seu cabeçalho quais exceções pode lançar, mas que ele não trata. Um compilador C++ ignora cláusulas `throw`, mas um compilador Java garante a listagem de todas as exceções que um método possa lançar em sua cláusula `throws`.

Terceiro, a adição da cláusula `finally` é uma grande conveniência em certas situações. Ela permite que certas ações de limpeza ocorram independentemente de como uma sentença composta termine.

Por fim, a JVM implicitamente lança uma variedade de exceções pré-definidas, como para índices de matrizes fora de faixa e o desreferenciamento de referências nulas, que podem ser tratadas por qualquer programa de usuário. Um programa C++ pode tratar apenas as exceções que ele lança explicitamente (ou que sejam lançadas por classes de biblioteca).

Relativamente ao tratamento de exceções de Ada, as facilidades de Java são de um modo geral comparáveis. A presença de uma cláusula `throws` em um método Java é uma boa ajuda para a legibilidade, enquanto Ada não tem um recurso correspondente. Java é certamente mais próxima de Ada do que de C++ em uma área – aquela de permitir que os programas tratem exceções detectadas pelo sistema.

C# inclui construções de tratamento de exceções muito parecidas com as de Java, exceto que C# não tem uma cláusula `throws`.

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 criadas explicitamente pelo código do usuário ou implicitamente por hardware ou por um interpretador de software, os eventos são criados por ações exter-

nas, como interações de usuário por uma interface gráfica de usuário (GUI). Nesta seção, são introduzidos os fundamentos do tratamento de eventos, menos complexos do que os do tratamento de exceções.

Em programação convencional (não dirigida por eventos), o código 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 completamente imprevisíveis, geralmente disparadas por interações de usuários com o programa em execução.

O tipo particular 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 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. Estritamente falando, 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 está sendo discutido aqui.

Um **tratador de evento** é um segmento de código executado em resposta à aparição de um evento. Tratadores de evento habilitam um programa a responder às ações do usuário.

Apesar de a programação dirigida por eventos estar sendo usada 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 um exemplo, considere as GUIs apresentadas aos usuários de navegadores Web. Muitos documentos Web apresentados aos usuários são agora dinâmicos. Tal documento pode apresentar 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 para verificar erros simples e omissões nos elementos de um formulário, seja quando são mudados ou quando o formulário é submetido para o servidor Web, de forma a 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 sua corretude então deve ser verificada por um programa ou *script* residente no servidor antes de serem processados. Esse tipo de programação dirigida por eventos é feita usando uma linguagem de *scripting* do lado cliente, como JavaScript.

14.6 TRATAMENTO DE EVENTOS COM JAVA

Java suporta duas abordagens para apresentar visualizações interativas aos usuários: por meio de programas aplicativos ou de *applets*. Ambos usam as mesmas classes para definir os componentes GUI e os tratadores de eventos que fornecem a interatividade. Apesar de discutirmos apenas *applets*, esta seção também se aplica aos programas aplicativos.

A versão inicial de Java fornecia um suporte de certa forma primitivo para componentes GUI. Na versão 1.2 da linguagem, uma nova coleção de componentes foi adicionada. Esses componentes são coletivamente chamados de Swing.

14.6.1 Componentes de GUI do Java Swing

O pacote Swing, definido em `javax.swing` inclui uma coleção de componentes GUI. Como nosso interesse aqui é no tratamento de eventos, não nos componentes 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 um 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 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 implicitamente passa a ser não pressionado. O construtor de `JRadioButton`, usado para criar os botões de rádio, recebe dois parâmetros: o rótulo e o estado inicial do botão de rádio (`true` ou `false`, para pressionado e não pressionado, respectivamente). Após os botões de rádio serem criados, eles são colocados em seu grupo de botões com o método `add` do objeto de grupo. Considere o seguinte exemplo:

```
ButtonGroup payment = new ButtonGroup();
JRadioButton box1 = new JRadioButton("Visa", true);
JRadioButton box2 = new JRadioButton("Master Charge",
                                    false);
JRadioButton box3 = new JRadioButton("Discover", false);
payment.add(box1);
payment.add(box2);
payment.add(box3);
```

Uma visualização de um *applet* é, na verdade, um *frame*, ou seja, uma estrutura com múltiplas camadas. Estamos interessados em apenas uma dessas camadas, o painel de conteúdo, onde os *applets* colocam suas saídas. Os programas do usuário não colocam nada diretamente no painel de conteúdo; em vez disso, colocam objetos gráficos em um painel e então o adicionam ao painel de conteúdo. Para aplicações, um *frame* é criado e o painel construído é adicionado ao painel de conteúdo desse *frame*.

Um painel de conteúdo é criado como um objeto *Container*, usando o método `getContentPane`, como em

```
Container contentPane = getContentPane();
```

Objetos gráficos pré-definidos, como componentes GUI, podem ser colocados diretamente em um painel criado no *applet* então adicionado ao painel de conteúdo deste. O código a seguir cria o objeto painel que usaremos na discussão a seguir sobre componentes:

```
JPanel myPanel = new JPanel();
```

Após os componentes terem sido criados com construtores, eles devem ser colocados no painel com o método `add`, como em

```
myPanel.add(button1);
```

14.6.2 O modelo de eventos de Java

Interações de usuários com componentes GUI criam eventos capazes de ser capturados por tratadores de eventos, os quais fornecem as computações associadas. Componentes GUI são considerados geradores de eventos; todos eles geram eventos. Em Java, tratadores de eventos são chamados de **escutadores de eventos**, que são conectados aos geradores de eventos pelo registro de um escutador de evento. O registro de um escutador como esse é feito com um método da classe que implementa a interface escutadora, como descrito posteriormente. O objeto painel no qual os componentes são colocados pode ser o escutador de eventos para esses componentes. Apenas escutadores de eventos registrados para um evento especificado são notificados quando tal evento ocorre.

Um gerador de eventos informa a um escutador de um evento pelo envio de uma mensagem para o escutador (em outras palavras, chamando um dos métodos do escutador). O método escutador que recebe a mensagem implementa um tratador de evento. Para fazer os métodos estarem em conformidade com um protocolo padrão, é usada uma interface que prescreve protocolos de métodos padronizados, mas não fornece implementações desses métodos. Esse protocolo poderia ser especificado forçando o gerador de eventos a ser uma subclasse de uma classe da qual ele herdaria o protocolo. No entanto, a classe `JApplet` já tem uma superclasse, e em Java uma classe

pode ter apenas uma classe pai. Logo, o protocolo deve vir de uma interface. Uma classe não pode ser instanciada a menos que ela forneça definições para todos os métodos nas interfaces que ela implementa.

Uma classe que precisa implementar um escutador deve implementar uma interface para esse escutador. Existem muitas classes de eventos e interfaces escutadoras. Uma classe de eventos é `ItemEvent`, associada com o evento de selecionar uma caixa de verificação, um botão de rádio ou um item de lista. A interface `ItemListener` prescreve um método, `itemStateChanged`, o tratador para eventos `ItemEvent`. Logo, para fornecer uma ação disparada pela ação 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 de 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 criado em um painel em um *applet* poderia ser implementado no painel. Então, para um botão de rádio chamado `button1` em um painel chamado `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, que fornece informações acerca deste. Classes de evento têm métodos para acessar tais informações. Por exemplo, quando chamado por um botão de rádio, `isSelected` retorna verdadeiro ou falso, 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 tais classes normalmente são importadas por qualquer classe de *applet* que use eventos.

O *applet* de exemplo a seguir, `RadioB`, ilustra o uso de eventos e de seu tratamento para mostrar conteúdo dinâmico em um *applet*. Esse *applet* controla botões de rádio que controlam o estilo da fonte do conteúdo de uma caixa de texto. Ele cria um objeto `Font` para cada um dos quatro estilos de fonte. Cada um deles tem um botão de rádio para habilitar que o usuário selecione o estilo. O *applet* cria uma cadeia de texto cujo estilo da fonte será controlado pelo usuário por meio dos botões de rádio. Ele configura, portanto, o estilo da fonte da cadeia de texto de acordo com a seleção do usuário.

```
/* RadioB.java
Um applet para ilustrar o tratamento de eventos com
botões de rádio Interativos que controlam o estilo da
fonte de um textfield
```

```
/*
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;

public class RadioB extends JApplet implements
    ItemListener {

    // Torna a maioria das variáveis como variáveis de classe,
    // porque
    // tanto a inicialização quanto o tratador de eventos devem
    // vê-las

    private Container contentPane = getContentPane();
    private JTextField text;
    private Font plainFont, boldFont, italicFont,
        boldItalicFont;
    private JRadioButton plain, bold, italic, boldItalic;
    private ButtonGroup radioButtons = new ButtonGroup();
    private JPanel myPanel = new JPanel();

    // O método init é onde o documento é inicialmente construído
    // built

    public void init() {

        // Configura a cor de fundo do painel

        myPanel.setBackground(Color.cyan);

        // Cria as fontes

        plainFont = new Font("Serif", Font.PLAIN, 16);
        boldFont = new Font("Serif", Font.BOLD, 16);
        italicFont = new Font("Serif", Font.ITALIC, 16);
        boldItalicFont = new Font("Serif", Font.BOLD +
            Font.ITALIC, 16);

        // Cria a cadeia de texto de teste, configura sua fonte,
        // e a adiciona ao painel

        text = new JTextField(
            "In what font style should I appear?", 30);
        myPanel.add(text);
        text.setFont(plainFont);

        // Cria os botões de rádio para as fontes e adiciona-os
        // ao painel
```

```
plain = new JRadioButton("Plain", true);
bold = new JRadioButton("Bold");
italic = new JRadioButton("Italic");
boldItalic = new JRadioButton("Bold Italic");
radioButtons.add(plain);
radioButtons.add(bold);
radioButtons.add(italic);
radioButtons.add(boldItalic);

// Registra os tratadores de evento em myPanel

plain.addItemListener(this);
bold.addItemListener(this);
italic.addItemListener(this);
boldItalic.addItemListener(this);

// Adiciona os botões ao painel

myPanel.add(plain);
myPanel.add(bold);
myPanel.add(italic);
myPanel.add(boldItalic);

// Adiciona o painel ao painel de conteúdo para o applet

contentPane.add(myPanel);

} // End of init()

// 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

} // Fim do applet RadioB
```

O applet RadioB produz a tela mostrada na Figura 14.2.



Figura 14.2 Saída do *applet* RadioB.

RESUMO

A maioria das linguagens de programação usadas inclui agora o tratamento de exceções.

Ada fornece extensivos recursos de tratamento de exceções e uma coleção pequena, mas completa, de exceções pré-definidas. Os tratadores são anexados às entidades de programas, apesar de as exceções poderem ser implícita ou explicitamente propagadas a outras entidades de programas se nenhum tratador local estiver disponível.

C++ não inclui exceções pré-definidas (exceto aquelas definidas na biblioteca padrão). Exceções C++ são objetos de um tipo primitivo, de uma classe pré-definida ou de uma 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 o método pode lançar.

As exceções Java são objetos cujos ancestrais devem ser rastreados para uma classe que descendia 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 levantar 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.

Um evento é uma notificação de que algo que precisa de tratamento ocorreu. Os eventos são criados por meio de interações dos usuários com um programa, normalmente por uma interface gráfica. Os tratadores de evento em Java são chamados de escutadores de eventos. Um escutador de eventos deve ser registrado para um evento se ele deve ser notificado de sua ocorrência. Dois dos escutadores de eventos mais usados são `actionPerformed` e `itemStateChanged`, cujos protocolos são fornecidos por interfaces associadas.

NOTAS BIBLIOGRÁFICAS

Um dos artigos mais importantes sobre tratamento de exceções que não está conectado com uma linguagem de programação em particular é o trabalho de Goodenough (1975). Os problemas com o projeto de PL/I para tratamento de exceções são cobertos em MacLaren (1977). O projeto de tratamento de exceções de CLU é descrito por

Liskov e Snyder (1979). Recursos de tratamento de exceções da linguagem Ada são descritos em ARM (1995) e criticamente avaliados por Romanovsky e Sandén (2001). O tratamento de exceções em C++ é descrito por Stroustrup (1997) e em Java, por Campione et al. (2001).

QUESTÕES DE REVISÃO

1. Defina exceção, tratamento de exceção, levantar uma exceção, desabilitar uma exceção, continuação, finalização e exceção pré-definida.
2. Quais são as duas alternativas para projetar continuações?
3. Quais são as vantagens de ter suporte pré-definido para tratamento de exceções em uma linguagem?
4. Quais são as questões de projeto para tratamento de exceções?
5. O que significa para uma exceção ser vinculada a um tratador de exceção?
6. Quais são os possíveis *frames* para exceções em Ada?
7. Para onde são propagadas as exceções não tratadas em Ada se foram lançadas em um subprograma? Para um bloco? Para um corpo de pacote? Para uma tarefa?
8. Onde a execução continua após uma exceção ser tratada em Ada?
9. Como uma exceção pode ser explicitamente levantada em Ada?
10. Quais são as quatro exceções definidas no pacote Standard de Ada?
11. Como é definida uma exceção definida pelo usuário em Ada?
12. Como uma exceção pode ser suprimida em Ada?
13. Descreva três problemas com o tratamento de exceções de Ada.
14. Qual é o nome de todos os tratadores de exceção em C++?
15. Como as exceções podem ser explicitamente levantadas em C++?
16. Como as exceções são vinculadas aos tratadores em C++?
17. Como um tratador de exceção pode ser escrito em C++ de forma que ele trate qualquer exceção?
18. Onde vai o controle de execução quando um tratador de exceção C++ completa sua execução?
19. C++ inclui exceções pré-definidas?
20. Por que o levantamento de uma exceção em C++ não é chamado de *raise*?
21. Qual é a classe raiz de todas as classes de exceção em Java?
22. Qual é a classe pai da maioria das classes de exceção definidas pelo usuário em Java?
23. Como um tratador de exceção pode ser escrito em Java de forma que ele trate qualquer exceção?
24. Quais são as diferenças entre uma especificação **throw** em C++ e uma cláusula **throws** em Java?
25. Qual é a diferença entre exceções verificadas e não verificadas em Java?
26. Você pode desabilitar uma exceção Java?
27. Qual é o propósito da cláusula **finally** em Java?
28. Que vantagem as asserções definidas pela linguagem possuem sobre construções de seleção simples (**if-write**)?

29. De que maneiras o tratamento de exceções e o tratamento de eventos são relacionados?
30. O que é um escutador de evento?

CONJUNTO DE PROBLEMAS

1. O que os projetistas de C obtiveram em retorno ao não requererem a verificação de faixas de índices?
2. Descreva três abordagens para o tratamento de exceções em linguagens que não fornecem suporte direto para tal.
3. A partir de livros texto das linguagens de programação PL/I e Ada, busque os respectivos conjuntos de exceções pré-definidas. Faça uma avaliação comparativa das duas, considerando tanto a completude quanto a flexibilidade.
4. A partir de ARM (1995), determine como exceções que ocorrem durante *rendezvous* são tratadas.
5. A partir de um livro texto sobre COBOL, determine como o tratamento de exceções é feito em programas COBOL.
6. Em linguagens sem recursos para tratamento de exceções, é comum fazer a maioria dos subprogramas incluir um parâmetro de “erro”, que pode ser configurado para algum valor representando “OK” ou outro representando “erro no procedimento”. Que vantagem um recurso de tratamento de exceções linguístico como o de Ada tem em relação a esse método?
7. Em uma linguagem sem recursos de tratamento de exceções, poderíamos enviar um procedimento de tratamento de erros como um parâmetro para cada procedimento capaz de detectar erros a serem tratados. Quais são as desvantagens existentes nesse método?
8. Compare os métodos sugeridos nos Problemas 6 e 7. Qual deles você acha melhor. Por quê?
9. Escreva uma análise comparativa da cláusula **throw** de C++ e a cláusula **throws** de Java.
10. Compare os recursos para tratamento de exceções de C++ com os de Ada. Qual projeto, em sua opinião, é o mais flexível. Qual torna possível a escrita de programas mais confiáveis?
11. 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) { ... }
    }
}

```

Em cada uma das quatro sentenças `throw`, onde a exceção é tratada? Note que `fun1` é chamada a partir de `fun2` na classe `Small`.

12. Escreva um comparativo detalhado das capacidades de tratamento de exceções de C++ com as de Java.
13. Com a ajuda de um livro sobre ML, escreva uma comparação detalhada das capacidades de tratamento de exceções de ML com as de Java.
14. Resuma os argumentos a favor dos modelos de continuação: terminação e reinício.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Escreva um segmento de código em Ada que tente novamente uma chamada a um procedimento, `Tape_Read`, que lê dados de entrada de um *drive* de fita e pode lançar a exceção `Tape_Read_Error`.
2. Suponha que você esteja escrevendo uma função C++ com três abordagens alternativas para implementar seus requisitos. Escreva uma versão esqueleto dessa função de forma que se a primeira alternativa levantar qualquer exceção, a segunda é tentada, e se a segunda alternativa levantar qualquer exceção, a terceira é executada. Escreva o código como se existissem três métodos onde os procedimentos são chamados de `alt1`, `alt2` e `alt3`.

3. Escreva um programa em Java que leia uma lista de valores inteiros na faixa de -100 a 100 do teclado e calcule a soma dos quadrados dos valores de entrada. Esse programa deve usar tratamento de exceções para garantir que os valores de entrada estão na faixa e são valores inteiros legais, 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 use-o para mostrar a saída do resultado. No caso de transbordamento da soma, uma mensagem de erro deve ser impressa e o programa deve ser terminado.
4. Escreva um programa em C++ para a especificação do Exercício 3.
5. Escreva um programa em Ada para a especificação do Exercício 3.
6. Revise o programa Java da Seção 14.4.5 para usar `EOFException` para detectar o final da entrada.
7. Reescreva o código Java da Seção 14.4.5 para usar uma cláusula `finally` em C++.

Capítulo 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 Aplicações de linguagens funcionais

15.10 Uma comparação entre linguagens funcionais e imperativas

Este capítulo introduz a programação funcional e algumas das linguagens de programação que foram projetadas para essa abordagem de desenvolvimento de software. Começamos revisando as ideias fundamentais das funções matemáticas, já que as linguagens funcionais são baseadas nelas. Depois, a ideia de uma linguagem de programação funcional é introduzida, seguida de uma visão da primeira linguagem funcional, LISP, e suas estruturas de dados e sintaxe funcional, baseada na notação lambda. 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 em Scheme. A seguir, fornecemos breves introduções a COMMON LISP, ML e Haskell. Uma seção posterior descreve algumas das aplicações das linguagens de programação funcional. Por fim, apresentamos um breve comparativo entre linguagens funcionais e imperativas.

15.1 INTRODUÇÃO

Os primeiros 14 capítulos deste livro se preocupam primariamente com as linguagens imperativas e orientadas a objetos. Com exceção de Smalltalk, as linguagens orientadas a objetos discutidas têm formas similares às 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 vistas coletivamente como uma progressão de desenvolvimentos para melhorar o modelo básico, que era 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 nos possíveis processos de desenvolvimento de software.

Outras bases para o projeto de linguagens existem, algumas orientadas mais para paradigmas de programação em particular ou metodologias para execução eficiente em uma arquitetura de computadores em particular. Até agora, entretanto, apenas uma minoria de programas tem sido escrita em linguagens não imperativas.

O paradigma de programação funcional, baseado em funções matemáticas, é a base de projeto para um dos estilos de linguagem não imperativos mais importantes. Esse estilo de programação é suportado por linguagens de programação funcional (ou linguagens aplicativas).

LISP começou como uma linguagem puramente funcional, mas rapidamente adquiriu alguns recursos imperativos importantes que aumentaram sua eficiência de execução. Ela ainda é a mais importante das linguagens fun-

cionais, ao menos no sentido de que foi a única a atingir uso disseminado. Scheme é um pequeno dialeto de LISP, de escopo estático. COMMON LISP é um amálgama de diversos outros dialetos de LISP do início dos anos 1980. ML e Haskell são linguagens funcionais fortemente tipadas com uma sintaxe mais convencional do que LISP e Scheme.

O Prêmio Turing da ACM de 1977 foi dado a John Backus por seu trabalho no desenvolvimento do Fortran. Cada agraciado apresenta uma palestra quando o prêmio é formalmente dado, e a palestra é publicada na Communications of the ACM. Em sua palestra do Prêmio Turing, Backus (1978) argumentou que linguagens de programação puramente funcionais são melhores do que linguagens imperativas porque resultam em programas mais legíveis, mais confiáveis e mais propensos a serem corretos. O cerne de seu argumento era que os programas puramente funcionais eram mais fáceis de serem entendidos, tanto durante quanto após o desenvolvimento, em grande parte porque os significados das expressões são independentes de seu contexto (um recurso característico de uma linguagem de programação funcional pura é que nem expressões, nem funções, têm efeitos colaterais).

Em sua palestra, Backus propôs uma linguagem funcional pura, FP (*Functional Programming*), que usou como base para sua argumentação. Apesar de a linguagem não ter sido bem-sucedida, ao menos em termos de atingir uso disseminado, sua ideia 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 o conceito de que as linguagens de programação funcional são superiores às linguagens imperativas tradicionais, apesar desses esforços terem fracassado em seus objetivos.

Um objetivo deste capítulo é fornecer uma introdução à programação funcional usando o básico de Scheme, intencionalmente deixando de fora seus recursos imperativos. Material suficiente sobre Scheme é incluído para permitir que o leitor escreva alguns programas simples, mas interessantes. É difícil adquirir um sentimento real acerca da programação funcional sem alguma experiência de programação real, então isso é fortemente encorajado.

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. A definição de função especifica os conjuntos domínio e imagem, explícita ou implicitamente, com o mapeamento, descrito por uma expressão ou, em alguns casos, por uma tabela. As funções são geralmente aplicadas a um elemento em particular do conjunto domínio, fornecido como um parâmetro para a função. Note que o conjunto domínio pode ser o produto vetorial de diversos conjuntos (refletindo que pode existir mais de um parâmetro). Uma função leva a, ou retorna, 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 expressões condicionais, e não por sequência e repetição iterativa, comuns nas linguagens de programação imperativas.

Outra característica importante das funções matemáticas é que, como elas não têm efeitos colaterais, sempre definem o mesmo valor quando fornecido o mesmo conjunto de argumentos¹. Efeitos colaterais em linguagens de programação estão conectados a variáveis que modelam posições de memória.

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. Na matemática, não existe o conceito do estado de uma função.

Uma função matemática define um valor, em vez de especificar uma sequência de operações sobre valores em memória para produzir um valor. Não existem variáveis no sentido das variáveis imperativas, então não podem existir efeitos colaterais.

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,

$$\text{cube}(x) \equiv x * x * x, \text{ onde } x \text{ é um número real}$$

Nessa definição, os conjuntos domínio e imagem são os números reais. O símbolo \equiv 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. É assim que 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. Por exemplo, $\text{cube}(2.0)$ leva ao valor 8.0. Mais uma vez, é importante notar que, durante a avaliação, o mapeamento de uma função não contém nenhum parâmetro desvinculado, onde um parâmetro vinculado é um nome para um valor em particular. Cada ocorrência de um parâmetro é vinculada a um valor do conjunto domínio e é considerada uma constante 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 ex-

¹ Note que funções matemáticas *definem* valores, enquanto funções de linguagens de programação *produzem* valores.

pressão lambda especifica os parâmetros e o mapeamento de uma função. A expressão lambda é a função propriamente dita, que é não nomeada. Por exemplo, considere

$$\lambda(x)x * x * x$$

Conforme dito anteriormente, antes da avaliação, um parâmetro representa qualquer membro do conjunto domínio, mas durante a avaliação ele é vinculado a um membro em particular. Quando uma expressão lambda é avaliada para um parâmetro, a expressão é dita aplicada a esse parâmetro. A mecânica de tal aplicação é a mesma para qualquer avaliação de função. A aplicação da expressão lambda de exemplo é denotada como no exemplo:

$$(\lambda(x)x * x * x)(2)$$

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 uma que recebe funções como parâmetros ou que leva a uma função como resultado, ou ambos. Um tipo comum de forma funcional é a **composição de funções**, 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 de funções é escrita como uma expressão, usando \circ 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

$$h(x) \equiv f(g(x)), \text{ or } h(x) \equiv (3 * x) + 2$$

Aplicar-para-todos (*apply-to-all*) é uma forma funcional que recebe uma única função como um parâmetro. Se aplicada para uma lista de argumentos, aplicar-para-todos aplica seu parâmetro funcional para cada um dos valores no argumento lista e coleta os resultados em uma lista ou em uma sequência. Aplicar-para-todos é denotada por α .

Considere

$$h(x) \equiv x * x$$

então

$\alpha(b, (2, 3, 4))$ resulta em $(4, 9, 16)$

Existem muitas 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 mimetizar funções matemáticas ao máximo possível. Esse objetivo resulta em uma abordagem para a solução de problemas fundamentalmente diferente de abordagens 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 atenção necessária às células de memória resulta em uma metodologia de programação de um 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. O compilador manipula o armazenamento dos resultados intermediários de avaliações de expressões em linguagens de alto nível. O armazenamento de resultados intermediários ainda é necessário, mas os detalhes são escondidos do programador.

Uma linguagem de programação puramente funcional não usa variáveis, nem sentenças de atribuição, liberando o programador de preocupações relacionadas às células de memória do computador no qual o programa é executado. Sem variáveis, as construções de iteração não são possíveis, já que elas são controladas por variáveis. As repetições devem ser especificadas com recursão em vez de com repetição. 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 muito mais simples do que a semântica das linguagens imperativas (e do que linguagens funcionais que incluem recursos imperativos).

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 definida, ela requer apenas um número relativamente pequeno de funções primitivas.

Apesar de as linguagens funcionais serem geralmente implementadas com interpretadores, os programas nelas podem ser compilados.

Uma das fontes da execução mais lenta de programas escritos em linguagens funcionais, em relação a programas semanticamente equivalentes em linguagens imperativas, é o uso da recursão para especificar repetição. Com recursão, a sobrecarga da ligação das chamadas e retornos de funções é parte do custo. O uso de recursão em cauda é uma solução parcial para esse problema. Brevemente, se uma chamada recursiva em uma função é a última expressão na função, ela é chamada de **recursão em cauda** e um compilador pode facilmente converter tal recursão em uma iteração, resultando em uma execução mais rápida. Além disso, é possível converter automaticamente algumas recursões que não são em cauda para recursão em cauda, expandindo a aplicabilidade do processo. A recursão em cauda é discutida com mais detalhes em termos de Scheme na Seção 15.5.11.

Linguagens imperativas normalmente fornecem apenas suporte limitado para programação funcional. A desvantagem mais séria do uso de uma linguagem imperativa para fazer programação funcional é que funções em linguagens imperativas têm restrições acerca dos tipos de valores que podem ser retornados. Em algumas linguagens, como o Fortran, apenas valores de tipos escalares podem ser retornados. Mais importante, linguagens imperativas normalmente não podem retornar uma função. Tais restrições limitam os tipos de formas funcionais que podem ser fornecidas. Outro problema sério com as funções de linguagens imperativas é a possibilidade de efeitos colaterais funcionais. Como vimos no Capítulo 7, os efeitos colaterais funcionais complicam a legibilidade e diminuem a confiabilidade de expressões.

15.4 A PRIMEIRA LINGUAGEM DE PROGRAMAÇÃO FUNCIONAL: LISP

Muitas linguagens de programação funcional têm sido desenvolvidas. A mais antiga e a mais utilizada é LISP. Estudar linguagens funcionais pela LISP é de certa forma equivalente a estudar as linguagens imperativas por Fortran: LISP foi a primeira linguagem funcional, mas apesar de ela ter evoluído continuamente por quase 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 de alguma maneira estranhas, os descendentes do LISP original representam bem os conceitos fundamentais da programação funcional e, dessa forma, merecem estudo.

15.4.1 Tipos de dados e estruturas

Existiam apenas dois tipos de objetos de dados no LISP original: átomos e listas². Eles não são tipos no sentido dos das linguagens imperativas. Na verdade, o 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. Na medida em que a linguagem se desenvolveu, entretanto, viu-se que em LISP raramente eram necessárias as operações de listas gerais de inserção e remoção.

As listas são especificadas em LISP ao delimitarmos seus elementos dentro de 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; o quarto é a sublista (E (F G)), que tem como segundo elemento a sublista (F G).

Internamente, as listas são normalmente armazenadas como estruturas de lista simplesmente encadeadas nas quais cada nó tem dois ponteiros e representa um elemento. Um nó para um átomo tem o primeiro ponteiro apontando para alguma representação do átomo, como seu símbolo ou valor numérico. 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 de nossas duas listas de exemplo são mostradas na Figura 15.1. Note que os elementos de uma lista são mostrados horizontalmente. O último 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, para metanotação. Deveria existir um compilador que traduziria programas escritos em notação-M para programas em código de máquina equivalentes para o IBM 704.

² Na verdade, listas são a forma mais usada de uma estrutura de dados mais geral, o par entre pontos (*dotted pair*). Ignoraremos os pares entre pontos, exceto para explicar na Seção 15.5.8 como um deles pode ser criado acidentalmente.

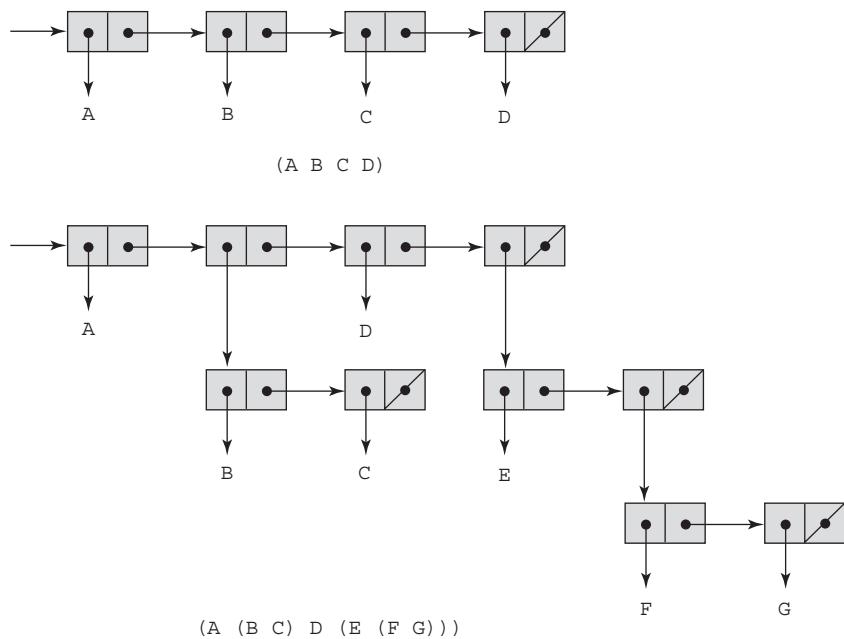


Figura 15.1 Representação interna de duas listas LISP.

No início do desenvolvimento de LISP, McCarthy decidiu escrever um artigo que promoveria o processamento de listas como uma abordagem para o processamento simbólico geral. McCarthy acreditava que o processamento de listas poderia ser usado para estudar computabilidade, na época estudada usando máquinas de Turing.

McCarthy achou que o processamento de listas simbólicas era um modelo mais natural de computação do que as máquinas de Turing, que operavam em símbolos escritos em fitas. Um dos requisitos mais comuns do estudo da computação é ser possível às pessoas provarem certas características de computabilidade da classe completa de qualquer modelo de computação que estiver sendo usado. No caso do modelo da máquina de Turing, alguém pode construir uma máquina de Turing universal capaz de mimetizar as operações de qualquer outra máquina de Turing. A partir desse conceito, veio a ideia de construir uma função LISP universal, possível de avaliar qualquer outra função em LISP.

O primeiro requisito para a função LISP universal era uma notação que permitisse as funções de serem expressas da mesma maneira que os dados eram. A notação de listas entre parênteses na Seção 15.4.1 já havia sido adotada para os dados de LISP, então decidiu-se inventar convenções para definições de funções e chamadas de funções que também poderiam ser expressas em no-

tação de lista. As chamadas a funções eram especificadas em uma forma de lista pré-fixada, chamada de Cambridge Polonesa, como a seguir:

(nome_da_função argumento_1 ... argumento_n)

Por exemplo, se + é uma função que recebe dois parâmetros numéricos, (+ 5 7)

é avaliada como 12.

A notação lambda descrita na Seção 15.2.1 foi escolhida para especificar definições de funções. Ela precisou ser modificada, entretanto, para permitir a vinculação de funções a nomes de forma que as funções pudessem ser referenciadas por outras e por elas próprias. A vinculação de nome era especificada por uma lista consistida no nome da função e uma lista contendo a expressão lambda, como em

(nome_da_função (LAMBDA (arg_1 ... arg_n) expressão))

Se você nunca teve exposição prévia à programação funcional, pode parecer estranho até mesmo considerar uma função sem nome. Entretanto, funções sem nomes são algumas vezes úteis em programação funcional (bem como na matemática). 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 ela é aplicada apenas no ponto de sua construção. Um exemplo disso é dado na Seção 15.5.10.

Funções LISP especificadas com essa nova notação eram chamadas de expressões-S, para 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. Iremos, em geral, referenciar-nos a expressões-S simplesmente como expressões.

McCarthy desenvolveu de maneira bem-sucedida uma função universal que poderia avaliar qualquer outra função. Ela foi chamada EVAL e era a própria na forma de uma expressão. Duas das pessoas no Projeto IA, 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).

Existiram diversos resultados importantes dessa implementação rápida, fácil e inesperada. Primeiro, todas as primeiras implementações de LISP copiaram EVAL e eram dessa forma interpretativas. Segundo, a definição da notação-M, a notação de programação planejada para LISP, nunca foi completada 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 15.5.12. Terceiro, muito do projeto original da linguagem foi efetivamente conge-

lado, mantendo certos recursos estranhos na linguagem, 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 acontecido de pouco pensamento ser dado 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.

15.5 UMA INTRODUÇÃO A SCHEME

Nesta seção, descrevemos uma parte de Scheme (Dybvig, 2003). Escolhemos Scheme porque é relativamente simples e popular em faculdades e universidades, e interpretadores Scheme estão prontamente disponíveis (gratuitamente) para uma ampla variedade de computadores. A versão de Scheme descrita nesta seção é Scheme 4. Note que esta seção cobre apenas uma pequena parte de Scheme, e não inclui nenhum dos recursos imperativos da linguagem.

15.5.1 Origens de Scheme

A linguagem Scheme, um dialeto de LISP, emergiu no MIT em meados dos anos 1970 (Sussman e Steele, 1975). Ela é caracterizada por seu pequeno tamanho, seu uso exclusivo de escopo estático e seu tratamento de funções como entidades de primeira classe. Como entidades de primeira classe, as funções Scheme podem ser os valores de expressões e elementos de listas, e podem ser atribuídas para variáveis e passadas como parâmetros. As primeiras versões de LISP não forneciam todas essas capacidades.

Como uma pequena linguagem com sintaxe e semântica simples, Scheme é bem adequada para aplicações educacionais, como cursos sobre programação funcional, e para introduções gerais a programação.

A maioria das funções Scheme nas seções a seguir requer apenas pequenas modificações para serem reescritas como funções LISP.

15.5.2 O interpretador Scheme

O interpretador Scheme é um laço infinito de leitura-avaliação-escrita. Ele repetidamente lê uma expressão digitada pelo usuário (na forma de uma lista), interpreta a expressão e mostra o valor resultante. 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 chamadas para funções primitivas são avaliadas da se-

guinte maneira: primeiro, cada uma das expressões de parâmetros é avaliada, em nenhuma ordem em particular. Então, a função primitiva é aplicada para os valores de parâmetros e o valor resultante é mostrado.

15.5.3 Funções numéricas primitivas

Esta subseção introduz Scheme ao discutir suas funções primitivas que tratam apenas com átomos numéricos, de maneira oposta aos átomos simbólicos e listas.

Scheme inclui funções primitivas para as operações aritméticas básicas, que são `+`, `-`, `*` 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 `+` adiciona todos os seus parâmetros juntos. A função `*` multiplica todos os seus parâmetros juntos. As funções `/` e `-` podem ter dois ou mais parâmetros. No caso da subtração, todos exceto o primeiro parâmetro são subtraídos do primeiro. A divisão é similar à subtração. Alguns exemplos são:

<i>Expressão</i>	<i>Valor</i>
<code>42</code>	<code>42</code>
<code>(* 3 7)</code>	<code>21</code>
<code>(+ 5 7 8)</code>	<code>20</code>
<code>(- 5 6)</code>	<code>-1</code>
<code>(- 15 7 2)</code>	<code>6</code>
<code>(- 24 (* 4 3))</code>	<code>12</code>

`SQRT` retorna a raiz quadrada de seu parâmetro numérico, se o valor do parâmetro não for negativo.

15.5.4 Definindo funções

Um programa Scheme é uma coleção de definições de funções. Consequentemente, saber como definir essas funções é um pré-requisito para escrever o programa mais simples. Lembre-se, da Seção 15.2.1, de que a forma das funções Scheme é baseada na notação lambda. Em Scheme, uma função não nomeada na verdade 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 uma **variável vinculada** dentro da expressão lambda. 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.

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. O primeiro uso pode levar a entender que **DEFINE** pode ser usada para criar variáveis no estilo das linguagens imperativas. Entretanto, essas vinculações de nomes criam constantes nomeadas, não variáveis.

DEFINE é chamada de uma forma especial porque ela 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 uma 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 mostrados podem ter mais dígitos do que os mostrados aqui.

Nomes em Scheme podem ser formados de letras, dígitos e caracteres especiais exceto parênteses; eles não são sensíveis à capitalização 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 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 a função, com o nome da função seguido pelos parâmetros formais, juntos em uma lista. A segunda lista contém uma expressão com a qual o nome será vinculado. A forma geral de tal **DEFINE**³ é

```
(DEFINE (nome_da_função parâmetros)
       (expressão)
     )
```

A seguinte chamada de exemplo a **DEFINE** vincula o nome *square* a expressão que o segue, que recebe um parâmetro:

```
(DEFINE (square number) (* number number))
```

³ Na verdade, a forma geral de **DEFINE** tem como corpo uma lista contendo uma sequência de uma ou mais expressões, apesar de, na maioria dos casos, apenas uma ser incluída. Incluímos apenas uma por questões de simplicidade.

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` não estivesse ainda vinculado a um valor, isso seria um erro. Além disso, se `x` já fosse definido, também seria um erro, porque `DEFINE` tentaria redefinir `x`, o que é ilegal.

A seguir, temos outro exemplo de uma 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 inclui algumas funções de saída simples, como

```
(DISPLAY expressão)
```

e

```
(NEWLINE)
```

com semântica óbvia. A maioria da saída dos programas Scheme, entretanto, é a normal do interpretador, mostrando os resultados de aplicar `EVAL` às funções de nível mais alto.

15.5.6 Funções de predicado numérico

Uma função de predicado é uma que retorna um valor booleano (seja verdadeiro ou falso). Scheme inclui uma coleção de funções de predicado para dados numéricos. Dentre essas, estão:

<i>Função</i>	<i>Significado</i>
=	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 para todas as funções de predicado pré-definidas que têm palavras para nomes terminam com pontos de interrogação. Em Scheme, os dois valores booleanos são #T e #F. As funções de predicado pré-definidas em Scheme retornam a lista vazia, (), em vez de #F (os dois são equivalentes). Qualquer lista não nula retornada por uma função de predicado é interpretada como #T. De forma a melhorar a legibilidade, todas as nossas funções de predicado neste capítulo retornam #F, em vez de () .

15.5.7 Controle de fluxo

Scheme usa três construções para controle de fluxo: uma modelada na construção de seleção das linguagens imperativas e duas baseadas no controle de avaliação usado em funções matemáticas.

O seletor de dois caminhos de Scheme, chamado IF, tem três parâmetros: uma expressão de predicado, uma expressão então e uma expressão senão. Uma chamada a IF tem a forma

(IF predicado expressão_então expressão_senão)

Por exemplo,

```
(DEFINE (factorial n)
  (IF (= n 0)
      1
      (* n (factorial (- n 1))))
  ))
```

Note o quanto fortemente o formato dessa função está relacionado ao da definição matemática de fatorial dada abaixo.

O controle de fluxo em definições de funções matemáticas é consideravelmente diferente do de programas em linguagens de programação imperativas. Enquanto funções em linguagens imperativas são definidas como coleções de sentenças que podem incluir diversos tipos de sentenças de controle de fluxo, funções matemáticas não têm sentenças múltiplas e usam apenas recursão e expressões condicionais para a avaliação do controle de fluxo. Por exemplo, a função fatorial pode ser definida com essas duas operações como em

$$f(n) \equiv \begin{cases} 1 & \text{if } n = 0 \\ n * f(n - 1) & \text{if } n > 0 \end{cases}$$

Uma expressão condicional matemática é na forma de uma lista de pares, cada um com uma expressão com guarda. Cada expressão com guarda consiste

em uma guarda de predicado e uma expressão. O valor de tal expressão condicional é o valor da expressão associada com o predicado que é verdadeiro. Apenas um dos predicados pode ser verdadeiro para um dado parâmetro ou lista de parâmetros.

O seletor múltiplo de Scheme, baseado em expressões condicionais matemáticas, é uma forma especial chamada COND. COND é uma versão levemente 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 requer um número fixo de parâmetros reais. Cada parâmetro para COND é um par de expressões no qual a primeira é um predicado.

A forma geral de COND é

```
(COND
  (predicado_1 expressão)
  (predicado_2 expressão)
  ...
  (predicado_n expressão)
  [(ELSE expressão)])
)
```

onde a cláusula ELSE é opcional.

A semântica de COND é a seguinte: os predicados dos parâmetros são avaliados um de cada vez, 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 é avaliada e o valor, retornado. Se nenhum dos predicados for verdadeiro e não existir um ELSE, o valor de COND é não especificado. Logo, todos os CONDS devem incluir um ELSE.

Note a similaridade entre um COND e a sentença de seleção múltipla com uma cláusula “otherwise” no final, como na sentença **case** de Ada.

A seguir, temos um exemplo de uma função simples que usa COND:

```
(DEFINE (compare x y)
  (COND
    ((> x y) "x is greater than y")
    ((< x y) "y is greater than x")
    (ELSE "x and y are equal"))
)
```

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 controlar repetição. A maioria das funções de exemplo na Seção 15.5.10 usa recursão.

15.5.8 Funções de lista

O uso mais comum de linguagens de programação baseadas em LISP – na verdade, o uso mais comum das linguagens funcionais – é o processamento de listas. Esta subseção introduz as funções de Scheme para lidar com listas.

A primeira primitiva de Scheme descrita aqui é uma função utilitária necessária pela natureza da operação de aplicação de funções de Scheme, EVAL. Ela é chamada para tratar da parte de avaliação da ação leitura-avaliação-escrita do interpretador Scheme. Quando aplicada a uma função primitiva, EVAL primeiro avalia os parâmetros da função. Essa ação é necessária quando os parâmetros reais de uma chamada a função são eles próprios chamadas a funções, o que com frequência é o caso. 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, ele 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 da função seja determinar se o dado átomo está na lista dada. Nem o átomo e nem a lista devem ser avaliados; eles são dados literais a serem examinados. Para evitar avaliar um parâmetro, ele primeiro é passado como um 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)
```

No restante deste capítulo, a abreviação comum da chamada a QUOTE é usada, feita simplesmente precedendo a expressão a ser citada por um apóstrofo ('). Então, em vez de (QUOTE (A B)), '(A B) poderia ser usado.

Uma linguagem de programação para processamento de listas deve incluir primitivas para manipular listas. Em particular, deve fornecer operações para selecionar partes de uma lista, o que em certo sentido desmantela a lista, e ao menos uma operação para construir listas. Existem dois seletores de listas primitivos em Scheme: CAR e CDR (pronuncia-se “could-er”). A função CAR retorna o primeiro elemento de uma lista. Os seguintes exemplos ilustram CAR:

```
(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
```

A função CDR retorna o restante de uma lista após seu CAR ter sido removido:

```
(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 peculiares, na melhor hipótese. A origem desses nomes é resultante da 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 também incluía duas instruções de máquina, também chamadas de CAR (Contents of Address Register – Conteúdo do Registrador de Endereço) e CDR (Contents of Decrement Register – Conteúdo do Registrador de Decremento), que extraiam 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 poderia armazenar um nó muito bem. Usando essas convenções, as instruções CAR e CDR do 704 forneceram seletores de listas eficientes. Os nomes continuaram nas primitivas de todos os dialetos de LISP.

Como outro exemplo de uma função simples, considere

```
(DEFINE (second lst) (CAR (CDR lst)))
```

Uma vez que essa função for avaliada, ela pode ser usada, como em

```
(second '(A B C))
```

que retorna B.

CONS é um construtor de lista primitivo. Ele constrói uma lista a partir de seus dois argumentos. O primeiro pode ser um átomo ou uma lista; o segundo é normalmente uma lista. CONS insere o primeiro parâmetro como o novo CAR de seu segundo parâmetro. Considere os seguintes exemplos:

```
(CONS 'A '()) retorna (A)
(CONS 'A '(B C)) retorna (A B C)
(CONS '() '(A B)) retorna ((A B) 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. CAR e CDR separam uma lista, e CONS constrói uma nova lista a partir de partes de uma lista. Os dois parâmetros de CONS se tornam o CAR e o CDR da nova lista. Logo, se lis é uma lista, então

```
(CONS (CAR lis) (CDR lis))
```

retorna uma lista idêntica a lis.

Lidando apenas com os problemas relativamente simples e os programas discutidos neste capítulo, provavelmente ninguém aplicaria intencionalmente CONS a dois átomos. Entretanto, isso é permitido e comumente acontece de

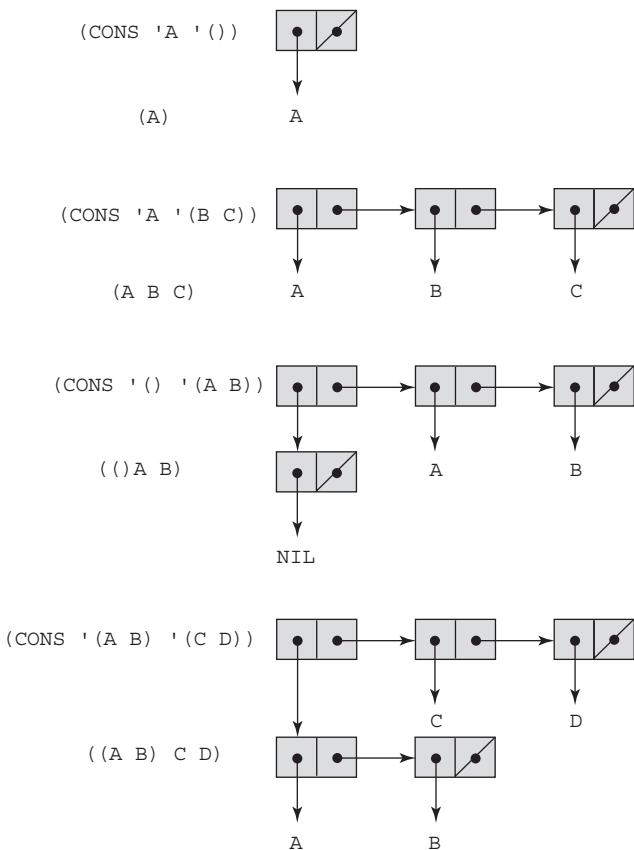


Figura 15.2 O resultado de diversas operações CONS.

forma acidental. O resultado de tal aplicação é um par entre pontos, nomeados dessa forma por causa da maneira com que são mostrados em Scheme. Por exemplo, considere a seguinte chamada:

`(CONS 'A 'B)`

Se o resultado disso for mostrado, ele aparece como

`(A . B)`

Esse par entre pontos 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 em

```
(LIST 'apple 'orange 'grape) retorna (apple orange 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 dois átomos simbólicos como parâmetros. Ela retorna `#T` se ambos os parâmetros forem átomos e ambos forem o mesmo; caso contrário, retorna `#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)) returns #F or #T
```

Como o último caso indica, o resultado de comparar listas com `EQ?` é dependente da implementação, algumas levam a `#T` e outras a `#F`. A razão para essa diferença é que `EQ?` é geralmente implementada como uma comparação de ponteiros (dois ponteiros dados apontam para o mesmo lugar?), e duas listas que são exatamente a mesma são frequentemente não duplicadas na memória. No momento em que o sistema Scheme cria uma lista, ele verifica se já não existe tal lista. Se existir, a nova lista é nada mais que um ponteiro para a 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`.

Note que `EQ?` funciona para átomos simbólicos, mas não funciona 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 ser capaz de testar dois átomos em relação a sua igualdade 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. A razão primária para usar `EQ?` ou `=` em vez de `EQV?` quando for possível é que `EQ?` e `=` são mais rápidas do que `EQV?`.

A função de predicado `LIST?` retorna `#T` se seu único argumento é uma lista e `#F` 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 ela é a lista vazia e retorna `#T` caso seja. 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, ele é uma lista contendo um elemento, a lista vazia.

15.5.10 Funções de exemplo 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 uma **lista simples**. Se a função é chamada `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 usando recursão. A função pode comparar o átomo com o `CAR` da lista. Se eles casam, o valor `#T` é retornado. Se eles não casam, o `CAR` da lista deve ser ignorado e a busca deve continuar no `CDR` da lista. Isso pode ser feito ao termos a função chamando ela mesma com o `CDR` como o parâmetro de lista e retornando 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 maneiras de sair da recursão: ou a lista está vazia em alguma chamada e `#F` é retornado, ou um casamento é encontrado 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, que causa a chamada recursiva. Esses

são os três parâmetros para COND, com o último sendo o caso padrão disparado por um predicado ELSE. A função completa é a seguinte:

```
(DEFINE (member atm lis)
  (COND
    ((NULL? lis) #F)
    ((EQ? atm (CAR lis)) #T)
    (ELSE (member atm (CDR lis))))
  ))
```

Essa forma é típica de simples funções de processamento de listas em Scheme. Em tais funções, os dados na lista são processados um elemento de cada vez. Os elementos individuais são especificados com CAR, e o processo é continuado usando recursão no CDR da lista.

Note que o teste de nulo deve preceder o teste de igual, porque aplicar CAR a uma lista vazia é um erro.

Como outro exemplo, considere o problema de determinar se duas listas são iguais. Se as duas listas são simples, a solução é relativamente fácil, apesar de algumas técnicas de programação com as quais o leitor pode não estar familiarizado estarem envolvidas. Uma função de predicado, equalsimp, para comparar listas simples é mostrada aqui:

```
(DEFINE (equalsimp lis1 lis2)
  (COND
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) #F)
    ((EQ? (CAR lis1) (CAR lis2))
     (equalsimp (CDR lis1) (CDR lis2)))
    (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 CDRs dos dois parâmetros de lista como seus parâmetros, a primeira lista pode ser vazia em tal chamada se tiver todos os seus elementos removidos por chamadas recursivas prévias. Quando a primeira lista está vazia, também deve ser verificado se a segunda está vazia. Se estiver, elas são iguais (seja inicialmente ou se os CARS eram iguais em todas as chamadas recursivas prévias), e NULL? corretamente retorna #T. Se a segunda lista não é vazia, ela é maior do que a primeira lista e #F deve ser retornado, como ocorre usando NULL?. Lembre-se de que qualquer lista não vazia retornada por uma função de predicado é interpretada como #T.

O próximo caso trata com a segunda lista sendo vazia quando a primeira não é. Essa situação ocorre quando a primeira lista é maior do que a segunda. Apenas a segunda lista deve ser testada, porque o primeiro caso captura todas as instâncias da primeira lista ser vazia.

O terceiro caso é o passo recursivo que testa pela igualdade entre dois elementos correspondentes nas duas listas. Isso é feito comparando-se os CARS das duas listas não vazias. Se eles forem iguais, as duas listas estão iguais até esse ponto, então a recursão é usada nos CDRS das duas. Esse caso falha quando dois átomos não iguais são encontrados. 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 opera corretamente se um ou ambos os parâmetros forem átomos.

O problema de comparar listas gerais é levemente mais complexo do que isso, porque as sublistas podem ser rastreadas completamente no processo de comparação. Nessa situação, o poder da recursão é unicamente apropriado, porque a forma das sublistas é a mesma das listas. Em qualquer momento 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 abordagem dividir-e-conquistar. Se os elementos correspondentes das duas listas são átomos, eles podem ser simplesmente comparados usando EQ?.

A definição da função completa é:

```
(DEFINE (equal lis1 lis2)
  (COND
    ((NOT (LIST? lis1)) (EQ? lis1 lis2))
    ((NOT (LIST? lis2)) #F)
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) #F)
    ((equal (CAR lis1) (CAR lis2))
     (equal (CDR lis1) (CDR lis2)))
    (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 quarto casos são para a situação em que uma ou ambas as listas são vazias. Esses casos também previnem casos subsequentes de tentarem aplicar CAR a uma lista vazia. O quinto caso de COND é o mais interessante. O predicado é uma chamada recursiva com os CARS das listas como parâmetros. Se essa chamada recursiva retorna #T, a recursão é usada mais uma vez nos CDRS 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 ela é muito mais lenta do que EQ? e EQV?.

Outra operação de listas comumente necessária é uma 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 pas-

sada 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

```
(DEFINE (append lis1 lis2)
  (COND
    ((NULL? lis1) lis2)
    (ELSE (CONS (CAR lis1) (append (CDR lis1) lis2))))
  ))
```

O primeiro caso `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 `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. Assuma que os parâmetros sejam listas simples.

```
(DEFINE (guess lis1 lis2)
  (COND
    ((NULL? lis1) '())
    ((member (CAR lis1) lis2)
     (CONS (CAR lis1) (guess (CDR lis1) lis2)))
    (ELSE (guess (CDR lis1) lis2)))
  ))
```

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.

`LET` é uma função que permite aos nomes serem temporariamente vinculados a valores de subexpressões. Ela é geralmente usada para fatorar subexpressões comuns de expressões mais complicadas. Esses nomes podem ser usados na avaliação de outra expressão. Sua forma geral é

```
(LET (
  (nome_1 expressão_1)
  (nome_2 expressão_2)
  ...
  (nome_n expressão_n))
  expressão { expressão}
)
```

A semântica de `LET` é que as primeiras n expressões são avaliadas e os valores resultantes são vinculados aos seus nomes associados. Então, as ex-

pressões no corpo, as quais consistem dos parâmetros restantes, são avaliadas. O resultado de `LET` é o valor da última expressão em seu corpo. O seguinte exemplo ilustra o uso de `LET`. Ele computa a raiz de uma equação quadrática, assumindo que as raízes são reais⁴:

```
(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 criam o resultado.

`LET` cria um novo escopo estático local de uma maneira praticamente igual ao `declare` de Ada. Novas variáveis podem ser criadas, usadas e então descartadas quando o final do novo escopo for alcançado. Os componentes nomeados de `LET` são parecidos com sentenças de atribuição, mas podem ser usados apenas no novo escopo de `LET`. Além disso, não podem ser revinculados a novos valores em `LET`.

`LET` é, na verdade, apenas um atalho para uma expressão `LAMBDA`. As duas expressões 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`.

15.5.11 Recursão em cauda em Scheme

A recursão em cauda foi introduzida na Seção 15.3. Esta seção discute com mais detalhes esse tópico e a ilustra em Scheme.

Lembre-se de que uma função é **recursiva em cauda** se sua chamada recursiva é a última operação na função. Por exemplo, a função membro da Seção 15.5.10, repetida aqui, é recursiva em cauda.

```
(DEFINE (member atm lis)
  (COND
    ((NULL? lis) #F)
    ((EQ? atm (CAR lis)) #T)
    (ELSE (member atm (CDR lis))))
  ))
```

⁴ Algumas versões Scheme incluem tipos de dados “complexos” e computarão as raízes da equação, independentemente de elas serem reais ou complexas.

Essa função pode ser automaticamente convertida por um compilador para usar iteração, resultando em uma execução mais rápida do que em 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 usa um parâmetro acumulador e uma função auxiliar. Como um exemplo dessa abordagem, considere a função factorial da seção 15.5.7, repetida aqui.

```
(DEFINE (factorial n)
  (IF (= n 0)
    1
    (* n (factorial (- n 1))))
  ))
```

A última operação dessa função é a multiplicação. A função trabalha criando a lista de números a serem multiplicados juntos e então faz as multiplicações para produzir o resultado. Essa função 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 0)
    factpartial
    facthelper((- n 1) (* n factpartial)))
  ))
(DEFINE (factorial n)
  (facthelper n 1)
)
```

A definição da linguagem Scheme requer que os sistemas 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. Alguns compiladores otimizadores 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.

15.5.12 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.

15.5.12.1 Composição funcional

A composição funcional é a única forma funcional primitiva fornecida pelo LISP original. Todos os dialetos subsequentes de LISP, incluindo Scheme, também a provêm. A composição funcional é a essência de como EVAL funciona. Assume-se que todas as listas sem aspas são chamadas a funções, o que requer parâmetros avaliados primeiro. Isso se aplica recursivamente até a menor lista em qualquer expressão, precisamente o que a composição funcional significa. Os exemplos a seguir ilustram a composição funcional:

```
(CDR (CDR '(A B C))) retorna (C)
(CAR (CAR '((A B) B C))) retorna A
(CDR (CAR '((A B C) D))) retorna (B C)
(NULL? (CAR '((() B C)))) retorna #T
(CONS (CAR '(A B)) (CDR '(A B))) retorna (A B)
```

Note que os nomes das funções em chamadas internas não estão entre aspas, porque eles devem ser avaliados em vez de tratados como dados literais.

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 (CADDR x) é equivalente a (CAR (CDR (CDR (CAR x)))). Qualquer combinação de As e Ds, até quatro, são legais entre o C e o R no nome da função.

15.5.12.2 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. A mais simples dessas é mapcar, que tem dois parâmetros: uma função e uma lista. A forma funcional mapcar aplica a função a cada elemento da lista, e ela retorna uma lista de resultados dessas aplicações. Uma definição em Scheme de mapcar é mostrada a seguir:

```
(DEFINE (mapcar fun lis)
  (COND
    ((NULL? lis) '())
    (ELSE (CONS (fun (CAR lis)) (mapcar fun (CDR lis))))
  ))
```

Note a forma simples de mapcar, que expressa uma forma funcional complexa. Essa função é um atestado do grande poder de expressividade de Scheme.

Como um exemplo do uso de mapcar, suponha que quiséssemos todos os elementos de uma lista elevados ao cubo. Podemos realizar isso com:

```
(mapcar (LAMBDA (num) (* num num num)) '(3 4 2 6))
```

Essa chamada retorna (27 64 8 216).

Note que, nesse exemplo, o primeiro parâmetro para `mapcar` é uma expressão `LAMBDA`. Quando `EVAL` avalia a expressão `LAMBDA`, ela constrói uma função com a mesma forma que qualquer função pré-definida, 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.

15.5.13 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 é uma função chamada `EVAL`. O sistema Scheme aplica `EVAL` a cada expressão digitada no interpretador de comandos de Scheme. A função `EVAL` também pode ser chamada diretamente por programas Scheme. Isso fornece a possibilidade de um programa criar expressões e chamar `EVAL` para avaliá-las.

Um dos exemplos mais simples desse processo envolve átomos numéricos. Scheme inclui uma função chamada `+`, que recebe qualquer número de átomos numéricos como argumentos e retorna sua soma. Por exemplo, `(+ 3 7 10 2)` retorna 22.

Nosso problema é: suponha que em um programa tivéssemos uma lista de átomos numéricos e precisássemos da soma. Não podemos aplicar `+` diretamente na lista, porque `+` pode ser aplicado 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 usando seu `CDR`, usando recursão para percorrer a lista. Tal função é mostrada a seguir:

```
(DEFINE (adder lis)
  (COND
    ((NULL? lis) 0)
    (ELSE (+ (CAR lis) (adder (CDR lis)))))
  ))
```

Uma solução alternativa para o problema seria escrever uma função que construísse uma chamada a `+` com forma de parâmetros apropriada. Isso pode ser feito usando `CONS` para inserir o átomo `+` na lista de números. Essa nova lista pode então ser submetida à `EVAL` para avaliação, como:

```
(DEFINE (adder lis)
  (COND
    ((NULL? lis) 0)
    (ELSE (EVAL (CONS '+ lis))))
  ))
```

Note que o nome da função `+` é colocado com uma aspa para prevenir que `EVAL` o avaliasse na avaliação de `CONS`. Como um exemplo, considere que a chamada

```
(adder ' (3 4 6) )
```

fizesse `adder` construir a lista

```
(+ 3 4 6)
```

A lista é então submetida à `EVAL`, que invoca `+` e retorna o resultado, 13.

Em todas as versões anteriores de Scheme, a função `EVAL` avaliava sua expressão no escopo mais externo do programa. A última versão de Scheme, Scheme 4, requer um segundo parâmetro para `EVAL` que especifica o escopo no qual a expressão será avaliada. Por questões de simplicidade, deixamos o parâmetro de escopo de fora de nosso exemplo, e não discutimos nomes de escopo aqui.

15.6 COMMON LISP

COMMON LISP (Steele, 1984) foi criada em um esforço para combinar os recursos de diversos dialetos anteriores de LISP, incluindo Scheme, em uma única linguagem. Sendo algo como a união de linguagens, ela é significativamente grande e complexa. Sua base, entretanto, é o LISP original, então sua sintaxe, funções primitivas e natureza fundamental vêm dessa linguagem.

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 declarar uma variável como “especial” (`special`), o escopo dela se torna dinâmico.

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 poderosas; e uma forma de pacotes para modular coleções de funções e de dados, e também para fornecer controle de acesso.

Em certo sentido, Scheme e COMMON LISP são opostos. Scheme é muito menor e semanticamente mais simples, em parte por causa do uso exclusivo de escopo estático. COMMON LISP foi criada para ser uma linguagem comercial e foi bem-sucedida ao se tornar uma linguagem amplamente utilizada para aplicações de IA. Scheme, por outro lado, é mais usada em cursos de faculdades para o ensino de programação funcional. É mais provável de ela ser estudada como uma linguagem funcional por causa de seu pequeno tamanho. Um critério de projeto importante de COMMON LISP que a transformou em uma linguagem muito grande foi o desejo de torná-la compatível com diversos dos dialetos de LISP dos quais ela foi derivada.

15.7 ML

ML (Milner et al., 1990) 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. Talvez a diferença mais importante é ML ser uma linguagem fortemente tipada, enquanto Scheme é essencialmente desprovida de tipos. ML possui declarações de tipos, apesar de, por causa de sua inferência de tipos (discutida brevemente no Capítulo 5), elas não serem frequentemente usadas. O tipo de cada variável e expressão podem ser determinados em tempo de compilação. Outra diferença importante entre Scheme e ML é que esta usa uma sintaxe mais fortemente relacionada a uma linguagem imperativa do que a de LISP. Por exemplo, expressões aritméticas são escritas em ML usando a notação infixa (a mais comumente usada em linguagens imperativas).

ML tem tratamento de exceções e um recurso de módulo para implementar tipos de dados abstratos. Um breve histórico do desenvolvimento e dos recursos primários de ML é dado no Capítulo 2.

Declarações de funções em ML aparecem na forma geral

```
fun nome_da_função (parâmetros_formais) = expressão_corpo_da_função;
```

Por exemplo,

```
fun square (x : int) = x * x;
```

O tipo do valor de retorno pode ser especificado após a lista de parâmetros, como em

```
fun square(x : int) : int = x * x;
```

É claro, nessa função, o tipo do valor de retorno não precisa ser explicitamente especificado.

A linha a seguir

```
fun square (x) = x * x;
```

define a mesma função da última definição de função, porque ML assume o tipo **int**, já que o operador na expressão ($x * x$) indica que x é algum tipo numérico. Então, funções que usam operadores aritméticos não podem ser polimórficas. O mesmo é verdade para funções que usam operadores relacionais, exceto $=$ e $<>$, e operadores booleanos. Entretanto, funções que usam apenas operadores de lista, $=$, $<>$, e operadores de tupla (aqueles para formar tuplas e para seleção de componentes), podem ser polimórficos.

Se quiséssemos uma função **square** para valores de tipo **real** e já tivéssemos definido **square** para valores inteiros (**int**), precisaríamos defini-la com um nome diferente, porque funções definidas pelo usuário sobrecarregadas não são permitidas. Além disso, se definíssemos **square**

para parâmetros reais, ela não poderia ser chamada com um parâmetro real inteiro, porque ML, assim como Ada, não faz coerção de valores inteiros para o tipo `real`.

A construção de fluxo de controle de seleção de ML é similar àquelas das linguagens imperativas. Ela tem a forma geral

```
if expressão then expressão_então else expressão_senão
```

onde a primeira expressão deve avaliar para um valor booleano.

As expressões condicionais de Scheme podem aparecer no nível de definição de função em ML. 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 mimetizar a forma e o significado de definições funcionais condicionais na matemática. Em ML, a expressão particular que define o valor de retorno de uma função é escolhida pelo casamento de padrão em relação ao parâmetro dado. Por exemplo, sem usar esse casamento de padrão, uma função para computar o fatorial poderia ser escrita como:

```
fun fact(n : int) : int = if n = 0 then 1  
                           else n * fact(n - 1);
```

Múltiplas definições de uma função podem ser escritas usando 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

```
fun fact(0) = 1  
|   fact(n : int) : int = n * fact(n - 1);
```

Se `fact` fosse chamada com o parâmetro real `0`, a primeira definição seria usada; se um valor `int` diferente de zero fosse informado, a segunda definição seria usada.

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, 5, 9]
```

é avaliada como `[3, 5, 7, 9]`.

Os elementos de uma lista devem ser do mesmo tipo, então a lista a seguir seria ilegal:

```
[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]
```

Por causa da disponibilidade de parâmetros de funções baseados em padrões, as funções `hd` e `tl` são muito menos usadas em ML do que em Scheme. Por exemplo, em um parâmetro forma, a expressão

```
(h :: t)
```

é, na verdade, dois parâmetros formais, o primeiro elemento e o restante da dada lista passada como parâmetro, 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 função APPEND de Scheme:

```
fun append([], lis2) = lis2
| append(h :: t, lis2) = h :: append(t, lis2);
```

O primeiro caso dessa função trata da situação da função quando 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 (CAR e CDR). O primeiro elemento é anexado no resultado da função recursiva usando `CONS`, que usa a cauda da lista como seu primeiro parâmetro.

Em ML, os nomes podem ser vinculados a valores com sentenças de declarações de valores na forma

```
val novo_nome = expressão;
```

Por exemplo

```
val distance = time * speed;
```

Não pense que essa sentença 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. Na verdade, se você realmente revincular um nome com uma segunda sentença `val`, isso faz criar uma nova entrada no ambiente não relacionada à versão anterior do nome⁵. Na verdade, após a nova vinculação, a entrada de ambiente anterior (para a

⁵ O ambiente pode ser visto como uma tabela de símbolos que armazena nomes, e também os valores aos quais os nomes são vinculados, durante a execução.

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 atual e o vinculam a um valor, como a forma especial `LET` de Scheme. O uso normal de `val` é em uma expressão `let`, cuja forma geral é

```
let val novo_nome = expressão_1 in expressão_2 end
```

Por exemplo,

```
let
  val pi = 3.14159
in
  pi * radius * radius
end;
```

Não existem coerções de tipo em ML; os tipos dos operandos de um operador ou atribuição simplesmente devem casar para evitar erros de sintaxe. ML também possui tipos enumerados, matrizes e tuplas, similares aos registros.

ML tem tido um impacto significativo na evolução das linguagens de programação. Para pesquisadores de linguagens, ela se tornou uma das mais estudadas. Além disso, gerou diversas linguagens subsequentes.

Caml foi desenvolvida no Instituto Nacional Francês para Pesquisa em Ciência da Computação e Controle (INRIA) em meados dos anos 1980. Caml começou com ML, portanto, usa a sintaxe, o sistema de tipos e a inferência de tipos de ML. Em meados dos anos 1990, o suporte a programação orientada a objetos foi adicionada a Caml e foi renomeada para Objective Caml, ou apenas OCaml.

O desenvolvimento de F# na Microsoft era uma parte do projeto .NET. F# inclui OCaml como sua linguagem básica. A parte interessante de F# é que ela é uma linguagem de primeira classe no Framework .NET. Isso significa que os programas F# 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 a partir da Microsoft (<http://research.microsoft.com/fsharp/fsharp.aspx>).

15.8 HASKELL

Haskell (Thompson, 1999) é similar a ML no sentido de que usa uma sintaxe similar, tem escopo estático, é fortemente tipada e usa o mesmo método de inferência. Existem duas características de Haskell que a separam de ML. Primeiro, as funções em Haskell podem ser polimórficas (a maioria das 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. Tanto as semânticas não estritas quanto o polimorfismo são discutidos em maiores detalhes posteriormente.

O código desta seção é escrito na versão 1.4 de Haskell.

Considere a seguinte definição da função factorial, que usa casamento de padrões em seus parâmetros:

```
fact 0 = 1
fact n = n * fact (n - 1)
```

Note as diferenças sintáticas entre essa definição e sua versão ML na Seção 15.7. Primeiro, não existe uma palavra reservada para introduzir a definição de função (fun em ML). Segundo, não são usados parênteses para delimitar os parâmetros formais⁶. Terceiro, definições alternativas de funções (com parâmetros formais diferentes) têm todas 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 > 0 = n * fact(n - 1)
```

Essa definição de factorial é mais precisa do que a anterior, visto que restringe a faixa dos valores dos parâmetros reais para aqueles com os quais ela funciona. O casamento de padrões poderia, é claro, falhar nesse uso, visto que o padrão de parâmetro é n para ambas as expressões de valores. Essa forma de uma definição de função é chamada de uma expressão condicional, nomeadas por causa das expressões matemáticas nas quais são baseadas.

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.

⁶ Os parênteses são, na verdade, opcionais em ML.

Considere a seguinte definição de função, cujo propósito é o mesmo da função ML correspondente na Seção 15.7:

```
square x = x * x
```

Nesse caso, entretanto, por causa do suporte de Haskell para polimorfismo, essa função pode receber um parâmetro de qualquer tipo numérico.

Como com ML, as listas são escritas em colchetes em Haskell, como em

```
colors = ["blue", "green", "red", "yellow"]
```

Haskell inclui uma coleção de operadores de lista. Por exemplo, listas podem ser concatenadas com `++`, `:` serve como uma versão infixada de `CONS`, e `..` é usado para especificar séries aritméticas. Por exemplo,

```
5:[2, 7, 9]  results in [5, 2, 7, 9]
[1, 3..11] results in [1, 3, 5, 7, 9, 11]
[1, 3, 5] ++ [2, 4, 6] results in [1, 3, 5, 2, 4, 6]
```

Note que o operador `:` é simplesmente similar ao operador `::` de ML⁷. Usando `:` e casamento de padrões, podemos definir uma função simples para computar o produto de uma lista de números:

```
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]
```

Compreensões de lista, suportadas por Python e introduzidas no Capítulo 6, fornecem um método de descrever listas que representam conjuntos. A sintaxe de uma compreensão de lista em Haskell é a mesma da

⁷ É interessante que ML use `:` para anexar um nome de tipo a um nome e `::` para `CONS`, enquanto que Haskell usa esses dois operadores de maneiras opostas.

quela geralmente usada para descrever conjuntos na matemática, cuja forma geral é

[corpo | qualificadores]

Por exemplo,

[n * n * n | n <- [1..50]]

define uma lista dos cubos dos números de 1 a 50. Ela é lida como “a lista de todos $n \cdot n \cdot n$ tal que n é obtido da faixa de 1 a 50”. Nesse caso, o qualificador está no formato de um **gerador**. Ele gera os números de 1 a 50. Em outros casos, os qualificadores estão na forma de expressões booleanas, nesse caso chamados de **testes**. Essa notação pode ser usada para descrever algoritmos para fazer muitas coisas, como encontrar permutações de listas e ordená-las. Por exemplo, considere a seguinte função, 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 $n \text{ `mod`} i$ é zero. Essa é, na verdade, uma definição bastante excessiva 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 mostrada 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 do que o primeiro é ordenado e concatenado com os resultados prévios. Essa definição do quicksort é significativamente menor e mais simples do 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 depende da ordem em que os parâmetros são avaliados. Uma linguagem é **não estrita** se ela não tem o requisito de ser estrita. Linguagens não estritas podem ter diversas vantagens em relação às linguagens estritas. Primeiro, linguagens não estritas são geralmente mais eficientes, porque algu-

ma avaliação é evitada⁸. Segundo, algumas capacidades interessantes são possíveis com linguagens não estritas e não são possíveis com linguagens estritas. Dentre essas, estão as listas infinitas. Linguagens não estritas 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. A avaliação preguiçosa significa que 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 execução em particular da função o primeiro não é usado, o parâmetro real passado para tal execução não será avaliado. 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 são avaliados apenas uma vez, se é que o são, mesmo se esse parâmetro real aparecer mais de uma vez em uma chamada a função.

Conforme mencionado, a avaliação preguiçosa permite que alguém defina 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 previne o seu uso se a avaliação preguiçosa for usada. Por exemplo, se quiséssemos verificar a lista de quadrados com uma função de pertinência. Suponha que tivéssemos uma função de predicado chamada `member` que terminasse se uma lista contivesse um dado átomo. Então, poderíamos usá-la como em

```
member squares 16
```

que retornaria `True`. A definição de `square` 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 (a:x) b = (a == b) || member x b
```

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 ou (`b`) se a chamada recursiva com a cauda retornar `True`.

A definição de `member` funcionaria corretamente com `squares` apenas se o número fosse um quadrado perfeito. Se não, `squares` continuaria gerando quadrados para sempre, ou até ocorrer alguma limitação de memória, bus-

⁸ Note como isso está relacionado com a avaliação em curto-circuito de expressões booleanas, feita em algumas linguagens imperativas.

cando 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 do que o número buscado fosse encontrado.

```
member2 (m:x) n
| m < n      = member2 x n
| m == n     = True
| otherwise   = False
```

A avaliação preguiçosa algumas vezes fornece uma ferramenta de modularização. Considere a composição funcional. Suponha que em um programa exista uma chamada a uma função f e o parâmetro para f é o valor de retorno de uma função g ⁹. Então, temos $f(g(x))$. Em seguida, suponha que g produzisse uma grande quantidade de dados, um pouco de cada vez, e que f devesse processar esses dados, um pouco de cada vez. Com avaliação preguiçosa, as execuções de f e g poderiam, de forma implícita, ser fortemente 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 ela produzisse uma quantidade de saída infinita. A função g seria forçada a terminar quando f terminasse. Então, sob avaliação preguiçosa, g rodaria 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 escolheria o subconjunto apropriado.

No entanto, a avaliação preguiçosa não ocorre sem custos. Seria certamente surpreendente se tal poder de expressão e tal flexibilidade fossem de graça. Nesse caso, o custo é em uma semântica muito mais complicada, resultando em uma velocidade de execução muito mais lenta.

15.9 APLICAÇÕES DE LINGUAGENS FUNCIONAIS

Nos últimos 50 anos na história das linguagens de programação de alto nível, apenas algumas linguagens funcionais tiveram seu uso disseminado. A mais proeminente dentre elas é LISP.

LISP é uma linguagem versátil e poderosa. Em seus primeiros 15 anos, pensava-se nela, em sua maioria pelos não usuários, como uma linguagem de aparência estranha e comportamento esquisito e muito cara de usar. De fato, era comum nos anos 1960 e no início dos 1970 pensar em duas categorias de linguagens: uma contendo LISP e outra com todas as outras linguagens de programação.

LISP foi desenvolvida para computação simbólica e aplicações de processamento de listas, que residem basicamente no universo da computação de

⁹ Esse exemplo aparece em Hughes (1989).

IA. Em muitas aplicações de IA, LISP e suas derivadas ainda são o padrão em termos de linguagens.

Dentro de IA, diversas áreas têm sido desenvolvidas, primariamente por meio do uso de LISP; apesar de outros tipos de linguagens poderem ser usadas – para algumas aplicações de IA, linguagens de programação lógicas; para outras, linguagens imperativas são usadas. A maioria dos sistemas especialistas existentes, por exemplo, foi desenvolvida em LISP. A linguagem também domina nas áreas de representação de conhecimento, aprendizagem de máquina, sistemas de treinamento inteligentes e modelagem de fala.

Fora de IA, LISP também tem sido bem-sucedida para algumas aplicações. Por exemplo, o editor de texto Emacs é escrito em LISP, assim como o sistema de matemática simbólica MACSYMA, que faz diferenciação e integração simbólicas, dentre outras coisas. A máquina LISP era um computador pessoal cujo software de sistema foi escrito em LISP. LISP também tem sido bem-sucedida para construir sistemas experimentais em uma variedade de áreas de aplicação.

15.10 UMA COMPARAÇÃO ENTRE LINGUAGENS FUNCIONAIS E IMPERATIVAS

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.

A semântica das linguagens funcionais também é mais simples do que a das imperativas. Por exemplo, na descrição em semântica denotacional de uma construção de laço imperativa dada na Seção 3.5.3, o laço é convertido de uma construção iterativa para uma construção recursiva. Essa conversão é desnecessária em uma linguagem funcional pura, onde não existe iteração. Além disso, assumimos 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, porque todas as linguagens baseadas em C incluem efeitos colaterais em expressões. Essa restrição não é necessária para as descrições denotacionais de linguagens funcionais puras.

Algumas pessoas na comunidade de programação funcional afirmam que seu uso resulta em um aumento na produtividade de uma ordem de magnitude, devido aos programas funcionais terem apenas 10% do tamanho de seus correspondentes imperativos. Embora tais números tenham sido de fato mostrados para certas áreas de problemas, para outras os programas funcionais são 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 produtivida-

de. Certamente, nem todas as linhas de código fonte têm uma complexidade igual, nem levam o mesmo tempo para serem produzidas. Na verdade, por causa da necessidade de lidar com variáveis, os programas imperativos têm muitas linhas trivialmente simples para inicializar e realizar pequenas mudanças às variáveis.

A eficiência de execução é outra base para comparação. Quando programas funcionais são interpretados, eles são mais lentos do que seus correspondentes imperativos. 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 grandes. Alguém pode ficar tentado a afirmar que, como os programas funcionais são significativamente menores do que os imperativos equivalentes, eles devem executar muito mais rápido do que os imperativos. Entretanto, isso geralmente não é o caso, por causa de uma coleção de características das linguagens funcionais que têm um forte 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). Isso pode parecer uma diferença significativa que poderia 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 importante. Por exemplo, considere que muitos programas escritos em linguagens imperativas, como software para a Web escrito em JavaScript e PHP, são interpretados e, dessa forma, são muito mais lentos do que suas versões equivalentes compiladas. Para essas aplicações, a velocidade de execução não é prioridade.

Outra vantagem em potencial das linguagens funcionais é a legibilidade. Em muitos programas imperativos, os detalhes de lidar com variáveis obscurece a lógica do programa. Considere uma função que computa a soma dos cubos dos primeiros n positivos inteiros. Em C, essa função 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. Construa a lista de números ([1..n]).
2. Crie uma nova lista mapeando uma função que computa a terceira potência de cada um dos números da lista.
3. Some a nova lista.

Por causa dos detalhes de variáveis e de controle de iteração, essa versão é mais legível do que a em C*.

A execução concorrente nas linguagens imperativas é difícil de projetar e de usar. Por exemplo, considere o modelo de tarefas de Ada, no qual a cooperação entre tarefas concorrentes é responsabilidade do programador. Programas funcionais podem ser executados primeiro por meio de sua tradução em grafos. Esses grafos podem então ser executados por um processo de redução de grafos, podendo ser feito com uma boa dose de concorrência que não foi especificada pelo programador. A representação em grafo naturalmente expõe muitas oportunidades para execução concorrente. Essa concorrência é possível, em parte, por causa da falta de efeitos colaterais em linguagens funcionais puras. A sincronização de cooperação nesse processo não é uma preocupação do programador. Uma descrição detalhada desse processo está além do escopo deste livro.

Em uma linguagem imperativa, o programador deve criar uma divisão estática do programa em suas partes concorrentes, então escritas como tarefas. Isso pode ser um processo complicado. Programas em linguagens funcionais podem ser divididos em partes concorrentes dinamicamente pela execução do sistema, tornando o processo altamente adaptável ao sistema de hardware no qual ele está sendo executado. Entender programas concorrentes em linguagens imperativas é muito mais difícil.

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 acreditem que os programas escritos em linguagens funcionais ainda são lentos. Além disso, a maioria dos programadores inicia com linguagens imperativas – assim, muitas vezes os programas funcionais parecem estranhos e difíceis de entender. Para aqueles que estão confortáveis com programação imperativa, a troca para a funcional pode ser difícil e desestimulante. Por outro lado, quem começa com uma linguagem funcional não estranha os programas funcionais.

Existem aqueles, obviamente programadores imperativos, que acreditam que, por estarem de bem com a programação imperativa, ela é de certa forma uma maneira intrinsecamente mais natural de programar. Alguns programadores funcionais se sentem da mesma forma a respeito dos programas funcionais. É claro, ninguém determinou uma maneira efetiva de

* N. de R. 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);}

medir “naturalidade”. Um último pensamento: talvez a proximidade da programação funcional com a matemática, embora resulte em concisão e elegância, pode tornar os programas funcionais menos acessíveis a muitos programadores, especialmente para aqueles que não ficam muito confortáveis com matemática.

RESUMO

Funções matemáticas são mapeamentos nomeados ou não nomeados que usam apenas expressões condicionais e recursão para controlar suas avaliações. Funções complexas podem ser construídas usando formas funcionais, nas quais funções são usadas como parâmetros, valores retornados ou ambos.

Linguagens de programação funcional são modeladas usando funções matemáticas. Em sua forma pura, não usam variáveis ou sentenças de atribuição para produzir resultados; em vez disso, usam aplicações funcionais, expressões condicionais e recursão para o controle da execução e formas funcionais para construir funções complexas. LISP começou como uma linguagem puramente funcional, mas logo teve adicionados recursos de linguagens imperativas de forma a aumentar sua eficiência e facilidade de uso.

A primeira versão de LISP cresceu 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.

A primeira implementação de LISP foi extremamente bem afortunada: 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 primitivas primárias de Scheme incluem funções para construir e separar listas, funções para expressões condicionais e predicados simples para números, símbolos e listas. Scheme inclui algumas operações imperativas, como para modificar um elemento de uma lista.

COMMON LISP é uma grande 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.

ML é uma linguagem de programação funcional de escopo estático e fortemente tipada que usa uma sintaxe mais fortemente relacionada à de uma linguagem imperativa do 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.

Haskell é similar a ML, exceto que todas as expressões em Haskell são avaliadas usando um método preguiçoso, permitindo aos programas lidarem com listas infinitas. Haskell também suporta compreensões de lista, que fornecem uma sintaxe conveniente e familiar para descrever conjuntos.

Apesar de a área primária de aplicação de LISP ser IA, ela tem sido usada de maneira bem-sucedida para diversas áreas de resolução de problemas.

Apesar de poderem existir vantagens no uso de linguagens funcionais puras sobre as linguagens imperativas, sua velocidade de execução mais lenta em máquinas von Neumann previu que elas fossem consideradas por muitos como substitutas das linguagens imperativas.

NOTAS BIBLIOGRÁFICAS

A primeira versão publicada de LISP pode ser encontrada em McCarthy (1960). Uma versão amplamente usada 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 (1984). A linguagem Scheme, com algumas de suas inovações e vantagens, é discutida em Rees e Clinger (1986).

Dybvig (2003) é uma boa fonte de informação a respeito de programação em Scheme. ML é definida em Milner et al. (1990). Ullman (1998) é um livro texto introdutório excelente para ML. A programação em Haskell é introduzida em Thompson (1999).

Uma discussão rigorosa de programação funcional de um modo geral 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 é uma expressão lambda específica?
3. Que tipos de dados eram parte do 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. Quais são os três parâmetros para IF?
7. Quais são as diferenças entre =, EQ?, EQV?, e EQUAL??
8. Quais são as diferenças entre o método de avaliação usado para a forma especial DEFINE de Scheme e aquele usado para suas funções primitivas?
9. Quais são as duas formas de DEFINE?
10. Descreva a sintaxe e a semântica de COND.
11. Por que CAR e CDR são chamadas dessa forma?
12. Se CONS é chamada com dois átomos, digamos A e B, o que é retornado?
13. Descreva a sintaxe e a semântica de LET.
14. Quais são as diferenças entre CONS, LIST e APPEND?
15. Descreva a sintaxe e a semântica de mapcar.
16. O que é recursão em cauda? Por que é importante definir funções que usam recursão para especificar repetição como recursivas em cauda?
17. Por que recursos imperativos foram adicionados à maioria dos dialetos de LISP?
18. De que maneiras COMMON LISP e Scheme são opostas?
19. Que regra de escopo é usada em Scheme? Em COMMON LISP? Em ML? Em Haskell?
20. Quais são as três maneiras pelas quais ML é significativamente diferente de Scheme?

21. O que é inferência de tipos, usada em ML? (Veja o Capítulo 5).
22. As funções ML que lidam com escalares numéricos podem ser genéricas?
23. Quais são os três recursos de Haskell que a tornam significativamente diferente de Scheme?
24. Quais são as duas características de Haskell que a tornam significativamente diferente de ML?
25. O que significa avaliação preguiçosa?
26. O que é uma linguagem de programação estrita?
27. Qual é uma das características das linguagens de programação funcional que tornam sua semântica mais simples do que a de linguagens imperativas?
28. Qual é a falha em usar linhas de código para comparar a produtividade de linguagens funcionais e imperativas?
29. Por que a concorrência pode ser mais fácil com linguagens funcionais do que com linguagens imperativas?

CONJUNTO DE 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 para qualquer linguagem 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 cuidadosamente e compare a dificuldade de escrever programas LISP em seu sistema com a de escrever usando INTERLISP (assumindo que você não usa INTERLISP normalmente).
4. Consulte um livro sobre programação em LISP e determine que argumentos suportam a inclusão do recurso PROG em LISP.
5. Uma linguagem funcional pode usar alguma estrutura de dados além de listas. Por exemplo, poderia usar sequências de símbolos. Que primitivas deveria ter tal linguagem no lugar de CAR, CDR e CONS de Scheme?
6. Crie uma lista de recursos de F# que não estão em ML.
7. Se Scheme fosse uma linguagem funcional pura, ela poderia incluir DISPLAY? Justifique sua resposta.
8. O que a seguinte função em Scheme faz?

```
(define (y s lis)
  (cond
    ((null? lis) '())
    ((equal? s (car lis)) lis)
    (else (y s (cdr lis))))
  ))
```

9. O que a seguinte função em Scheme faz?

```
(define (x lis)
  (cond
    ((null? lis) 0)
    ((not (list? (car lis)))
     (cond
       ((eq? (car lis) nil) (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 deve 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 Scheme que remova todas as instâncias de nível superior de um dado átomo de uma dada lista.
8. Escreva uma função Scheme que remova o último elemento de uma dada lista.
9. Repita o Exercício de Programação 5, exceto que o átomo pode ser ou um átomo ou uma lista.
10. Escreva uma função Scheme que receba dois átomos e uma lista como parâmetros e substitua todas as ocorrências do primeiro átomo dado na lista com o segundo átomo dado, independentemente de quão profundamente o primeiro átomo está 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 para a igualdade estrutural de duas listas dadas. As duas listas são estruturalmente iguais se elas 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 que representam conjuntos.
14. Escreva uma função Scheme com dois parâmetros, um átomo e uma lista, que retorne a lista com todas as ocorrências, independentemente de quão profundas, do dado átomo apagadas. A lista retornada não pode conter nada no lugar dos átomos apagados.

15. Escreva uma função Scheme que receba uma lista como parâmetro e retorne-a com o segundo elemento do nível superior removido. Se a lista não tem dois elementos, a função deve retornar () .
16. Escreva uma função Scheme que recebe uma lista simples de números como seu parâmetro e retorne a lista com os números ordenados de maneira ascendente.
17. Escreva uma função Scheme que recebe uma lista simples como seu parâmetro e retorna uma lista de todas as permutações da lista dada.
18. Escreva o algoritmo de ordenação rápida (quicksort) em Scheme.
19. Reescreva a seguinte função Scheme como uma função recursiva em cauda:

```
(DEFINE (doit n)
  (IF (= n 0)
    0
    (+ n (doit (- n 1)) )
  ) )
```

Capítulo 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 Uma visão geral da programação lógica

16.5 As origens do Prolog

16.6 Os elementos básicos do Prolog

16.7 Deficiências do Prolog

16.8 Aplicações de programação lógica

O objetivo deste capítulo é introduzir 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 o cálculo de predicados pode ser usado para sistemas automáticos de prova de teoremas. Então, apresentamos uma visão geral de programação lógica. A seguir, uma extensa seção introduz o básico 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 seções finais descrevem alguns dos problemas de Prolog como uma linguagem lógica e algumas áreas de aplicação nas quais o Prolog tem sido usado.

16.1 INTRODUÇÃO

O Capítulo 15 discutiu o paradigma de programação funcional, significativamente diferente das metodologias de desenvolvimento de software usadas com as linguagens imperativas. Neste capítulo, descrevemos outra metodologia de programação. Nesse caso, 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 de os procedimentos detalhados para produzi-los.

A programação que usa uma forma de lógica simbólica como uma 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**. Escolhemos descrever a linguagem de programação lógica Prolog, porque ela é a única amplamente usada.

A sintaxe das linguagens de programação lógica é notavelmente diferente da sintaxe das linguagens funcionais e imperativas. A semântica de programas lógicos também tem pouca semelhança com a dos programas em linguagens imperativas. Essas observações devem trazer alguma curiosidade ao leitor acerca da natureza da programação lógica e das linguagens declarativas.

16.2 UMA BREVE INTRODUÇÃO AO CÁLCULO DE PREDICADOS

Antes que possamos 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** pode ser vista como uma sentença lógica que pode ou não ser verdadeira. Ela consiste em objetos e relacionamentos de objetos uns com os outros. 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 a 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 assume verdadeiras.

Existe um forte relacionamento entre lógica formal e matemática. Na verdade, muito da matemática pode ser pensado em termos de lógica. Os axiomas fundamentais de teoria dos números e dos conjuntos formam o conjunto inicial de proposições, os quais se assumem 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 isso ser um pouco impreciso, nos referenciaremos a essa forma como cálculo de predicados). Nas subseções a seguir, apresentamos uma breve visão do cálculo de predicados. Nossa objetivo é montar as bases para uma discussão de programação lógica e da 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 as variáveis em 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 um no qual a verdade da proposição é algo que deve ser determinado. Em outras palavras, as proposições podem ser definidas como fatos ou consultas. As proposições de exemplo 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, símbolos e o significado dos conectores lógicos do cálculo de predicados são:

Nome	Símbolo	Exemplo	Significado
negação	\neg	$\neg a$	a não é verdadeiro
conjunção	\cap	$a \cap b$	a e b são verdadeiros
disjunção	\cup	$a \cup b$	a ou b são verdadeiros
equivalência	\equiv	$a \equiv b$	a é equivalente a b
implicação	\supset	$a \supset b$	a implica em b
	\subset	$a \subset b$	b implica em a

Os seguintes são exemplos de proposições compostas:

$$\begin{aligned} a \cap b &\supset c \\ a \cap \neg b &\supset d \end{aligned}$$

O operador \neg tem a precedência mais alta. Os operadores \cap, \cup , e \equiv todos têm precedência mais alta do que \supset e \subset . Então, o segundo exemplo 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 descritos 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:

$$\begin{aligned} \forall X.(\text{woman}(X) \supset \text{human}(X)) \\ \exists X.(\text{mother}(\text{mary}, X) \wedge \text{male}(X)) \end{aligned}$$

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 do quantificador universal e do 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 do 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 com o 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 lógicos, mas se o cálculo de predicados será usado em um sistema automatizado (computadorizado), é um problema sério. Para simplificar as coisas, uma forma padrão para proposições é desejável. A forma clausal, relativamente simples de proposições, é uma das formas padrão. Sem a perda de generalidade, todas as proposições podem ser expressas em forma clausal. Uma proposição em forma clausal tem a seguinte sintaxe geral:

$$B_1 \cup B_2 \cup \dots \cup B_n \subset A_1 \cap A_2 \cap \dots \cap A_m$$

na qual os As e Bs são termos. O significado dessa proposição em forma clausal é: se todos os As são verdadeiros, então ao menos um B é verdadeiro. A primeira característica de proposições na forma clausal é: quantificadores existenciais não são necessários; quantificadores universais são implícitos no uso de variáveis nas proposições atômicas; e nenhum operador além da conjunção e da disjunção são necessários. 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 algorítmicamente convertidas para a 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 conse-

quência da verdade do antecedente. Como exemplos de proposições na forma clausal, considere:

$$\begin{aligned} \text{likes(bob, trout)} &\subset \text{likes(bob, fish)} \cap \text{fish(trout)} \\ \text{father(louis, al)} \cup \text{father(louis, violet)} &\subset \text{father(al, bob)} \cap \text{mother(violet, bob)} \cap \\ &\quad \text{grandfather(louis, bob)} \end{aligned}$$

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* e *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, que buscam descobrir novos teoremas que podem ser inferidos a partir de axiomas e teoremas conhecidos.

Os primeiros dias da ciência da computação (os anos 1950 e o início dos anos 1960) viram uma boa dose de 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: suponha que existam duas proposições com as formas

$$\begin{aligned} P_1 &\subset P_2 \\ Q_1 &\subset Q_2 \end{aligned}$$

Seu significado é que P_2 implica em P_1 e Q_2 implica em Q_1 . A seguir, suponha que P_1 é idêntica a Q_2 , então poderíamos renomear P_1 e Q_2 como T . Então, poderíamos reescrever as duas proposições como

$$\begin{aligned} T &\subset P_2 \\ Q_1 &\subset T \end{aligned}$$

Agora, como P_2 implica em T e T implica em Q_1 , é logicamente óbvio que P_2 implica em Q_1 , que 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:

$$\begin{aligned} \text{older(joanne, jake)} &\subset \text{mother(joanne, jake)} \\ \text{wiser(joanne, jake)} &\subset \text{older(joanne, jake)} \end{aligned}$$

A partir dessas proposições, as seguintes proposições podem ser construídas usando resolução:

$$\text{wiser(joanne, jake)} \subset \text{mother(joanne, jake)}$$

A mecânica dessa construção de resolução é simples: os termos dos lados esquerdos das duas proposições são unidos por um E para fazer o lado esquerdo da nova proposição. Então a mesma coisa é feita para obter 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 múltiplos 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 tivéssemos

$$\begin{aligned} \text{father(bob, jake)} \cup \text{mother(bob, jake)} &\subset \text{parent(bob, jake)} \\ \text{grandfather(bob, fred)} &\subset \text{father(bob, jake)} \cap \text{father(jake, fred)} \end{aligned}$$

a resolução diz que

$$\begin{aligned} \text{mother(bob, jake)} \cup \text{grandfather(bob, fred)} &\subset \\ \text{parent(bob, jake)} \cap \text{father(jake, fred)} & \end{aligned}$$

que tem todas, exceto uma das proposições atômicas de ambas as proposições originais. A proposição atômica que permitia a operação `father(bob, jake)` no lado direito da primeira e no lado direito da segunda é deixada de fora. Em português, diríamos

- se:* bob é um dos pais de jake implica que bob é ou o pai ou a mãe de jake
e: bob é o pai de jake e jake é o pai de fred implica que bob é o avô de fred
então: se bob é um dos pais de jake e jake é o pai de fred
então: ou bob é a mãe de jake ou bob é o avô de fred

A resolução é mais complexa do que esses simples exemplos ilustram. Em particular, a presença de variáveis nas proposições requer a resolução para encontrar 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 para 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 habilidade de detectar qualquer inconsistência em um conjunto de proposições. Essa propriedade permite que a resolução seja usada para provar teoremas, podendo ser feito da seguinte forma: 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 iniciando como a nova proposição. O teorema é negado de forma que a resolução possa ser usada para provar o teorema achando uma inconsistência. Essa é uma prova por contradição. 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 um objetivo a ser inferido a partir das hipóteses, usando resolução.

Quando proposições são usadas para resolução, apenas um tipo restrito de forma clausal pode ser usado, que simplifica ainda mais o processo de resolução. Os tipos especiais de proposições, chamados de **cláusulas de Horn**, podem ser de apenas duas formas: uma única proposição atômica do lado esquerdo ou um lado esquerdo vazio¹. O lado esquerdo de uma proposição em forma clausal é chamado algumas vezes de *cabeça* (*head*), e as cláusulas de Horn com lados esquerdos vazios são chamadas de *cláusulas de Horn com cabeça*. Cláusulas de Horn com cabeça 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).

¹ As cláusulas de Horn são chamadas assim por causa de Alfred Horn (1951), que estudou cláusulas nesta forma.

16.4 UMA VISÃO GERAL 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 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, 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, assumindo 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. A ação resultante da sentença, então, 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 do 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 em *como*, exatamente, a computação deve ser feita. Em outras palavras, o computador é tratado como um simples dispositivo que obedece ordens. Deve ser informado cada um dos detalhes de tudo o que será computado. Alguns acreditam que essa é a essência da dificuldade de programar computadores.

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 para linguagens de programação lógica é um meio conciso de fornecer 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.

Um exemplo usado comumente para ilustrar a diferença entre sistemas procedurais e não procedurais é a ordenação. Em uma linguagem como C++, a ordenação é feita explicando em um programa C++ todos os detalhes de algum algoritmo de ordenação para um computador que possui um compilador C++. O computador, após traduzir o programa C++ para código de máquina ou algum código intermediário interpretável, segue as instruções e produz a lista ordenada.

Em uma linguagem não procedural, é necessário apenas 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 é 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 expressado como:

$$\begin{aligned} \text{sort}(\text{old_list}, \text{new_list}) &\subset \text{permute}(\text{old_list}, \text{new_list}) \cap \text{sorted}(\text{new_list}) \\ \text{sorted}(\text{list}) &\subset \forall j \text{ such that } 1 \leq j < n, \text{list}(j) \leq \text{list}(j+1) \end{aligned}$$

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 soar como 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. Além disso, a melhor forma para uma linguagem lógica ainda não foi determinada, e bons métodos para criar programas em linguagens de programação lógica para grandes problemas ainda não foram desenvolvidos.

16.5 AS ORIGENS DO PROLOG

Conforme dito 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, e 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, dentre outras coisas.

O desenvolvimento do 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 – Fifth Generation Computing Systems (FGCS; Fuchi, 1981; Moto-oka, 1981). Um dos objetivos primários do projeto era desenvolver máquinas inteligentes, e Prolog foi escolhido como base para esse esforço. O anúncio do FGCS fez surtir um forte interesse súbito 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 acerca da programação lógica e do Prolog, poucas coisas significativas foram descobertas. Isso levou ao declínio no interesse e no uso de Prolog, apesar de ele ainda ter suas aplicações e proponentes.

16.6 OS ELEMENTOS BÁSICOS DO PROLOG

Existem agora inúmeros dialetos de Prolog. Esses podem ser agrupados em diversas categorias: as que cresceram a partir do grupo de Marselha, as que vieram a partir do grupo de Edimburgo e alguns dialetos que têm sido desenvolvidos para microcomputadores, 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, às vezes chamado 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 – Free Software Organization (<http://www.gnu.org>).

16.6.1 Termos

Como programas em outras linguagens, os em Prolog consistem em coleções de sentenças. Existem apenas alguns tipos de sentenças em Prolog, mas elas podem ser complexas. Todas as sentenças em Prolog são construídas 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. 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 uma **instanciação**, que 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 uma 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 pensadas como objetos, permitindo que os fatos sejam definidos em termos de diversos átomos relacionados. Nesse sentido, as estruturas são relações, já que elas 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 de sentenças Prolog começa com aquelas sentenças usadas para construir as hipóteses ou base de dados de informações pré-definidas – 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 assume serem verdadeiras.

Os exemplos a seguir ilustram os tipos de fatos que alguém pode ter 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 elas 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*. O significado mais comum e mais direto, entretanto, é que *bill* é o pai (*father*) 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 com um conhecido teorema na matemática a partir do qual uma conclusão pode ser tirada se o conjunto das condições dadas for satisfeita. 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 elas 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 múltiplos termos separados por operações E lógicas. Em Prolog, a operação E é implicada. As estruturas que especificam proposições atômicas em uma conjunção são separadas por vírgulas, então alguém pode considerar as vírgulas como operadores E. Como um exemplo de uma conjunção, considere:

```
female(shelley), child(shelley).
```

A forma geral da sentença de cláusula de Horn com cabeça em Prolog é
consequente_1 :- expressão_antecedente.

Ela é lida da seguinte forma: “O consequente_1 pode ser concluído se a expressão antecedente for verdadeira ou puder ser tornada verdadeira por alguma instância de suas variáveis”. Por exemplo,

```
ancestor(mary, shelley) :- mother(mary, shelley).
```

afirma que se *mary* é a mãe (*mother*) de *shelley*, então *mary* 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 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).
sibling(X, Y) :- mother(M, X), mother(M, Y),
                 father(F, X), father(F, Y).
```

Essas sentenças dão regras de implicação entre algumas variáveis ou objetos universais. Nesse caso, os objetos universais são *X*, *Y*, *Z*, *M* e *F*. A primeira regra diz que se existem instâncias de *X* e *Y* tal que *mother(X, Y)* é verdadeira, então para essas mesmas instâncias de *X* e *Y*, *parent(X, Y)* é verdadeira.

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 é na forma de uma proposição que queremos que o sistema prove sua veracidade ou falsidade. Em Prolog, essas proposições são chamadas de **objetivos** ou **consultas**. A forma sintática das sentenças de objetivo em Prolog é idêntica a 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 o sistema não foi capaz de prová-lo.

Proposições conjuntivas e proposições com variáveis também são objetivos legais. Quando as variáveis estão presentes, o sistema não apenas avalia a validade do objetivo, mas também identifica as instâncias das variáveis que tornam o objetivo verdadeiro. Por exemplo,

```
father(X, mike).
```

pode ser perguntado. O sistema irá tentar, por unificação, encontrar uma instância de *X* que resulta 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 jeito de distinguir entre as duas. Implementações de Prolog interativas fazem isso por 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 do Prolog

Esta seção examina a resolução de Prolog. O uso eficiente de Prolog requer que o programador saiba precisamente o que o sistema Prolog 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 ou Q 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, \dots, P_n tal que

```
P2 :- P1
P3 :- P2
...
Q :- Pn
```

É claro, o processo pode ser, e frequentemente é, complicado por regras com lados direitos compostos e regras com variáveis. O processo de encontrar os P_i , 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 processo de casamento de proposições, ele é algumas vezes chamado de **casamento**. Em alguns casos, provar um subobjetivo é chamado de **satisfazer** o subobjetivo.

Considere a consulta:

```
man(bob).
```

Essa sentença de objetivo é do tipo mais simples. É relativamente fácil para a resolução determinar se ela é verdadeira ou falsa: o padrão desse objetivo é comparado com os fatos e com as regras na base de dados. Se a base incluir o fato

```
man(bob).
```

a prova é trivial. Se, entretanto, a base de dados contiver o seguinte fato e regra de inferência,

```
father(bob).
man(X) :- father(X).
```

o Prolog precisaria 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, o 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 o 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 abordagens 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 abordagem 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 o objetivo que levem a algum conjunto de fatos originais na base de dados. Essa abordagem 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 abordagem de encadeamento para frente é melhor quando o número de possíveis respostas corretas é grande; nessa situação o encadeamento para trás iria requerer 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 o encadeamento para trás seria mais adequado para uma classe maior de problemas do que o encadeamento para frente.

O seguinte exemplo ilustra a diferença entre o encadeamento para frente e o para trás. Considere a consulta:

```
man(bob) .
```

Assuma 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 casando a primeira proposição com o lado direito da segunda regra (`father(X)`) pela instanciação de X como `bob` e casando o lado esquerdo da segunda proposição para o objetivo. O encadeamento para trás iria primeiro casar o objetivo com o lado esquerdo da segunda proposição (`man(X)`) pela instanciação de X para `bob`. Como seu 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 tiver 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 abordagem de busca primeiro em profundidade porque ela pode ser feita com menos recursos computacionais. A abordagem primeiro em largura é uma busca paralela que pode requerer uma grande quantidade de memória.

O último recurso do mecanismo de resolução do Prolog que deve ser discutido é o rastreamento para trás (*backtracking*). Quando um objetivo com múltiplos subobjetivos está sendo processado e o sistema falha em mostrar a verdade de um de seus subobjetivos, o sistema abandona o subobjetivo que não pode provar. Ele então reconsidera o subobjetivo prévio, se existe 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 requerer uma boa dose de tempo e espaço porque pode ter de encontrar todas as provas possíveis para cada um dos subobjetivos. Essas provas de cada subobjetivo podem não estar organizadas para minimizar o tempo necessário para encontrar aquele que resultará na prova final completa, o que piora o problema.

Para solidificar o seu entendimento sobre rastreamento para trás, considere o seguinte exemplo. Assuma que existe um conjunto de fatos e regras em uma 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 *x* é um dos pais (*parent*) de *shelley*. Como seu primeiro passo, o Prolog encontra o primeiro fato na base de dados com homem (*male*) como seu *functor*. Ele então instancia *x* para o parâmetro do fato encontrado, *mike* digamos. Ele então tenta provar que *parent* (*mike*, *shelley*) é verdadeiro. Se for falso, ele volta para o primeiro subobjetivo, *male* (*x*), e tenta ressatisfazê-lo com alguma instanciação alternativa de *x*. O processo de resolução pode ter de encontrar cada homem 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 mais sobre o processo de resolução.

16.6.6 Aritmética simples

O Prolog suporta variáveis inteiros e aritmética de inteiros. Originalmente, os operadores aritméticos eram *functores*, então a soma de 7 com a variável `x` era formada com

```
+ (7, X)
```

O 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 não pode ter sido instanciada anteriormente. Por exemplo, em

```
A is B / 17 + C.
```

se `B` e `C` são instanciadas, mas `A` não é, essa cláusula fará `A` ser instanciada com o valor da expressão. Quando isso acontece, a cláusula é satisfeita. Se ou `B` ou `C` não for instanciada ou `A` estiver instanciada, a cláusula não é satisfeita e nenhuma instanciação de `A` pode ocorrer. A semântica de uma proposição `is` é consideravelmente diferente de uma sentença de atribuição em uma linguagem imperativa. Essa diferença pode levar a um cenário interessante. Como o operador `is` torna a cláusula na qual ele aparece algo parecido com uma sentença de atribuição, um programador iniciante em Prolog pode ficar tentado 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 requerido, ele pode ser vinculado para algum novo nome).

Prolog não tem sentenças de atribuição no mesmo sentido das linguagens imperativas. Elas simplesmente não são necessárias para a maioria da programação para a qual o Prolog foi projetado. A utilidade das sentenças de atribuição em linguagens imperativas depende da capacidade do programador em 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 simples exemplo do uso de computação numérica em Prolog, considere o seguinte problema: suponha que soubéssemos as velocidades médias de diversos automóveis em uma pista de corrida e o tempo em que eles estão na pista. Essa informação básica pode ser codificada como fatos, e o relacionamento 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 um carro em particular. Por exemplo, a consulta

```
distance(chevy, Chevy_Distance).
```

instanciará Chevy_Distance com o valor 2205. As primeiras duas 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, o Prolog também mostra o nome Chevy_Distance e o seu valor.

Nesse ponto, é instrutivo ter uma visão operacional de como um sistema Prolog produz resultados. O Prolog tem uma estrutura pré-definida 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 em termos de 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 ressatisfazer 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 pensados como subprogramas. Os outros dois eventos são únicos aos 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 está sendo 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, onde o tempo (time) 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

Alguém pode 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. O controle também pode deixar um objetivo de duas maneiras: se o objetivo for bem-sucedido, o controle deixa o objetivo pela porta sair; se o objetivo falhou,

o controle deixa o objetivo pela porta falhar. Um modelo do exemplo é mostrado na Figura 16.1. Nesse exemplo, o controle flui por cada subobjetivo duas vezes. O segundo subobjetivo falha a primeira vez, o que força um retorno pela porta refazer para o primeiro subobjetivo.

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 a 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, similar a estrutura de lista usada por LISP. Listas são sequências de qualquer número de elementos, onde os elementos podem ser átomos, proposições atômicas ou quaisquer outros termos, incluindo 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, onde a cabeça e a cauda correspondem a CAR e CDR em LISP. Isso é similar à notação usada em ML e Haskell.

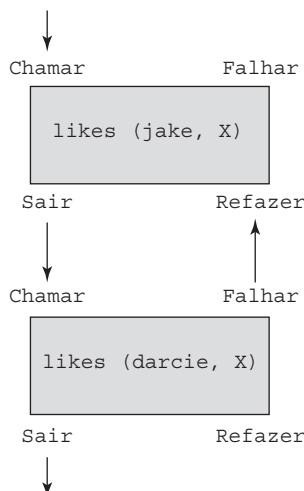


Figura 16.1 O Modelo de fluxo de controle para o objetivo `likes (jake, X), likes (darcie, X)`.

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 (um nome que inventamos). Essa sentença não vincula a lista a uma variável chamada new_list; em vez disso, ela faz o tipo de coisa que a proposição

```
male(jake)
```

faz. Isso 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 reinstantiadas 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 usado 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 falamos anteriormente, a notação de lista que inclui o símbolo | é universal: ela pode especificar uma construção de lista ou uma manipulação 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 um exemplo de tais ope-

rações em Prolog, examinamos uma definição de `append`, relacionada a tal função em LISP. Nesse exemplo, as diferenças e similaridades entre linguagens funcionais e declarativas podem ser vistas. Não precisamos especificar como o Prolog deve construir uma nova lista a partir de listas dadas; em vez disso, precisamos especificar 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 uma maneira similar para produzir a nova lista. No caso do Prolog, a recursão é causada e controlada pelo processo de resolução. Assim como com 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, essa outra 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 o Prolog casará as duas proposições em ordem, começando com a primeira (por causa de 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 o mesmo que o primeiro elemento da primeira lista, porque ambos são chamados `Head`. Sempre que `Head` for instanciado para um valor, todas as ocorrências de `Head` no objetivo são, 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.

Uma maneira de ler a segunda sentença de `append` é: anexar a lista `[Head | List_1]` em qualquer lista `List_2` produz a lista `[Head | List_3]`, mas apenas se a lista `List_3` for formada pela junção de `List_1` com `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 usando a função `append` propriamente

dita; os elementos retirados da primeira lista são adicionados, na ordem inversa, na segunda lista. A inversão é feita pela recursão.

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 casando com a primeira. O efeito disso é retornar como o 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.

As proposições `append` também podem ser usadas 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 de `list_op_2` 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).
```

Como você deve ter sido capaz de 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 abordagens 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 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 ser capazes de determinar se um símbolo está em uma lista. Uma descrição direta em Prolog disso seria

```
member(Element, [Element | _]).
member(Element, [_ | List]) :- member(Element, List).
```

O sublinhado indica uma variável “anônima”, usada para significar que não nos importamos que instânciação ela poderia obter com a unificação. A primeira sentença no exemplo anterior era bem-sucedida se `Element` fosse a cabeça da lista, seja inicialmente ou após diversas recursões por meio do segundo elemento. A segunda sentença era bem-sucedida se `Element` estivesse 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 DO PROLOG

Apesar de Prolog ser uma ferramenta útil, ele não é nem uma linguagem de programação lógica pura, nem perfeita. Esta seção descreve alguns dos problemas com o Prolog.

16.7.1 Controle da ordem de resolução

O 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 do início da base de dados e no final esquerdo de um objetivo, o usuário pode afetar profundamente a eficiência ordenando as sentenças da base de dados para otimizar uma aplicação em particular. Por exemplo, se o usuário sabe que certas regras são muito mais propensas a serem bem-sucedidas do que outras durante uma “execução” em particular, o programa pode ser mais eficiente colocando essas regras primeiro na base de dados.

A execução lenta de programas não é o único resultado negativo da ordenação definida pelo usuário em programas Prolog. É muito fácil escrever sentenças em formas que causam laços infinitos e logo uma falha total do programa. Por exemplo, considere a forma sentencial recursiva:

```
f(X, Y) :- f(Z, Y), g(X, Z).
```

Devido à ordem de avaliação da esquerda para a direita, primeiro em profundidade, do Prolog, independentemente do propósito da sentença, ela causará um laço infinito. Como um exemplo desse tipo de sentença, considere

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

Ao tentar satisfazer o primeiro subobjetivo do lado direito da segunda proposição, o Prolog instancia Z para tornar ancestor verdadeira. Ele então tenta satisfazer esse novo subobjetivo, voltando à definição de ancestor e repetindo o mesmo processo, levando à recursão infinita.

Esse problema em particular é idêntico ao que um analisador sintático descendente recursivo tem com a recursão à esquerda em uma regra gramatical, conforme discutido no Capítulo 3. Como era o caso com regras gramaticais na análise sintática, simplesmente inverter a ordem dos termos no lado direito da proposição de exemplo elimina o problema. O problema com isso é que uma simples mudança no ordenamento dos termos não deveria ser crucial para a corretude do programa. Até porque, a falta da necessidade de o programador se preocupar com a ordem de controle é supostamente uma das vantagens da programação lógica.

Além de permitir que o usuário controle a ordem da base de dados e dos subobjetivos, o Prolog, em outra concessão à eficiência, permite algum controle explícito de rastreamento para trás. Isso é feito com o operador de corte, especificado por um ponto de exclamação (!). O operador de corte é na verdade um objetivo, não um operador. Como um objetivo, ele sempre é bem-sucedido de imediato, mas não pode ser ressatisfatório 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 ressatisfatório 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 falha. Esse objetivo seria usado caso se soubesse que sempre que c falha, o que, neste caso, seria uma perda de tempo tentar ressatisfazer 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 ressatisfazer subobjetivos que presumivelmente não podem resultar em uma prova completa.

Como um 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 um subobjetivo em uma sentença de objetivo com múltiplos subobjetivos, pode existir um problema. O problema é que se member for bem-sucedida, mas o próximo subobjetivo失败, o rastreamento para trás tentará ressatisfazer 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 eventualmente faz todo o objetivo

falhar, apesar de quaisquer tentativas adicionais de ressatisfazer `member`. Por exemplo, considere:

```
dem_candidate(X) :- member(X, democrats), tests(X).
```

Esse objetivo determina se uma pessoa é um democrata e se é uma boa candidata para concorrer a uma determinada posição. O subobjetivo `tests` verifica uma variedade de características do democrata informado para determinar a adequação da pessoa para o cargo. Se o conjunto de democratas não tem duplicatas, não precisamos voltar ao subobjetivo `member` se o subobjetivo `tests` falhar – porque `member` procurará todos os outros democratas e falhará, porque 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 `member`, com o operador de corte como o único elemento, como em

```
member(Element, [Element | _]) :- !.
```

O rastreamento para trás não tentará ressatisfazer `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 a estratégia gerar e testar, o objetivo consiste em subobjetivos que geram soluções em potencial, as quais são então verificadas por subobjetivos posteriores do tipo “teste”. Soluções rejeitadas requerem rastreamento para trás para subobjetivos de “geração”, os quais geram novas soluções em potencial. Como um exemplo de um 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 realiza 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 instancia 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 satisfeita. 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`. O propósito do corte como o último

subobjetivo é simples: ele previne 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 previne tentativas desnecessárias de produzir soluções secundárias.

O uso do operador de corte tem sido comparado ao de desvios incondicionais (*gosos*) em linguagens imperativas (van Emden, 1980). Apesar de ele ser algumas vezes necessário, é possível abusar. De fato, ele é algumas vezes usado para fazer programas lógicos terem um fluxo de controle inspirado em estilos de programação imperativa.

A habilidade modificar o fluxo de controle em um programa Prolog é uma deficiência, porque é diretamente prejudicial a uma das vantagens mais importantes da programação lógica – que os programas não especificam como as soluções devem ser encontradas. Em vez disso, eles simplesmente dizem como a solução deve parecer. Esse projeto torna os programas mais fáceis de escrever e de ler. 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 requeira modificações no fluxo de controle, os programas Prolog geralmente as usam, em sua maioria por questões de eficiência.

16.7.2 A premissa do mundo fechado

A natureza da resolução em Prolog algumas vezes cria resultados enganosos. As únicas verdades, na preocupação de Prolog, são aquelas que podem ser provadas usando a sua base de dados. Qualquer consulta em que existe informação insuficiente na base de dados para prová-la em absoluto é tratada como falsa. O Prolog pode provar que um dado objetivo é verdadeiro, mas não pode prová-lo como falso. Ele simplesmente assume que, como não pode provar um objetivo como 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 opera da mesma maneira. Os suspeitos são inocentes até que se prove ao contrário. Eles não precisam ser provados inocentes. Se um julgamento não pode provar que uma pessoa é culpada, ela é considerada inocente.

O problema com a premissa do mundo fechado está relacionada ao problema da negação, discutido na subseção a seguir.

16.7.3 O problema da negação

Outro problema com Prolog é sua dificuldade com 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. Ele então 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 um irmão (*sibling*) de *Y* apenas se eles não têm os mesmos pais (*parents*) e não são os mesmos. Infelizmente, dizer que eles não são iguais não é direto em Prolog, como iremos discutir. O método mais exato requereria adicionar um fato para cada par de átomos, dizendo que eles não são a mesma coisa. Isso, é claro, torna a base de dados muito grande, o que geralmente é mais informação negativa do que positiva. Por exemplo, as pessoas têm 364 dias de “não aniversário” do 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 é satisfeito nesse caso se a resolução não puder satisfazer o subobjetivo *X = Y*. Logo; se *not* for bem-sucedido, 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, em que NÃO significa que o operando pode ser provado como verdadeiro. Essa não equivalência pode levar a um problema se ocorrer de termos 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 *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]))).`

a seguinte sequência de eventos ocorreria: primeiro o objetivo interno seria bem-sucedido, instanciando `X` para `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 do `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 integral de Prolog é a forma da cláusula de Horn:

$A :- B_1 \cap B_2 \cap \dots \cap B_n$

Se todas as proposições B forem verdadeiras, pode-se concluir que A é verdadeira. Mas independentemente da verdade ou da falsidade de qualquer um dos B s, não se pode provar que A é falso. A partir de lógica positiva, alguém pode concluir apenas lógica negativa. Então, o uso da forma de cláusula de Horn previne 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 realizado. 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 direto 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 com esse processo de ordenação é que ele não tem ideia de como ordenar, além de simplesmente enumerando todas as permutações de uma lista até que aconteça de ser criada uma que tem a lista ordenada – um processo muito lento, de fato.

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 isso. 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 ou para uso externo. Por fim, existe a possibilidade de desenvolvimentos de novas técnicas de inferência que permitirão 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 na forma de tabelas. Consultas em tais tabelas são geralmente descritas em uma linguagem de consulta chamada Linguagem de Consulta Estruturada – Structured Query Language (SQL). 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, ele 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 facilmente e convenien-

temente descritos por regras Prolog. O processo de recuperação é inerente à operação de resolução. As sentenças objetivo de Prolog fornecem as consultas para o SGBDR. A programação lógica é então um casamento 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 pré-definida. Os SGBDRs convencionais não podem deduzir nada 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 desvantagem primária 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 simplesmente levam mais tempo do 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 de 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 da sua base de dados de conhecimento inicial, que é fornecida por um especialista humano, os sistemas especialistas aprendem a partir do processo de serem usados, então suas bases de dados devem ser capazes de crescer dinamicamente. A seguir, um sistema especialista deve incluir a capacidade de interrogar o usuário para obter informação adicional quando ele determinar que tal informação é necessária.

Um dos problemas centrais para o projetista de um sistema especialista é tratar com as inconsistências e incompletudeis inevitáveis da base de dados. A programação lógica parece ser bem adequada para tratar desses problemas. Por exemplo, regras de inferência padrão podem ajudar com o problema da incompletude.

O Prolog pode e tem sido usado para construir sistemas especialistas, podendo facilmente satisfazer suas necessidades básicas. Isso ocorre usando resolução como a base para o processamento de consultas, utilizando sua habilidade de adicionar fatos e regras para fornecer a capacidade de apren-

dizagem, e ainda empregando seu recurso de rastreamento para informar o usuário acerca do “pensamento” por trás de um resultado. Falta ao Prolog a habilidade automática do sistema de solicitar ao usuário informações adicionais quando elas 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. Ele também inclui um segundo interpretador que produz explicações para suas respostas a consultas.

O APES tem sido usado de maneira bem-sucedida para produzir diversos sistemas especialistas, incluindo 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 programação lógica. 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 ser tornados 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 abordagem da programação lógica é usar como base de dados uma coleção de fatos e de regras que definem relacionamentos entre fatos e usar um processo automático de inferência para verificar a validade de novas proposições, assumindo que os fatos e as regras da base de dados sejam verdadeiros. Essa abordagem 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 de Robinson da regra de resolução para

inferência lógica. Prolog foi desenvolvido principalmente por Colmerau e Roussel em Marselha, com alguma ajuda de Kowalski em Edimburgo.

Os programas lógicos devem ser não procedurais. Isso significa que as características da solução são dadas, mas o processo completo 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 atividade primária 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 com o atual estado da programação lógica. Por questões de eficiência, e mesmo para evitar laços de repetição infinitos, os programadores devem algumas vezes ditar informações de fluxo de controle em seus programas. Existem também os problemas da premissa do mundo fechado e da negação.

A programação lógica tem sido usada em algumas áreas diferentes, primariamente 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 coberta em Clocksin e Mellish (2003). A implementação para microcomputadores é descrita em Clark e McCabe (1984).

Hogger (1991) é um livro excelente sobre a ideia geral de programação lógica. É a fonte do material da seção sobre aplicações de programação lógica deste capítulo.

QUESTÕES DE REVISÃO

1. Quais são os três usos primários de lógica simbólica na lógica formal?
2. Quais são as duas partes de um termo composto?
3. Quais são os dois modos nos 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 consequentes?
6. Dê 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 abordagens 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. O que está errado com a 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 pelo Prolog. Por que isso é uma limitação?
21. Explique o problema da negação no Prolog. Por que isso é uma limitação?
22. Explique a conexão entre a prova automática de teoremas e o processo de inferência do 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 o relacionamento entre resolução e unificação em Prolog?

CONJUNTO DE PROBLEMAS

1. Compare o conceito de tipagem de dados em Ada com o de Prolog.
2. Descreva como uma máquina multiprocessada poderia ser usada para implementar resolução. Poderia o Prolog, em sua definição atual, usar esse método?
3. Escreva uma descrição em Prolog de sua árvore genealógica (baseada apenas em fatos), voltando 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. Certifique-se de incluir todos os relacionamentos.
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 de Prolog, aprenda e escreva uma descrição de um problema de ocorrência-verificação. Por que o Prolog permite que esse problema exista em sua implementação?
10. Encontre uma boa fonte de informação acerca da forma normal de Skolem e escreva uma explanação breve, mas clara, sobre ela.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Escreva um programa Prolog que encontre o máximo de uma lista de números.
2. Escreva um programa Prolog que seja bem-sucedido se a interseção de duas listas passadas como parâmetros é vazia.
3. Escreva um programa Prolog que retorne uma lista contendo a união dos elementos de duas listas.
4. Escreva um programa Prolog que retorne o último elemento de uma lista.
5. Escreva um programa Prolog que implemente a ordenação rápida (*quicksort*).

Referências

- ACM. (1979) “Part A: Preliminary Ada Reference Manual” e “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, Março.
- ACM. (1993b) “High Performance FORTRAN Language Specification Part 1.” FORTRAN Forum, Vol. 12, No. 4.
- Aho, A. V., M. S. Lam, R. Sethi e J. D. Ullman. (2006) Compilers: Principles, Techniques, and Tools. 2e, Addison-Wesley, Reading, Estados Unidos.
- Aho, A. V., B. W. Kernighan e P. J. Weinberger. (1988) The AWK Programming Language. Addison-Wesley, Reading, Estados Unidos.
- Andrews, G. R. e 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, Nova York, Estados Unidos.
- ANSI. (1976) American National Standard Programming Language PL/I. ANSI X3.53–1976. American National Standards Institute, Nova York, Estados Unidos.
- ANSI. (1978a) American National Standard Programming Language FORTRAN. ANSI X3.9 – 1978. American National Standards Institute, Nova York, Estados Unidos.
- ANSI. (1978b) American National Standard Programming Language Minimal BASIC. ANSI X3. 60–1978. American National Standards Institute, Nova York, Estados Unidos.
- ANSI. (1985) American National Standard Programming Language COBOL. ANSI X3.23–1985. American National Standards Institute, Nova York, Estados Unidos.
- ANSI. (1989) American National Standard Programming Language C. ANSI X3.159–1989. American National Standards Institute, Nova York, Estados Unidos.
- ANSI. (1992) American National Standard Programming Language FORTRAN 90. ANSI X3. 198–1992. American National Standards Institute, Nova York, Estados Unidos.
- Arden, B. W., B. A. Galler e 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, Estados Unidos.
- Arnold, K., J. Gosling e D. Holmes (2006) The Java (TM) Programming Language, 4e. Addison-Wesley, Reading, Estados Unidos.
- 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, França, 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 e 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, Estados Unidos.
- Ben-Ari, M. (1982) *Principles of Concurrent Programming*. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- Birtwistle, G. M., O.-J. Dahl, B. Myhrhaug e K. Nygaard. (1973) *Simula BEGIN*. Van Nostrand Reinhold, Nova York, Estados Unidos.
- Bobrow, D. G., L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales e D. Moon. (1988) "Common Lisp Object System Specification X3J13 Document 88-002R." *ACM SIGPLAN Notices*, Vol. 17, No. 6, pp. 216–229.
- Bobwin, J. M., L. Bradley, K. Kanda, D. Little e 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. e 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. e D. Korn. (1995) *The New KornShell Command and Programming Language*. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- Booch, G. (1987) *Software Engineering with Ada*, 2e. Benjamin/Cummings, Redwood City, Estados Unidos.
- Bradley, J. C. (1989) *QuickBASIC and QBASIC Using Modular Structures*. W. C. Brown, Dubuque, Estados Unidos.
- Brinch Hansen, P. (1973) *Operating System Principles*. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- 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, Estados Unidos.
- Brinch Hansen, P. (1978) "Distributed Processes: A Concurrent Programming Concept." *Commun. ACM*, Vol. 21, No. 11, pp. 934–941.
- Brown, J. A., S. Pakin e R. P. Polivka. (1988) *APL2 at a Glance*. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- Campione, M., K. Walrath e A. Huml. (2001) *The Java Tutorial*, 3e. Addison-Wesley, Reading, Estados Unidos.
- Cardelli, L., J. Donahue, L. Glassman, M. Jordan, B. Kalsow e G. Nelson. (1989) *Modula-3 Report (revisado)*. Centro de Pesquisas de Sistemas Digitais, Palo Alto, Estados Unidos.
- Chambers, C. e 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.
- Church, A. (1941) *Annals of Mathematics Studies. Volume 6: Calculi of Lambda Conversion*. Princeton Univ. Press, Princeton, NJ. Reimpresso por Klaus Reprint Corporation, Nova York, 1965.

- Clark, K. L. e F. G. McCabe. (1984) Micro-PROLOG: Programming in Logic. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- Clarke, L. A., J. C. Wileden e 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, Estados Unidos.
- Cleaveland, J. C. e R. C. Uzgalis. (1976) Grammars for Programming Languages: What Every Programmer Should Know About Grammar. American Elsevier, Nova York, Estados Unidos.
- Clocksin, W. F. e C. S. Mellish. (2003) Programming in Prolog, 5e. Springer-Verlag, Nova York, Estados Unidos.
- Cohen, J. (1981) "Garbage Collection of Linked Data Structures." ACM Computing Surveys, Vol. 13, No. 3, pp. 341–368.
- Converse, T. e J. Park. (2000) PHP 4 Bible. IDG Books, Nova York, Estados Unidos.
- Conway, M. E. (1963). "Design of a Separable Transition-Diagram Compiler." Commun. ACM, Vol. 6, No. 7, pp. 396–408.
- Conway, R. e R. Constable. (1976) "PL/CS — A Disciplined Subset of PL/I." Relatório Técnico TR76/293. Departamento de Ciência da Computação, Universidade de Cornell, Ithaca, Estados Unidos.
- Cornell University. (1977) PL/C User's Guide, Release 7.6. Departamento de Ciência da Computação, Universidade de Cornell, Ithaca, Estados Unidos.
- Correa, N. (1992) "Empty Categories, Chain Binding, and Parsing." pp. 83–121, Principle-Based Parsing. Eds. R. C. Berwick, S. P. Abney, e C. Tenny. Kluwer Academic Publishers, Boston, Estados Unidos.
- Dahl, O.-J., E. W. Dijkstra e C. A. R. Hoare. (1972) Structured Programming. Academic Press, Nova York, Estados Unidos.
- Dahl, O.-J. e K. Nygaard. (1967) "SIMULA 67 Common Base Proposal." Norwegian Computing Center Document, Oslo, Noruega.
- Deitel, H. M., D. J. Deitel e T. R. Nieto. (2002) Visual BASIC .Net: How to Program, 2e. Prentice-Hall, Inc. Upper Saddle River, Estados Unidos.
- Deliyanni, A. e 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, Estados Unidos.
- Department of Defense. (1961) "COBOL — 1961, Revised Specifications for a Common Business Oriented Language." U.S. Department of Defense, Washington, Estados Unidos.
- Department of Defense. (1962) "COBOL — 1961 EXTENDED, Extended Specifications for a Common Business Oriented Language." U.S. Department of Defense, Washington, Estados Unidos.
- Department of Defense. (1975a) "Requirements for High Order Programming Languages, STRAWMAN." July. U.S. Department of Defense, Washington, Estados Unidos.
- Department of Defense. (1975b) "Requirements for High Order Programming Languages, WOODENMAN." August. U.S. Department of Defense, Washington, Estados Unidos.
- Department of Defense. (1976) "Requirements for High Order Programming Languages, TINMAN." June. U.S. Department of Defense, Washington, Estados Unidos.
- Department of Defense. (1977) "Requirements for High Order Programming Languages, IRONMAN." January. U.S. Department of Defense, Washington, Estados Unidos.
- Department of Defense. (1978) "Requirements for High Order Programming Languages, STEELMAN." June. U.S. Department of Defense, Washington, Estados Unidos.

- Department of Defense. (1980a) "Requirements for High Order Programming Languages, STONEMAN." February. U.S. Department of Defense, Washington, Estados Unidos.
- Department of Defense. (1980b) "Requirements for the Programming Environment for the Common High Order Language, STONEMAN." U.S. Department of Defense, Washington, Estados Unidos.
- DeRemer, F. (1971) "Simple LR(k) Grammars." *Commun. ACM*, Vol. 14, No. 7, pp. 453–460.
- DeRemer, F. e T. Pennello. (1982) "Efficient Computation of LALR(1) Look-Ahead Sets." *ACM TOPLAS*, Vol. 4, No. 4, pp. 615–649.
- Deutsch, L. P. e 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, Nova 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, Estados Unidos.
- Dybvig, R. K. (2003) *The Scheme Programming Language*, 3e. MIT Press, Boston, Estados Unidos.
- Ellis, M. A. e B. Stroustrup (1990) *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Estados Unidos.
- Farber, D. J., R. E. Griswold e 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, e D. L. Stock. (1984) "The Poe Language-Based Editor Project." *ACM SIGPLAN Notices*, Vol. 19, No. 5, pp. 21–29.
- Fischer, C. N. e R. J. LeBlanc. (1977) "UW-Pascal Reference Manual." Madison Academic Computing Center, Madison, Estados Unidos.
- Fischer, C.N. e R. J. LeBlanc. (1980) "Implementation of Runtime Diagnostics in Pascal." *IEEE Transactions on Software Engineering*, SE-6, No. 4, pp. 313–319.
- Fischer, C. N. e R. J. LeBlanc. (1991) *Crafting a Compiler* in C. Benjamin/Cummings, Menlo Park, Estados Unidos.
- Flanagan, D. (2002) *JavaScript: The Definitive Guide*, 4e. O'Reilly Media, Sebastopol, Estados Unidos.
- Floyd, R. W. (1967) "Assigning Meanings to Programs." *Proceedings Symposium Applied Mathematics. Mathematical Aspects of Computer Science* Ed. J. T. Schwartz. American Mathematical Society, Providence, Estados Unidos.
- 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, Estados Unidos.
- Friedman, D. P. e 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, Tóquio, Japão. Reproduzido (1982) por North-Holland Publishing, Amsterdã, Holanda.
- Gehani, N. (1983) Ada: An Advanced Introduction. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- Gilman, L. e A.J. Rose. (1976) APL: An Interactive Approach, 2e. J. Wiley, Nova York, Estados Unidos.
- Goldberg, A. e D. Robson. (1983) Smalltalk-80: The Language and Its Implementation. Addison-Wesley, Reading, Estados Unidos.
- Goldberg, A. e D. Robson. (1989) Smalltalk-80: The Language. Addison-Wesley, Reading, Estados Unidos.
- Goodenough, J. B. (1975) "Exception Handling: Issues and Proposed Notation." Commun. ACM, Vol. 18, No. 12, pp. 683–696.
- Goos, G. e 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, Nova York, Estados Unidos.
- Gordon, M. (1979) The Denotational Description of Programming Languages, An Introduction. Springer-Verlag, Berlim – Nova York, Estados Unidos.
- Graham, P. (1996) ANSI Common LISP. Prentice-Hall, Englewood Cliffs, NJ.
- Gries, D. (1981) The Science of Programming. Springer-Verlag, Nova York, Estados Unidos.
- Griswold, R. E. e M. T. Griswold. (1983) The ICON Programming Language. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- Griswold, R. E., F. Poage e I. P. Polonsky. (1971) The SNOBOL 4 Programming Language, 2e. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- Hammond, P. (1983) APES: A User Manual. Departamento de Computação Relatório 82/9. Imperial College of Science and Technology, Londres, Inglaterra.
- Harbison, S. P. III e G. L. Steele, Jr. (2002) A. C. Reference Manual, 5e, Prentice-Hall, Upper Saddle River, Estados Unidos.
- Henderson, P. (1980) Functional Programming: Application and Implementation. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- 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. Também publicado com um Relatório Técnico STAN-CS-73-403, Departamento de Ciência da Computação da Universidade de Stanford.
- 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. e 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, Londres, Inglaterra.
- Hogger, C. J. (1991) Essentials of Logic Programming. Oxford Science Publications, Oxford, Inglaterra.

- Holt, R. C., G. S. Graham, E. D. Lazowska e M. A. Scott. (1978) Structured Concurrent Programming with Operating Systems Applications. Addison-Wesley, Reading, Estados Unidos.
- Horn, A. (1951) "On Sentences Which Are True of Direct Unions of Algebras." *J. Symbolic Logic*, Vol. 16, pp. 14–21.
- Hudak, P. e J. Fasel. (1992) "A Gentle Introduction to Haskell," *ACM SIGPLAN Notices*, 27(5), Maio 1992, pp. T1–T53.
- Hughes, (1989) "Why Functional Programming Matters", *The Computer Journal*, Vol. 32, No. 2, pp. 98–107.
- Huskey, H. K., R. Love e 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, Nova York, Estados Unidos.
- IBM. (1956) "Programmer's Reference Manual, The FORTRAN Automatic Coding System for the IBM 704 EDPM." IBM Corporation, Nova York, Estados Unidos.
- IBM. (1964) "The New Programming Language." Laboratórios da IBM do Reino Unido.
- Ichbiah, J. D., J. C. Heliard, O. Roubine, J. G. P. Barnes, B. Krieg-Brueckner, e B. A. Wichmann. (1979) "Rationale for the Design of the Ada Programming Language." *ACM SIGPLAN Notices*, Vol. 14, No. 6, Part B.
- IEEE. (1985) "Binary Floating-Point Arithmetic." IEEE Standard 754, IEEE, Nova York, Estados Unidos.
- Ierusalimschy, R. (2006) Programming in Lua, 2e, Lua.org, Rio de Janeiro, Brasil.
- INCITS/ISO/IEC (1997) 1539-1-1997 Information Technology — Programming Languages — FORTRAN Part 1: Base Language. American National Standards Institute, Nova York, Estados Unidos.
- Ingerman, P. Z. (1967). "Panini-Backus Form Suggested." *Commun. ACM*, Vol. 10, No. 3, p. 137.
- Intermetrics. (1993) Programming Language Ada, Draft, Version 4.0. Cambridge, Estados Unidos.
- ISO. (1982) Specification for Programming Language Pascal. ISO7185–1982. International Organization for Standardization, Genebra, Suíça.
- ISO/IEC (1996) 14977:1996, Information Technology — Syntactic Metalanguage — Extended BNF. International Organization for Standardization, Genebra, Suíça.
- ISO. (1998) ISO14882-1, ISO/IEC Standard – Information Technology — Programming Language — C++. International Organization for Standardization, Genebra, Suíça.
- ISO. (1999) ISO/IEC 9899:1999, Programming Language C. American National Standards Institute, Nova York, Estados Unidos.
- ISO/IEC (2002) 1989:2002 Information Technology — Programming Languages — COBOL. American National Standards Institute, Nova York, Estados Unidos.
- Iverson, K. E. (1962) A Programming Language. John Wiley, Nova York, Estados Unidos.
- Jensen, K. e N. Wirth. (1974) Pascal Users Manual and Report. Springer-Verlag, Berlim, Alemanha.
- Johnson, S. C. (1975) "Yacc — Yet Another Compiler Compiler." Relatório de Ciência da Computação 32. AT&T Bell Laboratories, Murray Hill, Estados Unidos.
- Jones, N. D. (ed.) (1980) Semantic-Directed Compiler Generation. Lecture Notes in Computer Science, Vol. 94. Springer-Verlag, Heidelberg, Alemanha.
- Kay, A. (1969) The Reactive Engine. Tese de Doutorado. Universidade de Utah, Setembro.
- Kernighan, B. W. e M. Ritchie. (1978) The C Programming Language. Prentice-Hall, Englewood Cliffs, Estados Unidos.

- Knuth, D. E. (1965) "On the Translation of Languages from Left to Right." *Information & Control*, ol. 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, Estados Unidos.
- 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, Estados Unidos.
- Knuth, D. E. e L. T. Pardo. (1977) "Early Development of Programming Languages." In *Encyclopedia of Computer Science and Technology*, Vol. 7. Dekker, Nova York, pp. 419–493.
- Kowalski, R. A. (1979) *Logic for Problem Solving*. Artificial Intelligence Series, Vol. 7. Elsevier-North Holland, Nova York, Estados Unidos.
- Laning, J. H., Jr. e N. Zierler. (1954) "A Program for Translation of Mathematical Equations for Whirlwind I." Engineering memorandum E-364. Instrumentation Laboratory, Instituto de Tecnologia de Massachusetts (MIT), Cambridge, Estados Unidos.
- Ledgard, H. (1984) *The American Pascal Standard*. Springer-Verlag, Nova York, Estados Unidos.
- Ledgard, H. F. e 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, Estados Unidos.
- Liskov, B., R. L. Atkinson, T. Bloom, J. E. B. Moss, C. Scheffert, R. Scheifler e A. Snyder (1981) "CLU Reference Manual." Springer, Nova York, Estados Unidos.
- Liskov, B. e 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 J. of Research and Development*, Vol. 19, pp. 26–35.
- Lutz, M. e D. Ascher. (2004) *Learning Python*, 2e. O'Reilly Media, Sebastopol, Estados Unidos.
- MacLaren, M. D. (1977) "Exception Handling in PL/I." *ACM SIGPLAN Notices*, Vol. 12, No. 3, pp. 101–104.
- Marcotty, M., H. F. Ledgard e G. V. Bochmann. (1976) "A Sampler of Formal Definitions." *ACM Computing Surveys*, Vol. 8, No. 2, pp. 191–276.
- Mather, D. G. e S. V. Waite (eds.) (1971) *BASIC*. 6e. University Press of New England, Hanover, Estados Unidos.
- 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 e M. Levin. (1965) *LISP 1.5 Programmer's Manual*, 2e. MIT Press, Cambridge, Estados Unidos.
- McCracken, D. (1970) "Whither APL." *Datamation*, Set. 15, pp. 53–57.
- Metcalf, M., J. Reid e M. Cohen. (2004) *Fortran 95/2003 Explained*, 3e. Oxford University Press, Oxford, Inglaterra.
- Meyer, B. (1990) *Introduction to the Theory of Programming Languages*. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- Meyer, B. (1992) *Eiffel: The Language*. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- Microsoft. (1991) *Microsoft Visual Basic Language Reference*. Document DB20664-0491, Redmond, Estados Unidos.

- Milner, R., M. Toft e R. Harper. (1990) *The Definition of Standard ML*. MIT Press, Cambridge, Estados Unidos.
- Milos, D., U. Pleban e G. Loegel. (1984) “Direct Implementation of Compiler Specifications.” *ACM Principles of Programming Languages 1984*, pp. 196–202.
- Mitchell, J. G., W. Maybury e R. Sweet. (1979) *Mesa Language Manual, Version 5.0, CSL-79-3*. Xerox Research Center, Palo Alto, Estados Unidos.
- Moss, C. (1994) *Prolog++: The Power of Object-Oriented and Logic Programming*. Addison-Wesley, Reading, Estados Unidos.
- 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, Tóquio. Republicado (1982) por North-Holland Publishing, Amsterdã, Holanda.
- Naur, P. (ed.) (1960) “Report on the Algorithmic Language ALGOL 60.” *Commun. ACM*, Vol. 3, No. 5, pp. 299–314.
- Newell, A. e 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. e 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, Nova York, Estados Unidos.
- Ousterhout, J. K. (1994) *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Estados Unidos.
- Pagan, F. G. (1981) *Formal Specifications of Programming Languages*. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- Papert, S. (1980) *MindStorms: Children, Computers and Powerful Ideas*. Basic Books, Nova York, Estados Unidos.
- Perlis, A. e 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, Estados Unidos.
- Pratt, T. W. (1984) *Programming Languages: Design and Implementation*, 2e. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- Pratt, T. W. e M. V. Zelkowitz (2001) *Programming Languages: Design and Implementation*, 4e. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- Raymond, E. (2004) *Art of UNIX Programming*. Addison Wesley, Boston, Estados Unidos.
- Rees, J. e W. Clinger. (1986) “Revised Report on the Algorithmic Language Scheme.” *ACM SIGPLAN Notices*, Vol. 21, No. 12, pp. 37–79.
- Remington-Rand. (1952) “UNIVAC Short Code.” Coleção não publicada de notas ditadas. Prefácio por A. B. Tonik, datado de 25 de Outubro de 1955 (1 p.); Prefácio por J.R. Logan, sem data aparentemente de 1952 (1 p.); Exposição preliminary, 1952? (22 pp., onde na qual as páginas de 20 a 22 parecem ser uma substituição posterior); Informação suplementar de Short code, tópico um (7 pp.); Addenda #1, 2, 3, 4 (9 pp.).
- Richards, M. (1969) “BCPL: A Tool for Compiler Writing and Systems Programming.” *Proc. AFIPS SJCC*, Vol. 34, pp. 557–566.
- Robinson, J. A. (1965) “A Machine-Oriented Logic Based on the Resolution Principle.” *Journal of the ACM*, Vol. 12, pp. 23–41.
- Romanovsky, A. e B. Sandén (2001) “Except for Exception Handling,” *Ada Letters*, Vol. 21, No. 3, September 2001, pp. 19–25.
- Roussel, P. (1975) “PROLOG: Manual de Reference et D’utilisation.” Relatório de Pesquisa. Grupo de Inteligência Artificial, Univ. de Aix-Marseille, Luming, França.

- Rubin, F. (1987) “‘GOTO Statement Considered Harmful’ considered harmful” (carta ao editor). *Commun. ACM*, Vol. 30, No. 3, pp. 195–196.
- Rutishauser, H. (1967) *Description of ALGOL 60*. Springer-Verlag, Nova York, Estados Unidos.
- Sammet, J. E. (1969) *Programming Languages: History and Fundamentals*. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- 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, Estados Unidos.
- Schorr, H. e 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. e 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, Nova York, Estados Unidos, pp. 19–46.
- Scott, M. (2000) *Programming Language Pragmatics*, Morgan Kaufman, São Francisco, Estados Unidos.
- Sebesta, R. W. (1991) *VAX Structured Assembly Language Programming*, 2e. Benjamin/Cummings, Redwood City, Estados Unidos.
- Sergot, M. J. (1983) “A Query-the-User Facility for Logic Programming.” In *Integrated Interactive Computer Systems*, P. Degano e E. Sandewall (eds.). North-Holland Publishing, Amsterdã, Holanda.
- Shaw, C. J. (1963) “A Specification of JOVIAL.” *Commun. ACM*, Vol. 6, No. 12, pp. 721–736.
- Sommerville, I. (2005) *Software Engineering*, 7e. Addison-Wesley, Reading, Estados Unidos.
- Steele, G. L., Jr. (1984) *Common LISP*. Digital Press, Burlington, Estados Unidos.
- Stoy, J. E. (1977) *Denotational Semantics: The Scott–Strachey Approach to Programming Language Semantics*. MIT Press, Cambridge, Estados Unidos.
- 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.
- Stroustrup, B. (1986) *The C++ Programming Language*. Addison-Wesley, Reading, Estados Unidos.
- 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, Estados Unidos.
- Stroustrup, B. (1994) *The Design and Evolution of C++*. Addison-Wesley, Reading, Estados Unidos.
- Stroustrup, B. (1997) *The C++ Programming Language*, 3e. Addison-Wesley, Reading, Estados Unidos.
- Sussman, G. J. e 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.
- Tanenbaum, A. S. (2005) *Structured Computer Organization*, 5e. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- Tenenbaum, A. M., Y. Langsam e M. J. Augenstein. (1990) *Data Structures Using C*. Prentice-Hall, Englewood Cliffs, Estados Unidos.
- Teitelbaum, T., e 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, Estados Unidos.
- Thomas, D., C. Fowler e A. Hunt. (2005) Ruby: The Pragmatic Programmers Guide, 2e, The Pragmatic Bookshelf, Raleigh, Estados Unidos.
- Thompson, S. (1999) Haskell: The Craft of Functional Programming, 2e. Addison-Wesley, Reading, Estados Unidos.
- 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, Estados Unidos.
- van Emden, M.H. (1980) "McDermott on Prolog: A Rejoinder." SIGART Newsletter, No. 72, Agosto, pp. 19–20.
- van Wijngaarden, A., B. J. Mailloux, J. E. L. Peck e 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, Fevereiro 1998, pp. 25–30.
- Wall, L., J. Christiansen e J. Orwant. (2000) Programming Perl, 3e. O'Reilly & Associates, Sebastopol, Estados Unidos.
- Warren, D. H. D., L. M. Pereira e F. C. N. Pereira. (1979) "User's Guide to DEC System-10 Prolog." Artigo Ocasional 15. Departamento de Inteligência Artificial, Universidade de Edimburgo, Escócia.
- 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, Estados Unidos.
- Wexelblat, R. L. (ed.) (1981) History of Programming Languages. Academic Press, Nova York, Estados Unidos.
- 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 e 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, Estados Unidos.
- Wilkes, M. V., D. J. Wheeler e S. Gill. (1957) The Preparation of Programs for an Electronic Digital Computer, 2e. Addison-Wesley, Reading, Estados Unidos.
- Wilson, P. R. (2005) "Uniprocessor Garbage Collection Techniques." Disponível em <http://www.cs.utexas.edu/users/oops/papers.html#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, Estados Unidos. Wirth, N. (1975) "On the Design of Programming Languages." Information Processing 74 (Proceedings of IFIP Congress 74). North Holland, Amsterdã, 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, Nova York, Estados Unidos.

- Wirth, N. (1988) “The Programming Language Oberon.” *Software — Practice and Experience*, Vol. 18, No. 7, pp. 671–690.
- Wirth, N. e 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 e A. N. Habermann. (1971) “BLISS: A Language for Systems Programming.” *Commun. ACM*, Vol. 14, No. 12, pp. 780–790.
- Zuse, K. (1972) “Der Plankalkül.” Manuscrito preparado em 1945, publicado em Berichte der Gesellschaft für Mathematik und Datenverarbeitung, No. 63 (Bonn, 1972); Part 3, 285 pp. Tradução para o inglês de todas as páginas, exceto pp. 176–196 no No. 106 (Bonn, 1976), pp. 42–244.

Índice

A

Abordagem fechaduras e chaves (*locks and keys*), 321–323

Abordagem ansiosa (*eager*), 323–324

Abordagem preguiçosa (*lazy*), 323–324

Abordagens para a coleta de lixo, 323–324

Abstração

conceito de, 504–506

suporte para, 32–34

Abstração de dados, 32–34, 40–42, 94–96, 505–508. *Veja também* Tipos de dados abstratos

Abstração de processos, 504–506. *Veja também* Tipos de dados abstratos

Acesso profundo, 494–497

Acesso raso, 496–497

Ações de chamada e de retorno, subprograma, 472 *Veja também* Subprogramas.

Ada, 103–108

atribuições simples e, 361–362 avaliação e, 105–108

bibliotecas de encapsulamento e, 538–540

categorias de exceções e, 650–651

cláusulas accept e, 611–616

compiladores, 36–37

constantes nomeadas e, 259–260

criação de escopo e, 256–257

encapsulamento e, 509–510, 534–535

escopo estático e, 246–248

exemplo de programa e, 106–108

exemplo de tipo de dados abstrato na pilha, 514–516

expressões de modo misto e, 355–356

formato de nomes e, 228–229

literais de enumeração e, 282

matrizes e, 285–292, 294–295

matrizes multidimensionais e, 442–444

métodos de passagem de parâmetros e, 438–440

objetos protegidos e, 619–621

ocultamento de informações e, 509–511, 514

pacotes de corpo e, 510

pacotes de especificação e, 510

palavras reservadas e, 230–231

passagem de mensagens assíncronas e, 621–622

perspectiva histórica e, 103–104

ponteiros em, 317–318

pragma e, 650–652

prioridades e, 618–620

processo de projeto e, 104–106

semáforos binários e, 608–609

sentença case e, 381–383

sentença for e, 390–391, 394

sincronização de competição e, 616–619

sincronização de cooperação e, 615–617

sintaxe de registros e, 304–305

subprogramas genéricos em, 453–456

subtipos, 331–334

suporte a concorrência e, 611–622

término de tarefas, 618–619

tipagem forte e, 328–330

tipos access, 317–318

tipos de dados abstratos em, 509–516

tipos de dados parametrizados em, 527–528

tipos de subfaixa e, 283–284

tipos união e, 309–311

tratamento de continuação após exceção, 647–649

tratamento de exceções em, 645–654

variáveis dinâmicas da pilha e, 243–244

vinculação de exceções a tratadores, 647–649

vinculação dinâmica e, 258–259

visão geral da linguagem e, 105–106

Ada 83, 442–444

Ada 95

características gerais e, 575–577

herança e, 576–578

métodos de passagem de parâmetros e, 439–440

novos recursos e, 107–108

pacotes filho e, 578–580

vinculação dinâmica e, 578–579

Adição unária, 344

Advice Taker, 69–70

Aho, Al, 118–119

- AIMACO, 80–81
- ALGOL 60, 73–80
atribuições simples e, 361–362
avaliação e, 77–80
forma de Backus-Naur e,
139–141
métodos de passagem de parâmetros e, 438–439
perspectiva histórica e, 73–75
PL/I e, 91–92
processo de projeto e, 74–78
programa de exemplo em,
78–80
SIMULA 87 e, 95–96
visão geral do, 77–78
- ALGOL 68, 95–98
estruturas de dados do, 268
ortogonalidade e, 29–30
processo do projeto e, 96–97
visão geral da linguagem e,
96–97
- ALGOL 58, 77–78
recepção do relatório do, 76–77
visão geral do, 75–76
- ALGOL W, 438–439
- Algoritmos concorrentes,
594–595
- Algoritmos de deslocamento-redução**, 215–216
- Algoritmos Escaláveis, 594–595
- Algoritmos LL**, 202–203
- Alocação**, 241
Alocação de objetos, 553–555
Alvos condicionais, 352–353
Ambientes de programação,
51–53
- Ambientes de referenciamento**, 256–258
Ambientes de referenciamento local, 425–428
- Analisadores sintáticos descendentes**, 200–203
- Analisadores sintáticos LR**,
215–222
- Análise léxica, 191–200
casamento de padrões e, 191–193
construção da tabela de símbolos e, 199–200
construção do analisador e,
193–195
diagramas de estado e, 193–195
- implementação em C, 194–199
papel da, 192–193
subprogramas e, 194–195
tokens literais inteiros e,
194–195
- Análise sintática**, 200–204
ascendente, 200, 202–204,
212–222
descendente, 200–203
recursiva descendente, 202–213
classe de gramáticas LL e,
209–213
complexidade da, 203–204
EBNF e, 204–205
introdução à, 200–201
objetivos na, 200
RHS e, 140–141, 204–206,
210–213
- Análise sintática descendente**,
200, 202–204, 212–222
algoritmos de deslocamento-redução e, 215–216
analisadores sintáticos LR e,
215–222
autômatos de pilha e, 215–216
para o problema da análise
sintática, 213–216
- Análise sintática recursiva descendente**, 202–213
classe gramatical LL e, 209–213
processo de, 203–210
- Ancestrais estáticos**, 246–247
Aninhando seletores, 375–379
- Antecedente** (forma clausal), 730
- Apelidos**, 231–233
- APL, 93–95
critérios conflitantes e, 43–44
matrizes e, 292–294
- Aplicações científicas, 23–25
- Aplicações de negócios, 24–25
- APT, 43–44
- Aritmética simples, 742–746
- Armazenamento de dados de instância, 584–585
- Arquitetura**
Computadores, 37–41
Multiprocessadas, 594–597
Von Neumann, 37–39
- Arquitetura de computadores
Única Instrução Múltiplos Dados – Single-Instruction Multiple-Data (SIMD), 594–596
- Arquitetura de von Neumann**,
37–41
- Arquiteturas multiprocessadas,
concorrência e, 594–597
- Arquivos de cabeçalho**,
531–532
- Árvore binária, 33–34
- Árvores de análise sintática**,
46–48, 143–144
- Árvores de análise sintática completamente atribuídas**,
156–157
- ASCII (American Standard Code
for Information Interchange),
273–274
- Assembly**, 535–536
- Asserções**
em Java, 667–668
pós-condição, 170–171
pré-condição, 170–171
semântica axiomática e, 170–172
- Associação Europeia de Fabricantes de Computadores – European Computer Manufacturers Association (ECMA), 120–121
- Association for Computing Machinery (ACM), 76–77, 104–105
- Associatividade**, 148–150, 345–347
- Átomos** (Prolog), 736–737
Átomos simbólicos, 699–701
- Atribuição em modo misto,
365–366
- Atribuições de lista, 365–366
Atribuições simples, 158–159,
361–362
- Atributos**, 155–157
computando valores, 159–161
herdados, 156–157
intrínsecos, 157–158
sintetizados, 156–157
vinculando a variáveis, 233–235
- Automatically Programmed Tools (APT), 43–44
- Autômato de pilha** (PDA),
215–216
- Autômato finito**, 193–194
Avaliação. *Veja também* Critérios de avaliação de linguagens

- Ada, 105–108
 ALGOL 60, 77–80
 BASIC, 85–86–87
 C, 100–102
 C#, 126–128, 632–633
 C++, 112–113
 COBOL, 81–85
 Fortran, 66–69
 gramáticas de atributos, 160–162
 Java, 116–119
 LISP, 71–73
 matrizes, 294–295
 monitores e, 607–609
 Pascal, 98–100
 PL/I, 92
 Prolog, 103–104
 registros, 306–308
 semântica axiomática, 183–184
 semântica denotacional, 169–170
 semântica operacional, 163–165
 Smalltalk, 109–111
 suporte a concorrência em Ada, 622
 suporte a concorrência em Java, 630
 tipos cadeia de caracteres, 277–278
 tipos de enumeração, 282–284
 tipos de subfaixa, 284
 tipos ponteiro, 320–322
 tipos união, 311–312
 tratamento de exceções em Ada, 653–654
 tratamento de exceções em C++, 658–659
 tratamento de exceções em Java, 668–670
- Avaliação de curto circuito**, 359–362
Avaliação preguiçosa, 715–718
 awk, 118–119
Axiomas, 172–173
- B**
 B, como linguagem não tipada, 100–101
 Babbage, Charles, 60–61
Backtracking, 741–742
- Backus, John, 40–41, 62–65, 139–141, 681–683
BASIC, 85–89
 avaliação e, 85–87
 declarações implícitas e, 234–235
 formato de identificadores e, 30–32
 processo do projeto e, 85–86
 programa de exemplo em, 87
 visão geral da linguagem e, 85–86
BASIC Mínimo, 85–86
BASIC-PLUS, 85–86
 Bauer, Fritz, 74–75
 BCPL, como linguagem não tipada, 100–101
Bem definidos, 37–38
Bibliotecas de vinculação dinâmica (DLL), 535–536
Binary coded decimal (BCD), 272–273
 BLISS, 25–26
Blocos
 características de, 247–249
 subprogramas e, 492–494
 Boletim do ALGOL, 76–78
 Borland J Builder, 52–53
Busca primeiro em largura (*breadth first*), 741–742
Busca primeiro em profundidade (*depth first*), 741–742
 Byron, Augusta Ada, 104–105
Byte codes, 49–51
- C**
C, 99–102
 aninhamento de sentenças de seleção e, 376–378
 apelidos e, 232–233
 avaliação e, 100–102
 critérios conflitantes e, 43–44
 dados de cadeias e, 274–276
 deficiências de segurança em, 364–365
 encapsulamento em, 531–534
 equivalência de tipos e, 333–334
 forma e significado e, 31–32
 matrizes e, 288–292
 matrizes multidimensionais e, 442–444
- métodos de passagem de parâmetros e, 437–439
 ortogonalidade e, 30–31
 perspectiva histórica e, 99–101
 ponteiros em, 317–320
 programa de exemplo em, 100–102
 sentença *switch* e, 380–382
 tipagem forte e, 329–330
 tipos enumeração e, 281
 variáveis estáticas e, 242–243
 variáveis globais e, 249–250
C#, 124–128
 aninhamento de sentenças de seleção e, 376–378
 avaliação e, 126–128
 características gerais e, 573–574
 classes aninhadas e, 575
 concorrência e, 630–633
 definições de registros e, 304–305
 encapsulamento em, 535–536
 formato de nomes e, 228–230
 herança e, 573–575
 matrizes e, 289–290
 matrizes multidimensionais e, 445–446
 métodos de passagem de parâmetros e, 439–440
 operações básicas de linhas de execução, 630–632
 ordem de declaração e, 248–250
 parâmetros e, 419–420
 processo do projeto e, 125–126
 programa de exemplo em, 126–128
 programação orientada a objetos e, 573–575
 sentença *switch* e, 381–382
 sincronizando linhas de execução, 631–632
 subprogramas genéricos em, 459–461
 tipagem forte e, 329–330
 tipos de dados abstratos em, 522–524
 tipos de dados parametrizados em, 530–531
 tipos enumeração e, 279–284
 variáveis dinâmicas do monte explícitas e, 244–245

- vinculação dinâmica e, 574–575
 visão geral da linguagem e, 125–127
C++, 110–114
 abstração de dados e, 33–34
 aninhamento de sentenças de seleção e, 376–378
 apelidos e, 232–233
 avaliação e, 112–113
 breve histórico e, 512
 características gerais e, 558–559
 constantes nomeadas e, 259–260
 construtores e, 518
 continuação após tratamento de exceção, 656
 críticas ao, 513
 dados de cadeias e, 274–276
 deficiências de segurança e, 364–365
 destrutores e, 518
 diferenças com Java, 114–116
 encapsulamento e, 516–518, 533–535
 equivalência de tipos e, 333–334
 espaços de nomes, 536–538
 formato de nomes e, 228–230
 funções membro e, 516–518
 herança e, 558–566
 matrizes e, 288–292
 matrizes multidimensionais e, 442–444
 membros de dados e, 516–517
 métodos de passagem de parâmetros e, 438–439
 ocultamento de informações e, 518
 onipresença de, 512
 ordem de declaração e, 248–249
 ortogonalidade e, 30–31
 parâmetros padrão e, 419
 ponteiros em, 317–320
 processo do projeto e, 110–112
 programação orientada a objetos e, 558–570
 relacionamento com Delphi, 113–114
 relacionamento com Eiffel, 112–114
 Smalltalk e, 568–570
 subprogramas genéricos e, 455–458
 tipagem forte e, 329–330
 tipos de dados abstratos em, 516–521
 tipos de dados parametrizados em, 528–530
 tipos de referência e, 319–321
 tratamento de exceções em, 653–659
 variáveis dinâmicas da pilha e, 242–244
 variáveis dinâmicas do monte explícitas e, 243–244
 variáveis estáticas e, 242–243
 variáveis globais e, 249–250
 vinculação dinâmica e, 258–259, 566–569
 vinculando exceções a tratadores, 655–656
 visão geral da linguagem e, 111–113
C99, 248–250
Cabeçalho de subprograma, 415
Cadeias de tamanho dinâmico, 277–280
Cadeia de tamanho estático, 277–280
Cadeias de tamanho limitado, 277–280
 Cálculo de predicados
 introdução ao, 727–731
 provando teoremas e, 731–734
Cálculo de predicados de primeira ordem, 728
 Cambridge Polonesa, 688–689
 Caml, 712–713
Campos, 301, 304
Característica de vivacidade, 600–602
Casamento (Prolog), 741–742
 Casamento de padrões
 análise léxica e, 191–193
 cadeias de caracteres e, 274–275
Casts sem conversão, 329–330
 Categorias de linguagens, 41–44
 CBL, 80–81
 Celes, Waldemar, 124–125
 Célula de memória, 232–233
chain_offset, 485
Chamada a subprograma, 415
 Chamadas a métodos, 584–587
Chaves, 298–299
 Chomsky, Noam, 139–140
 Church, Alonzo, 683–684
Ciclo obter-executar, 38–39
Classe aninhada local, 573–574
Classe derivada, 547–548, 559, 562
Classe gramatical LL, 209–213
Classe pai, 547–548
Classe StringBuffer, 275–276
Classes, 547–548, 559, 562
 abstratas, 549–550, 567–568, 571–572, 574–575
 aninhadas, 554–555, 563, 572–575, 578–580
Classes aninhadas
 C# e, 575
 Java e, 572–574
 linguagens orientadas a objetos e, 554–555
Classes internas, 572–574
 Classes *wrapper*, 550–551
 Cláusula `else-if`, 385–387
 Cláusula `finally` (Java), 666–667
 cláusula `public`, 518
 Cláusula `throw`, C++, 625–626
 Cláusula `throw`, Java, 663
 Cláusulas *Accept* (Ada), 611–616
Cláusulas de Horn, 733–734.
Veja também Prolog
Clientes, 506–507
 COBOL, 80–85
 avaliação e, 81–85
 declaração de registros e, 304–305
 estruturas de dados do, 268
 FLOW-MATIC e, 80–81
 movendo registros e, 306–307
 palavras reservadas e, 230–231
 perspectiva histórica e, 80–81
 processo do projeto e, 80–82
 programa de exemplo em, 82–85
 referências a campos e, 305–306
 COBOL 60, 91–92
Coerção, 326–327
 Coerção em expressões, 354–358
 Coerções de `int` para `float`, 116
 Colmerauer, Alain, 102–103, 735–736
 Comandos, em guarda, 403–406, 610
 Comitê Short Range, 81–82
 Common Gateway Interface (CGI), 119–121

- Common Language Runtime (CLR), 631–632
COMMON LISP, 72–74, 708–709
Communicating Sequential Processes (CSP), 613–614
Communications of the ACM (CACM), 76–77, 682–683
 Compartilhamento de tempo, 85–89
 Compilação, 45–49
 Compiladores Just-in-time (JIT), 191–192
 Complexidade da análise sintática, 203–204
Composição de funções, 684–685
Compreensões de lista, 290–291
 Computador BINAC, 61–62
 Computador PDP-11, 365–366
 Computador UNIVAC I, 61–62
 Computador Whirlwind, 63–64
 Computadores de grande porte da IBM, 28–30
 Computadores de Múltiplas Instruções e Múltiplos Dados (MIMD), 595–596
 Computadores virtuais, 45–46
 Computadorização de registros de negócios, 80–85
COMTRAN, 80–81
 Concatenação, 291–292
 Concorrência
 arquiteturas multiprocessadas e, 594–597
 categorias de, 596–598
 concorrência física, 596–597
 concorrência lógica, 596–597
 concorrência no nível de sentença, 594–595, 632–635
 concorrência no nível de subprograma, 597–602
 concorrência no nível de unidade, 594–595
 concorrência oculta, 595–596
 linhas de execução em C# e, 630–633
 linhas de execução em Java e, 622–630
 monitores e, 606–609
 motivações para usar, 597–598
 passagem de mensagens e, 608–610
 projeto de linguagem para, 601–602
 semáforos e, 601–607
 suporte de Ada para, 611–622
 usos de, 593
 Concorrência no nível de subprogramas, 597–602
 conceitos fundamentais de, 597–602
 projeto de linguagem e, 601–602
 questões de projeto e, 601–602
 sincronização e, 598–600
 tarefas e, 597–599
Condições de corrida, 599–600
 Conferência Internacional em Processamento de Informação, 76–77
 Confiabilidade, 33–38
 apelidos e, 34–36
 legibilidade e facilidade de escrita e, 35–36
 tratamento de exceções e, 34–35
 verificação de tipos e, 34–35
Conjunções, 738–739
Consequente (forma clausal), 730
 Consórcio Unicode, 273–274
 Constantes
 manifestas, 258–259
 nomeadas, 258–260
Constantes de enumeração, 279–280
Construção de iteração, 387–388
Construções de encapsulamento, 530–536
 em C, 531–534
 em C++, 516–518, 533–535
 em montagens C#, 535–536
 em pacotes Ada, 509–510, 534–535
 subprogramas aninhados e, 531–532
Construções de seleção múltipla, 378–387
 exemplos de seleção múltipla, 379–384
 implementando, 384–385
 questões de projeto e, 378–380
 usando `if`, 385–387
 Construções orientadas a objetos
 armazenamento de dados de instância e, 584–585
 implementação e, 583–587
 registros de instâncias de classe e, 584–587
 tabelas de métodos virtuais e, 584–585
 vinculação dinâmica de chamadas a métodos aos métodos, 584–587
Construtores, 518
Consultas (Prolog), 739–740
Contador de programa, 38–39
Contadores de referências, 323–324
Contagem de iteração, 389
Contagem de referências adiada, 323–324
 Contêiner para **Servlets**, 128–129
Continuação, tratamento de exceções e, 644–645
 Controle da ordem de resolução (Prolog), 750–754
 Controle de continuação, 644–645
Conversão de alargamento, 357–359
Conversão de estreitamento, 357–358
Conversões, 356–357
 Conversões de tipos, 353–358
 coerção em expressões e, 354–358
 conversão de tipos explícita, 356–357
 erros em expressões e, 356–358
 Cooper, Alan, 88–89
 Cooper, Jack, 104–105
 Corpo
 de construção iterativa, 387–388
 no cabeçalho do pacote, 509–510
 Corretude
 parcial, 180–181
Corretude total, semântica axiomática e, 180–181
Corrotinas, 95–96, 462–465
 Corte, Prolog, 751–752

Critérios de avaliação de linguagens, 25–38
 boa definição, 37–38
 confiabilidade, 33–38
 custo, 35–38
 facilidade de escrita, 32–34
 generalidade, 37–38
 legibilidade, 26–32
 portabilidade, 36–38
 Currie, Malcolm, 104–105
 Custo, 35–38

D

Dados de ponto flutuante, 90, 271–272, 505–507
 Dahl, Ole-Johan, 94–96
 de Figueiredo, Luis Henrique, 124–125
Decimal
 tipos de dados, 272–274
Declaração explícita, 234–235
Declarações de importação, 538–539
 Declarações implícitas, 234–236
Decorando a árvore de análise sintática, 159–160
 Delphi, 113–114
 Departamento de Defesa, 82–83, 103–104
 Derivação privada, 562–566
Derivações, 141–143
Derivações mais à esquerda, 142–143
 Descendentes do ALGOL, 97–102
Descriptor de tarefa, 602–603
 Descriptor em tempo de compilação, 295–296
Descritores, 269–270
Deslocamento local (local_offset), 480–482
Desreferenciamento de ponteiros, 314–315
Destruidores, 518
 Desvio, incondicional, 402–404
 Detecção de erros, 269–270
 Detecção de erros de faixas de índices, 641
Diagrama de estados, 193–195
 Diagrama de transições, 193–194
Dicionários, 122–123
 Dijkstra, Edsger, 400–401, 602–603

Dinâmicas do monte implícitas, 96–97
Dispersões, 119–120, 299–300
Display, 492–493
Domínio semântico, 164–165
Domínio sintático, 164–165
 Domínios de programação
 aplicações científicas, 23–25
 aplicações de negócios, 24–25
 inteligência artificial, 24–25
 programação de sistemas, 24–26
 software para a Web, 25–26
Dynabook, 108–109

E

Edwards, Daniel J., 688–689
 Efeitos colaterais funcionais, 350–351, 460–461
 Eiffel, 112–114
Elaboração, 242–243
 Empacotamento, 570–571
Encadeamento para frente, 741–742
Encadeamento para trás, 741–742
Encadeamentos de chamadas, 480–481
Encadeamentos dinâmicos, 480–481
Encadeamentos estáticos, 485–493
 Endereçamento absoluto, 62–63
 Endereço de uma variável, 231–233
 Epílogo da ligação de subprogramas, 473–475
 Equipe de projeto da Cii Honeywell/Bull, 104–105
 Equivalência de tipos, 329–334
Equivalência de tipos estrutural, 330–331
Equivalência de tipos por nome, 330–331
Erro de tipo, 326–327
 Erros em expressões, 356–358
Escopo, 245–255
 blocos e, 247–249
 escopo estático, 245–248, 252–253
 escopo global, 249–253
 ordem de declaração e, 248–250
 tempo de vida e, 254–256

Escopo de pacote, 537–538
Escopo dinâmico, 252–255
 acesso profundo e, 494–497
 acesso raso e, 496–497
 LISP e, 689–690
 subprogramas e, 494–497
Escutadores de eventos, 671–672
Estrutura (Prolog), 736–737
Estrutura de controle, 373–374
 Estruturas de lista (Prolog), 745–751
Etiquetas, 309–310
Exceções, 357–358
Exceções levantadas, 641
 Exceções verificadas, 664
 Exclusividade de objetos, 550–551
 Expressão de controle, sentenças de seleção de dois caminhos e, 374–375
 Expressividade, 33–34
Expressões
 booleanas, 129–130, 358–360
 condicionais, 349
 de controle, 368
 de modo misto, 354–357
 gramática não ambígua para, 147
 lambda, 676
 ordem de avaliação, 343–349
 relacionais, 357–359
 semântica denotacional e, 167–169
 versões BNF e EBNF e, 153–155
Expressões Aritméticas
 associatividade e, 345–347
 efeitos colaterais e, 350–353
 em Prolog, 742–746
 expressões condicionais e, 349
 expressões Ruby e, 348
 ordem de avaliação de operadores, 343–349
 ordem de avaliação de operandos, 349–353
 parênteses e, 348
 precedência e, 343–345
 propósito das, 343
 transparência referencial e, 351–353
 Expressões de seleção múltipla, 382–384

- Expressões lambda**, 683–684, 691
- Expressões regulares**, 276–277
- Expressões-S, 688–689
- F**
- Facilidade de escrita, 32–36
- Faixa**, tipo ponto-flutuante e, 271–272
- Farber, D.J., 94–95
- Fatbars*, 403–404
- Fatias** de uma matriz, 293–295
- Fatoração à esquerda**, 212–213
- Fila de tarefas prontas**, 600–601
- Finalização, 645–646
- Flex, 108–109
- FLOW-MATIC, 80–81
- Flynn, Michael, 594–595
- Forma clausal**, 730–731
- Forma de Backus-Naur** (BNF), 139–141
- BNF estendida, 152–155
 - formalismo de descrição de sintaxe e, 191–192
 - introdução a, 76–77
 - semântica estática e, 155–156
- Forma estendida de Backus-Naur (EBNF), 152–155, 204–205
- Forma funcional aplicar a todos, 684–685, 706–707
- Forma sentencial**, 142–143
- Formas funcionais**, 684–685, 705–707
- Formato de nomes, 228–230
- Formatos de identificadores, 30–32
- Fortran
- atribuições simples e, 361–362
 - avaliação e, 66–69
 - dados de cadeias e, 275–276
 - declarações implícitas e, 234–236
 - desenvolvimento e, 64–67
 - exemplo de, 67–69
 - Fortran de alto desempenho, 632–635
 - matrizes e, 286–287
 - matrizes multidimensionais e, 444–446
 - palavras-chave e, 229–230
- perspectiva histórica e, 63–65
- processo do projeto e, 64–65
- sucesso do, 67–68
- Fortran 90, 110–111, 66–67
- Fortran 95
- declarações de registros e, 304–305
 - expressões constantes e, 258–259
 - formato de nomes e, 228–229
 - matrizes e, 289–290, 292–295
 - mudanças feitas no, 66–67
 - palavras especiais e, 31–32
 - sentença DO e, 389–391
 - tipagem forte e, 328–329
 - variáveis dinâmicas da pilha e, 243–244
- Fortran 77, 33–34, 66–67
- Fortran de Alto Desempenho (HPF), 632–635
- Fortran I, 64–66, 357–358
- Fortran II, 65–66
- Fortran IV, 65–67
- Fortran List Processing Language (FLPL), 69–71
- Framework* de desenvolvimento para Web Rails, 120–121
- Frases**, em derivações, 214–216
- Frases simples**, 214–216
- Função de computação de atributos**, 156–157
- Função virtual pura**, 567–568
- Funções, questões de projeto para, 460–462
- Funções de lista (Scheme), 696–698
- Funções de predicado numéricas (Scheme), 693–694
- Funções matemáticas, 682–685
- características das, 682–683
 - formas funcionais, 684–685
 - funções simples, 683–684
- Funções membro**, 516–518, 559, 562
- Funções numéricas primitivas (Scheme), 691
- Funções predicado**, 156–157, 699–701
- Functores**, 728
- G**
- GAMM, 74–75
- Gargalo de von Neumann**, 48–49
- Genealogia, linguagens de programação de alto nível, 58–59
- Generalidade**, 37–38
- Geração**, 138–139
- Gerador**, 715–716
- Geradores de linguagens, 138–140
- Gerar e testar** (Prolog), 752–753
- Gerenciamento do monte, 322–327
- células de tamanho único, 322–326
 - células de tamanho variável, 325–327
- Glennie, Alick E., 63–64
- GNOME**, 52–53
- Gosling, James, 114–115, 660–661
- GPSS, 43–44
- Gramática não ambígua, 146–147, 149–152
- Gramáticas**, 140–143
- ambíguas, 144–146
 - árvore de análise sintática e, 146
 - gramáticas de atributos, 154–162
 - para expressões simples, 146–147
 - para sentenças de atribuição simples, 143
 - para uma linguagem pequena, 142–143
- Gramáticas de atributos**, 154–162
- atributos intrínsecos, 157–158
 - avaliação e, 160–162
 - computando valores de atributos, 159–161
 - conceitos básicos e, 155–157
 - definidas, 156–157
 - exemplos de, 157–160
 - nota histórica e, 155–156
 - para sentenças de atribuição simples, 158–159
 - semântica estática e, 155–156
- Gramáticas de van Wijngaarden, 96–97
- Gramáticas livres de contexto**, 139–141
- Griswold, R.E., 94–95
- Grupo de usuários GUIDE, 90
- Guardas**, 602–603

H

Hansen, Per Brinch, 607–610
 Haskell, 123–124, 712–718
 Hejlsberg, Anders, 113–114, 125–126
Herança
 Ada e, 576–578
 C# e, 573–575
 C++ e, 558–570
 compartilhada, 552–553
 diamante, 552–554
 Java e, 571–573
 múltipla, 548–549, 551–553
 programação orientada a objetos e, 547–549
 Ruby e, 583–584
 Smalltalk e, 557
Herança simples, 548–549, 551–553
 High-Order Language Working Group (HOLWG), 104–105
Hipóteses, 733–734
 Hoare, C. A. R. (Tony), 97–98
 Hopper, Grace, 62–63, 80–81
HTML
 Java Server Pages e, 128–130
 JavaScript e, 120–121
 PHP e, 122–123

I

IAL, 75–76
 IBM, 76–79, 90–92, 94–95
 IBM 704, 63–65
 Ichbiah, Jean, 104–105
 Identificadores, 137
 Ierusalimschy, Roberto, 124–125, 302–303
Imagem executável, 48–49
Impasse, 601–602
Implementação de Compiladores, 45–49
 Indicativos, 96–97
Índices, matrizes e, 285–287
 Inferência de tipo, 240–241
Infix (operadores binários), 343
 Influências no projeto de linguagens, 37–42
 arquitetura de computadores e, 37–41
 metodologias de projeto de programas e, 40–42
 Information Processing Language (IPL), 68–69

Inicialização, 259–260
Instância de registro de ativação, 474–475
Instanciações, 732, 736–737
Inteiros, 229–231, 270–272
 Inteligência Artificial, 24–25, 68–70
Interface, 571–572
Interface de mensagem, 547–548
 Interface gráficas com o usuário (GUI)
 o pacote Java Swing e, 670–672
 tratamento de eventos e, 668–670
 UNIX e, 52–53
 International Standards Organization (ISO), 100–101, 111–112, 120–121
 Interpretação pura, 48–51
 Invariantes de laço, 177–181
 IT, 73–74
 Iteração baseada em estruturas de dados, 399–403
Iteradores, 399–400
 Iteradores pré-definidos, 400–401
 Iverson, Kenneth E., 94–95

J

Java, 113–119
 abstração de dados e, 33–34
 asserções e, 667–668
 avaliação e, 116–119
 características gerais e, 569–572
 classes aninhadas e, 572–574
 cláusula finally, 666–667
 compilador/interpretador gratuitos, 36–37
 componentes de GUI e, 670–672
 constantes nomeadas e, 258–259
 critérios conflitantes e, 43–44
 dados de cadeias e, 275–276
 definições de registros e, 304–305
 diferenças entre C++ e, 114–116
 escolhas de projeto de exceções e, 663–664
 Java 5.0, 530–531
 Java Server Pages (*JSP*), 128–130
 Java Server Pages Standard Tag Library (JSTL), 42–43, 128–130
 formato de nomes e, 228–230
 herança e, 571–573
 ímpeto inicial para, 114–115
 matrizes e, 288–290
 matrizes multidimensionais e, 445–446
 métodos de passagem de parâmetros e, 438–439
 modelo de eventos e, 671–675
 nascimento de, 660–661
 nomeação de encapsulamento e, 537–539
 ordem de declaração e, 248–250
 ortogonalidade e, 30–31
 pacote Swing, 670–672
 portabilidade e, 116
 processo do projeto e, 113–115
 programa de exemplo em, 117–119
 programação orientada a objetos e, 569–574
 sentença switch e, 381–382
 sentenças de iteração definidas pelo usuário e, 400–402
 sentenças while em, 136–137
 subprogramas genéricos em, 458–460
 tipagem forte e, 329–330
 tipos de dados abstratos em, 521–522
 tipos enumeração e, 282
 tratamento de exceções em, 659–668
 variáveis de referência e, 320–321
 variáveis dinâmicas da pilha e, 242–243
 variáveis dinâmicas do monte explícitas e, 244–245
 vinculação dinâmica e, 259–260, 572–573
 vinculando exceções a tratadores, 662–663
 visão geral da linguagem e, 114–116

- JavaScript**
 categorias e, 42–43
 dados de cadeias e, 275–277
 documentos HTML e, 120–121
 expressões relacionais e, 357–358
 interpretação pura e, 49–51
 matrizes e, 289–290
 origens e características e, 120–121
 programa de exemplo em, 121
 vinculação dinâmica de tipos e, 235–237
- JScript.NET, 120–121
- K**
- Kay, Alan, 108–109
 Kemeny, John, 85–86
 Knuth, Donald, 400–401
 Korn, David, 118–119
 Kowalski, Robert, 735–736
 Kurtz, Thomas, 85
- L**
- Laboratórios Bell, 113–114, 118–119
Laço while, 176–178
Laços, 387–388
Laços controlados logicamente, 396–399
 exemplos de, 396–399
 questões de projeto e, 396–397
Laços controlados por contador, 388–397
 questões de projeto e, 388–389
 sentença `for` de Ada, 390–391, 394
 sentença `for` de Python, 395–397
 sentenças `do` em Fortran 95, 389–391
 sentenças `for` das linguagens baseadas em C, 391, 394–396
 Laços lógicos com pré-teste
 semântica axiomática e, 176–181
 semântica denotacional e, 169–170
Lado direito (RHS), 140–141, 204–206, 210–213
Lado esquerdo (LHS), 140–141
- Lápides**, 321–322
Legibilidade, 26–32, 35–36
 ortogonalidade, 28–31
 projeto de sintaxe e, 30–32
 simplicidade geral e, 27–29
 tipos de dados e, 30–31
- Lerdorf, Rasmus, 122–123, 238–239
- Lexemas**, 137, 192–193
- Liberação**, 241
Liberação de objetos, 553–555
- Ligação de subprogramas**, 472, 476–477
- Ligação e carga**, 48–49
- Ligações dinâmicas**, 476–477
- Ligações estáticas**, 485
Linguagem Algorítmica Internacional (IAL), 75–76
Linguagem de marcação extensível (XML), 128–130
Linguagem de programação estrita, 715–716
Linguagem de programação não estrita, 715–716
- Linguagem estruturada em blocos**, 247–248
Linguagem extensível de transformações de folhas de estilo (XSLT), 42–43, 128–129
- Linguagem fonte**, 46–47
Linguagem Intermediária (IL), 125–126
Linguagem JOVIAL, 76–77
Linguagem MAD, 76–77
Linguagem Miranda, 73–74
Linguagem NELIAC, 76–77
Linguagem orientada a objetos baseada no paradigma imperativo, 113–119
Linguagem Perl
 aninhamento de sentenças de seleção e, 376–378
 dados de cadeias e, 275–277
 escopo dinâmico e, 253–254
 formato de nomes e, 228–229
 índices em, 286–287
 linguagens de *scripting* e, 430–431
 matrizes associativas e, 299–300
 matrizes e, 288–289
 métodos de passagem de parâmetros, 439–440
 nascimento de, 392–393
- origens e características e, 118–121
 programa de exemplo, 119–121
Linguagem Scheme, 72–73
Linguagens.NET, 42–43, 50–51
- Linguagens baseadas em C.**
Veja também linguagem estruturada em blocos e, 247–248
 expressões de modo misto e, 355–357
 formato de nomes e, 228–230
 palavras especiais e, 229–231
 sentença `for` e, 391, 394–396, 400–401
 subprogramas aninhados e, 245–246
- Linguagens de marcação/programação híbridas**, 128–129
 emergência das, 42–44
JSP, 128–130
XSLT, 128–129
- Linguagens de programação funcional**
 aplicações de, 717–719
COMMON LISP, 708–709
 comparadas com as linguagens imperativas, 718–721
 funções matemáticas e, 682–685
 fundamentos de, 684–686
Haskell, 712–718
 história das, 681–683
LISP, 68–74, 686–690
ML, 708–713
Scheme, 689–709
 transparência referencial e, 685–686
- Linguagens de programação lógica**, 42–43, 101–104. *Veja também Prolog*
 aplicações de, 756–759
 cálculo de predicados e, 727–731
 forma clausal e, 730–731
 processamento de linguagem natural e, 758–759
 proposições e, 728–730
 provando teoremas e, 731–734
 semântica declarativa e, 733–735
 sistemas de gerenciamento de bancos de dados relacionais e, 757–758

- sistemas especialistas e, 757–759
visão geral das, 733–736
- Linguagens de scripting**, 118–125
como categoria, 42–43
especificidades acerca, 430–431
- JavaScript, 120–121
Linguagem Perl, 118–121
Lua, 124–125
PHP, 122–123
Python, 122–124
Ruby, 123–124
- Linguagens declarativas**, 727
- Linguagens imperativas**, 37–38, 718–720
- Linguagens orientadas a objetos**, 107–111
alocação e liberação de objetos, 553–555
C# e, 573–575
exclusividade de objetos e, 550–551
herança e, 547–549
herança simples e múltipla e, 552–554
introdução a, 537–548
questões de compatibilidade de tipos e, 333–334
questões de projeto e, 549–555
subclasses e subtipos e, 550–552
suporte de Ada 95 e, 554–581
suporte de Java e, 569–574
suporte de Ruby e, 580–584
suporte de Smalltalk e, 554–559
suporte em C++ e, 558–570
verificação de tipos e polimorfismo, 551–553
vinculação dinâmica e estática e, 548–550, 554–555
- Linguagens regulares**, 193–194
- Linguagens visuais, 42–43
- Linha de execução de controle**, 596–597
- Linhas de execução, 116, 622–630
- Linhas de execução em Java
classe Thread e, 623–626
concorrência e, 622–630
prioridades e, 625–626
sincronização de competição e, 625–627
sincronização de cooperação e, 627–630
- LISP, 68–74, 686–690
avaliação e, 71–73
descendentes do, 72–74
gerenciamento do monte e, 322–324
ortogonalidade e, 30–31
primeiro interpretador LISP, 687–690
processo do projeto e, 69–70
processos e, 70–71
programa de exemplo, 71–73
representação interna de, 71–72
sintaxe de, 70–72
tipos de dados e estruturas, 69–71, 686–688
- Listas, descrevendo, 141–142
- Literais sobre carregados**, 282
- LiveScript, 120–121
- Lixo** (variáveis), 316–317
- Lógica para Funções Computáveis – Logic for Computable Functions (LCF), 73–74
- Logical Theorist, 68–69
- LOGO, 108–109
- LR canônico**, 216–217
- Lua, 302–303
construção de seleção e, 377–378
definições de registros e, 304–305
`for` genérico e, 401–402
matrizes associativas e, 300–301
matrizes e, 289–290
origens e características de, 124–125
parâmetros e, 421–422
- M**
- Macros, 51–52
- MACSYMA, 717–718
- Mapeamentos finitos**, 285
- Máquina Virtual Java (JVM), 116
- Máquinas de Turing, 687–689
- Marcar e varrer incremental**, 324–326
- MATH-MATIC, 73–74
- Matrizes**, 284–299
avaliação e, 294–295
fatias e, 293–295
implementação e, 295–299
índices e, 285–287
inicialização e, 289–292
matrizes dinâmicas da pilha, 287–288
matrizes dinâmicas do monte, 287–288
matrizes estáticas, 287–288
matrizes fixas dinâmicas da pilha, 287–288
matrizes fixas dinâmicas do monte, 287–288
matrizes irregulares, 293–294
matrizes retangulares, 293–294
multidimensionais, 296–297, 442–447
operações e, 291–294
parâmetros formais, 420
questões de projeto e, 285
vinculações de índices e, 286–290
- Matrizes associativas**, 298–301
estrutura e operações, 299–301
implementação e, 300–301
- Matrizes `char`, 274–276, 289–291
- Matrizes estáticas**, 287–288
- Matrizes `flex`, 96–97
- Matsumoto, Yukihiko, 123–124
- Mauchly, John, 61–62
- McCarthy, John, 69–70, 687–689
- McCracken, Daniel, 43–44
- Mecanismos de controle de laço posicionados pelo usuário, 398–400
- Membros de dados**, 516–517, 559, 562
- Mensagens**, 547–548
- Metalinguagem (ML)**, 73–74, 140–141, 240
- Metassímbolos**, 153
- Método abstrato**, 549–550
- Método `final`, 571–572
- Método `finalize`, 570–571
- Método sobrescrito**, 548–549
- Metodologias de projeto de programas, 40–42
- Métodos**, 547–548
- Métodos de classe**, 548–549
- Métodos de implementação, 44–52
compilação e, 45–49
de cadeias de caracteres, 277–280
de construções orientadas a objetos, 583–587

- de diagramas de estado, 193–200
de matrizes, 296–299
de matrizes associativas, 300–301
de ponteiros, 321–327
de registros, 307–308
de subprogramas, 471–497
de tipos ordinais definidos pelo usuário, 284
de tipos união, 311–312
interpretação pura e, 48–51
pré-processadores e, 51–52
sistemas de implementação híbridos, 49–51
- Métodos de instância**, 548–549
- Métodos de passagem de parâmetros**, 427–450
considerações de projeto e, 446–447
de linguagens comuns, 437–441
exemplos de, 446–450
implementando, 436–438
matrizes multidimensionais como parâmetros, 442–447
modelo semântico, 427–429
modelos de implementação e, 428–436
passagem por atribuição e, 440–441
passagem por nome e, 436
passagem por referência e, 434–435
passagem por resultado e, 432–433
passagem por valor e, 428–432
passagem por valor-resultado e, 434
verificação de tipos de parâmetros e, 440–443
- Meyer, Bertrand, 112–113
Microsoft Intermediate Language (MSIL), 125–126
Microsoft Visual Studio.NET, 52–53
Milner, Robin, 73–74
Minsky, Marvin, 69–70
MIT, 69–70, 72–73
Mixins, 583–584
ML, 708–713
Modelo semântico de **modo de entrada**, 428–429
Modelo semântico de **modo de entrada e saída**, 428–429
- Modelo semântico de **modo de saída**, 428–429
Modificador **synchronized**, 116
Módulo de carga, 48–49
Módulos, 539–540
Monitores, 606–609
avaliação e, 607–609
sincronização de competição e, 607–608
sincronização de cooperação e, 607–608
Monte, 312–313
Multiplicidade de recursos, 27–28
- N**
Naur, Peter, 76–78, 139–141
Navegadores Web
concorrência e, 593
tratamento de eventos e, 669–670
NetBeans, 52–53
Newell, Alan, 68–69
Nomeação de encapsulamento, 535–540
espaços de nomes em C++ e, 536–538
módulos Ruby e, 539–540
pacotes Ada e, 538–540
pacotes Java e, 537–539
- Nomes**, 227–231
formatos de nomes, 228–230
palavras especiais, 229–231
questões de projeto e, 228–229
variável, 231–232
- Nomes **sensíveis à capitalização**, 229–230
Notação de CamelCase, 228–229
Notação de **complemento de dois**, 270–271
Notação de complemento de um, 270–272
Notação lambda, 683–684
Números de nível, 304–305
Nygaard, Kristen, 94–96
- O**
O problema do produtor-consumidor, 598–599
Objective Caml, 712–713
Objetos
abstração de dados e, 505–506
instâncias de classe e, 547–548
- Objetos protegidos, Ada e, 619–621
Ocultamento de informações em Ada, 509–511, 514
em C++, 518
Opções de tamanho de cadeias, 276–278
Operações de atribuição, cadeias de caracteres e, 274–275
Operações de comparação, cadeias de caracteres e, 274–275
Operador de exponenciação, 345
Operador identidade, 344
Operador **mod**, 345
Operador relacional, 357–358
Operador **rem**, 345
Operadores **Binários**, 343
Operadores de atribuição compostos, 362–363
Operadores de atribuição unários, 362–364
Operadores sobrecarregados, 352–354
Operadores sobrecarregados definidos pelo usuário, 461–462
Operadores **ternários**, 343
Operadores unários, 343
Ordem de avaliação de operadores, 343–349
associatividade e, 148–150, 345–347
expressões condicionais e, 349
expressões Ruby e, 348
parênteses e, 348
precedência e, 343–345
- Ordem de avaliação de operandos, 349–353
efeitos colaterais e, 350–351
transparência referencial e, 351–353
- Ordem de declaração, 248–250
Ordem principal de coluna, 296–297
Ordem principal de linha, 296–297
Ortogonalidade, 28–33
Otimização, 36–37, 46–48
- P**
Pacotes, 509–510
Pacotes de corpo, 510
Pacotes de especificação, 510

- Pacotes filhos**, Ada, 578–580
 Padrão de Ponto Flutuante IEEE 754, 271–273
Pai estático, 246–247
 Palavra reservada **params**, 126–127
 Palavras especiais, 31–32, 142–143, 229–231
Palavras reservadas, 230–231
 Palavras-chave, 229–230
 Papert, Seymour, 108–109
 Paradigmas, programação, 560–561
 Parâmetros como subprogramas, 449–452
Parâmetros de laço, 388–389
Parâmetros formais, 418
Parâmetros posicionais, subgrupo, 418
 Parâmetros que são palavras-chave, 418
Parâmetros reais, subgrupo, 418
 Parênteses, 348
 Pascal
 avaliação e, 98–100
 perspectiva histórica e, 97–99
 portabilidade e, 247–248
 programa de exemplo em, 98–100
 subprogramas e, 451–452
 tipos enumeração e, 281
 Passagem de mensagens, 608–610
 passagem síncrona, 610–612
 projeto de linguagem e, 608–610
 Passagem de mensagens assíncrona, 621–622
Passagem por atribuição, 440–441
Passagem por cópia, 434
Passagem por nome, 436
Passagem por referência, 434–435
Passagem por resultado, 432–433
Passagem por valor, 428–432
Passagem por valor-resultado, 434
Perfil de parâmetros, 416
 Perlis, Alan, 67–68, 73–74
 PHP
 dados de cadeias e, 275–277
 expressões relacionais e, 357–358
- formato de nomes e, 228–229
 interpretação pura e, 49–51
 matrizes associativas e, 300–301
 métodos de passagem de parâmetros e, 439–440
 origens e características de, 122–123
 sentenças `switch` e, 382–383
 variáveis globais e, 249–251
 vinculação dinâmica de tipos e, 235–236
 Web e, 239
- Pilhas de tempo de execução**, 477–478
 PL/C, 92
 PL/CS, 92
 PL/I, 90–94
 avaliação e, 92
 coerção e, 356–357
 estruturas de dados de, 268
 perspectiva histórica e, 90–92
 processo do projeto e, 91–92
 programa de exemplo em, 92–94
 tarefas concorrentes e, 600–601
 tratamento de exceções e, 644–645
 visão geral da linguagem e, 91–92
 PL/S, 25–26
 Plankalkül, 57–61
 perspectiva histórica, 59–60
 visão geral da linguagem, 59–61
 Plataforma de computação.NET, 111–112
Polimorfismo
 linguagens orientadas a objetos e, 551–553
 Smalltalk e, 555–557
Polimorfismo ad hoc, 453–454
Polimorfismo paramétrico, 453–454
 Ponteiro de Ambiente (PE), 472, 477–479
Ponteiros soltos, 314–317, 321–323
Portabilidade, 36–38
pragma, 618–619, 650–652
Precedência de operadores, 146–148, 343–347, 358–359
Precisão, tipo ponto-flutuante e, 271–272
- Pré-condições mais fracas, semântica axiomática e, 171–172
 pré-condições mais fracas e, 171–172
 Prefixados (operadores binários), 343
 Premissa do mundo fechado (Prolog), 753–755
Pré-processadores, 51–52
 Prioridades
 Ada e, 618–620
 linhas de execução de Java, 625–626
 Problema da negação (Prolog), 754–757
Processadores vetoriais, 595–596
 Processamento de linguagem natural, 758–759
 Processamento de listas, 68–70
 Processo de inferência (Prolog), 739–743
 Processo **marcar e varrer**, 323–326
Processos, 597–598
Produção, 140–141
Profundidade de aninhamento (nesting_depth), 485
 Programa Escalonador, 599–601
 Programação automática, 62–63
 Programação de sistemas, 631–632
 Programação em hardware mínima, pseudocódigos, 60–69
 Programação orientada a procedimentos, 41–42
 Programas com **múltiplas linhas de execução**, 596–597
 Programas **quasi-concorrentes**, 596–597
 Projeto de linguagens, 88–89
 Projeto de usuário, 88–89
 Projeto descendente, 40–41
 Projeto orientado a objetos, 41–42
 Projeto Ortogonal, 95–98
 Prolog, 101–104
 aritmética simples e, 742–746
 avaliação e, 103–104
 controle da ordem de resolução e, 750–754
 deficiências do, 750–757
 elementos básicos do, 736–751

- estruturas de lista e, 745–751
 limitações intrínsecas e, 756–757
 origens do, 735–736
 premissa do mundo fechado e, 753–755
 problema da negação e, 754–757
 processo de inferência do, 739–743
 processo do projeto e, 102–103
 sentenças de fatos e, 737–738
 sentenças de objetivo e, 738–740
 sentenças de regras e, 737–739
 termos e, 736–737
 visão geral da linguagem e, 102–103
- Prolog++**, 103–104
Prólogo da ligação de subprogramas, 472
Proposições, 727–730
Proposições atômicas, 728
 Propriedades, 523–524
Protocolo de mensagens, 547–548
Protocolo de subprogramas, 416
Protótipos, 417
 Provando teoremas, cálculo de predicados e, 731–734
 Provas de programa, semântica axiomática e, 180–184
 Pseudocódigos, 60–64
 endereçamento absoluto e e, 62–63
 Short Code, 61–63
 sistema de compilação UNI-VAC e, 62–63
 Speedcoding, 62–63
 Pseudovariável **super**, 557
Python, 122–124
 construção de seleção e, 378–379
 dados de cadeias e, 275–276
 matrizes associativas e, 300–301
 matrizes e, 289–295
 métodos de passagem de parâmetros e, 440–441
 parâmetros e, 421
 sentença **for** de, 395–397
 variáveis globais e, 249–251
- Q**
- Quasi-concorrência**, 462–463
Questões de Projeto
 construções de seleção múltipla, 378–380
 funções, 460–462
 laços controlados logicamente, 396–397
 laços controlados por contadores, 388–389
 linguagens orientadas a objetos, 549–555
 nomes, 228–229
 sentenças de seleção de dois caminhos, 374–375
 subprogramas, 425, 460–462
 suporte linguístico para concorrência, 601–602
 tipos de cadeias de caracteres, 274–275
 tipos de dados abstratos, 508–509
 tipos ponteiro, 312–313
 tipos união, 308–309
 tratamento de exceções e, 643–647
- R**
- Reconhecedores, 154–155
 Reconhecedores de linguagens, 138–139
Reconhecimento, 138–139
Recursão, 481–484
Recursão em cauda, 685–686, 704–706
Recursão esquerda, 209–213
Recursão esquerda direta, 209–210
Recursão esquerda indireta, 210–213
 Referências a subcadeias, 274–275
Referências completamente qualificadas, 305–306
Referências elípticas, 305–306
Referências polimórficas, 549–550
Referências soltas, 314–317
 Refinamento passo a passo, 40–41
Registro de instância de classe (RIC), 584–587
Registros, 301, 304
- Registros de ativação**
 subprograma, 474–476
 variáveis locais dinâmicas da pilha e, 476–480
- Regra**, 140–141
 Regra de apagamento, 210–211
Regra de inferência, 171–172
Regra recursiva, 141–142
Regra recursiva direita, 149–150
Regra recursiva esquerda, 149–150
Reinício, tratamento de exceções e, 644–645
Rendezvous, 610
 Report Program Generator (RPG), 43–44
Resolução, 731
Resolução descendente, 741–742
resume (coroutine), 462–463
 Richards, Martin, 99–100
 Ritchie, Dennis, 100–101
 Robinson, Alan, 731
 Roussel, Phillippe, 102–103, 735–736
 RPG, 43–44
Ruby
 Blocos e, 422–423
 características gerais e, 580–583
 como uma linguagem de *scripting*, 42–43
 construção de seleção e, 377–379
 dados de cadeias e, 275–277
 expressões *case* e, 382–384
 expressões e, 348
 expressões relacionais e, 357–358
 formato de nomes e, 228–229
 herança e, 583–584
 linguagens orientadas a objetos e, 580–584
 matrizes como parâmetros formais e, 420–421
 matrizes e, 289–295
 métodos de passagem de parâmetros e, 440–441
 nomeação de encapsulamento e, 539–540
 origens e características de, 123–124
 tipos de dados abstratos em, 524–527

- variáveis e, 236–237
vinculação dinâmica e, 583–584
Visual Basic e, 89
Russell, Stephen B., 688–689
Rutishauser, H., 78–79
- S**
- Sammet, Jean, 130–131
Scheme, 689–709
composição funcional e, 705–707
definindo funções e, 691–693
expressão lambda e, 691
fluxo de controle e, 694–696
forma funcional aplicar a todos e, 706–707
formas funcionais, 705–707
funções de exemplo, 700–705
funções de lista e, 696–698
funções de numéricas primitivas e, 691
funções de predicado para átomos simbólicos e listas, 699–701
funções de predição numéricas e, 693–694
funções de saída e, 693
funções que constroem código e, 707–709
interpretador Scheme, 689–691
origens de, 689–690
recursão em cauda em, 704–706
variáveis vinculadas e, 692
Schwartz, Jules, 78–79
- Semáforos**, 601–607
Ada e, 608–609
avaliação e, 606–607
binários, 605–606, 608–609
características de, 602–603
sincronização de cooperação e, 602–605
- Semântica**
axiomática, 170–184
definição, 136
denotacional, 164–170
dinâmica, 161–184
operacional, 161–165
passagem de parâmetros e, 427–429
- Semântica axiomática**, 170–184
asserções e, 170–172
avaliação e, 183–184
laços lógicos com pré-teste e, 176–181
provas de programas e, 180–184
sentenças de atribuição e, 172–175
sentenças de seleção e, 176–177
sequências e, 174–176
- Semântica declarativa**, 733–735
- Semântica denotacional**, 135–170
avaliação e, 169–170
estado de um programa e, 167–168
exemplos de, 165–168
expressões e, 167–169
laços lógicos com pré-teste e, 169–170
sentenças de atribuição e, 168–169
- Semântica dinâmica**, 161–184
axiomática, 170–184
denotacional, 164–170
operacional, 161–165
- Semântica estática, 155–156
- Semântica operacional**, 161–165
avaliação e, 163–165
processo básico e, 162–164
- Semântica operacional estrutural**, 162
- Semântica operacional natural**, 162
- Sentença Equivalence, Fortran, 231–232
- Sentenças
case, 98–99, 381–383
de atribuição, 361–366
de atribuição simples, 158–159
de controle, 372–374
de iteração definidas pelo usuário, 400–401
de quebra (break), 380–383
de seleção, 176–177, 374–387
do, 116, 389–391
for, 100–101, 401–402
if-then-else, 387–403
iterativas, 387–403
- switch, 100–101
while, 136–137
yield, 422–423
- Sentenças de atribuição, 361–366
Alvos condicionais e, 361–362
Atribuição como uma expressão, 363–365
atribuições de lista, 365–366
atribuições simples, 361–362
operadores de atribuição compostos e, 362–363
operadores de atribuição unários e, 362–364
semântica axiomática e, 172–175
semântica denotacional e, 168–169
- Sentenças de controle**, 372–374
- Sentenças de fatos (Prolog), 737–738
- Sentenças de iteração**, 387–403
iteração baseada em estruturas de dados, 399–403
laços controlados logicamente e, 396–399
laços controlados por contador, 388–397
mecanismos de controle de laço posicionados pelo usuário, 398–400
- Sentenças de iteração, **pré-teste** e, 387–388
- Sentenças de iteração definidas pelo usuário, 400–401
- Sentenças de regras (Prolog), 737–739
- Sentenças de seleção**, 374–387
construções de seleção múltipla e, 378–387
semântica axiomática e, 176–177
- Sentenças de seleção de dois caminhos**, 374–379
aninhando seletores e, 375–379
expressão de controle e, 374–375
fórmula clausal e, 375–376
questões de projeto e, 374–375
- Sentenças Do em Fortran 95, 389–391

- Sentenças `for`**
 em Ada, 390–391, 394
 em Java, 401–402
 em linguagens baseadas em C, 391, 394–396, 400–401
 em Python, 395–397
 perspectiva histórica e, 100–101
- Sentenças `foreach`**, 119–120, 126–127
- Sentenças `if-then-else`**, 149–152, 349
- Sentenças iterativas, pós-teste** e, 387–388
- Sentenças objetivo** (Prolog), 738–740
- Sentenças `switch`**, 100–101, 125–126, 379–383
- Sequências**, semântica axiomática, 174–176
- SHARE** (grupo de usuários de computadores), 74–77, 90
- Shaw, J. C., 68–69
- Short Code**, 61–63
- Símbolo inicial**, 141–142
- Símbolos não terminais**, 140–141
- Símbolos terminais**, 140–141
- Simon, Herbert, 68–69
- Simplicidade**, 27–29, 486–487
- Simplicidade e ortogonalidade**, 32–33
- SIMULA** 87
 abstração de dados e, 41–42
 Flex e, 108–109, 547–548
 processo do projeto e, 94–96
 Smalltalk e, 109–110
 visão geral da linguagem e, 95–96
- SIMULA I**, 94–95
- Sincronização**, 598–600
 competição, 605–608, 616–619, 625–627
 cooperação, 602–605, 607–608, 615–617, 627–630
- Sincronização de competição**, 598–600, 605–608
 Ada e, 616–619
 linhas de execução em Java e, 625–627
 monitores, 607–608
 semáforos, 605–607
- Sincronização de cooperação**, 598–599, 602–605, 607–608
 Ada e, 615–617
 linhas de execução em Java e, 627–630
 monitores, 607–608
 semáforos, 602–605
- Sintaxe**. Veja também Análise sintática
 ambiguidade e, 144–146
 árvores de análise sintática e, 143–144
 associatividade de operadores e, 148–150
 BNF estendida, 152–155
 definida, 136
 descrevendo listas e, 141–142
 forma de Backus-Naur, 139–141
 fundamentos e, 140–141
 geradores de linguagens e, 138–140
 gramáticas de atributos e, 154–162
 gramáticas e derivações e, 141–143
 gramáticas e reconhecedores e, 154–155
 gramáticas livres de contexto, 139–141
 métodos de descrição formal e, 139–155
 precedência de operadores e, 145–148
 problemas em descrever, 137–140
 projeto, 30–32
 reconhecedores de linguagens e, 138–139
- Sintaxe de Edimburgo**, 736–737
- Sistema APES**, 758–759
- Sistema de compilação UNIVAC**, 62–63
- Sistema de Laning e Zierler**, 63–65
- Sistema de Simulação de Propósito Geral – General Purpose Simulation System (GPSS)**, 43–44
- Sistemas de Computação de Quinta Geração – Fifth Generation Computing Systems (FGCS)**, 735–736
- Sistemas de gerenciamento de bancos de dados relacionais**, 757–758
- Sistemas de implementação híbridos**, 49–51, 190–192
- Sistemas de implementação Just-in-time (JIT)**, 50–51
- Sistemas de implementação pura**, 190
- Sistemas especialistas**, 757–759
- Smalltalk**, 107–111
 avaliação e, 109–111, 557–559
 C++ e, 568–570
 características gerais e, 555–556
 evolução de, 546
 herança e, 557
 processo do projeto e, 108–109
 programa de exemplo, 109–111
 programação orientada a objetos e, 41–42, 554–559
 verificação de tipos e polimorfismo e, 555–557
 visão geral da linguagem e, 109–110
- SNOBOL**, 93–95
- Sobrecarga de operadores**, 27–28, 352–354
- Sobrescrita**, 548–549, 574–575
- Software de sistema**, 24–25
- Software para a Web**, 25–26
- Solaris Common Desktop Environment (CDE)**, 52–53
- Speedcoding**, 62–63
- static_depth**, 485
- Stroustrup, Bjarne, 110–111, 512–513, 560–561
- Structured Query Language (SQL)**, 757–758
- Subclasse**s, 547–548, 550–552
- Subobjetivos** (Prolog), 739–740
- Subprograma sensível ao histórico**, 231–232, 243–244, 261, 425–426
- Subprogramas**. Veja também Subprogramas genéricos; Métodos de passagem de parâmetros
 ambientes de referenciamento local e, 425–428
 ativos, 257
 blocos e, 492–494

- blocos Ruby e, 422–423
 características gerais e, 414–415
 corrotinas e, 462–465
 definições básicas e, 415–417
 efeitos colaterais funcionais e, 460–461
 genéricos, 425, 453–461
 implementação de escopo dinâmico e, 494–497
 implementando subprogramas simples, 473–476
 métodos de passagem de parâmetros e, 427–450
 número de valores retornados e, 461–462
 operadores sobre carregados definidos pelo usuário e, 461–462
 parâmetros e, 417–422, 449–452
 parâmetros formais e, 418
 parâmetros que são palavras-chave e, 418
 parâmetros reais e, 418
 polimórficos, 453–454
 ponteiro de ambiente e, 472, 477–479
 procedimentos e funções e, 423–425
 questões de projeto para, 425, 460–462
 registros de ativação e, 474–476
 semântica de chamadas e de retornos e, 472
 sobre carregados, 425, 451–454
 subprogramas aninhados, 427–428, 483–493, 531–532
 tipos de valores retornados e, 460–462
 variáveis locais dinâmicas da pilha e, 475–484
 variáveis locais e, 425–428
- Subprogramas aninhados,** 483–493
 básico de, 483–485
 encadeamentos estáticos e, 485–493
 escopo estático e, 245–248
 ideia para, 427–428
 organização e, 531–532
 Subprogramas **ativos**, 257, 415
- Subprogramas genéricos**
 em Ada, 453–456
 em C#, 459–461
 em C++, 455–458
 em Java 5.0, 458–460
 questões de projeto e, 425
- Subprogramas polimórficos,** 453–454
- Subprogramas sobre carregados,** 425, 451–454
- Subtipos**, linguagens orientadas a objetos e, 550–552
- Subtração unária, 344
- Superclasse**, 547–548
- Supporte para abstração e, 32–34
- T**
- Tabela de método virtual**, 584–585
- Tabelas de análise sintática LR, 219–222
- Tabelas de símbolos, 46–48
- Tamanho de passo (stepsize)**, 388–390
- Tamanho inteiro **long**, 270–271
- Tamanho inteiro **short**, 270–271
- Tamanhos estáticos de cadeias**, 276–280
- Tarefas**, 597–599
- Tarefas atrizes, Ada, 613–614
- Tarefas **disjuntas**, 598–599
- Tarefas **leves**, 597–599
- Tarefas **pesadas**, 597–598
- Tarefas servidoras, 613–614
- Tempo de vida**, variáveis, 241, 254–256
- Tempo de vinculação**, 233–234
- Terminação**, tratamento de exceções e, 644–645
- terminate**, 616–617
- Término de tarefas, Ada e, 618–619
- Termos** (Prolog), 736–737
- Termos compostos**, 728
- Teste de disjunção par a par, 212–213
- Testes**, 715–716
- Thompson, Ken, 99–100
- TIOBE – Comunidade de Programação, 22–23
- Tipagem forte**, 328–330
- Tipo **byte**, 270–271
- Tipo **compatível**, 326–327
- Tipo definido pelo usuário, 268–270
- Tipo derivado**, 331–332
- Tipo **double**, 271–272
- Tipo **float**, 34–35, 271–272
- Tipo **int**, 232–233
- Tipo ordinal**, 279–280. *Veja também* Tipos ordinais definidos pelo usuário
- Tipos coringa, 459–460
- Tipos de Acesso, Ada, 317–318
- Tipos de cadeias de caracteres**, 274–280
 avaliação e, 277–278
 cadeias e suas operações, 274–277
 implementação de, 277–280
 opções de tamanho de cadeias, 276–278
 questões de projeto e, 274–275
- Tipos de dados**
 definidos, 268
 equivalência de tipos e, 329–334
 legibilidade e, 30–31
 matrizes associativas, 298–301
 ponteiros e referências, 312–329
 primitivos, 270–274
 registros e, 301, 304–308
 teoria e, 333–336
 tipagem forte e, 328–330
 tipo matriz, 284–299
 tipos de cadeias de caracteres, 274–280
 tipos ordinais definidos pelo usuário, 279–284
 tipos união, 308–312
 verificação de tipos e, 326–329
- Tipos de dados abstratos**, 506–508
 definidos pelo usuário, 506–508
 em Ada, 509–516
 em C#, 522–524
 em C++, 516–521
 em Java, 521–522
 em Ruby, 524–527
 exemplo de, 507–508
 parametrizados, 527–531
 ponto flutuante e, 505–507
 questões de projeto para, 508–509

- Tipos de dados parametrizados
em Ada, 527–528
em C# 2005, 530–531
em C++, 528–530
em Java 5.0, 530–531
- Tipos de dados primitivos**, 270–274
complexos, 271–273
de ponto-flutuante, 271–272
decimais, 272–274
inteiros, 270–272
numéricos, 270–272
tipos booleanos, 273–274
tipos caractere, 273–274
- Tipos de enumeração**, 279–284
avaliação, 282–284
projeto, 281–282
tipos de subfaixa, 283–284
- Tipos de referência**, 312–313, 319–322
- Tipos de subfaixa**, 283–284
- Tipos de valores**, 312–313
- Tipos enum**, 125–126, 333–334
- Tipos etiquetados**, 575
- Tipos ordinais definidos pelo usuário, 279–284
implementação de, 284
tipos de enumeração, 279–284
tipos de subfaixa, 283–284
- Tipos ponteiro, 312–329
Ada e, 317–318
avaliação e, 320–322
C e C++ e, 317–320
gerenciamento de ponteiros soltos e, 321–323
gerenciamento do monte e, 322–327
implementação e, 321–327
operações de ponteiros e, 314–316
ponteiros soltos, 314–317
problemas e, 314–318
questões de projeto e, 312–313
representação de, 321–322
variáveis dinâmicas do monte perdidas e, 316–318
- Tipos privados**, 509–511, 514, 518
- Tipos privados **limitados**, 511, 514
- Tipos privados que não são ponteiros, 511, 514
- Tipos registro, 301, 304–308
avaliação e, 306–308
definições de registros, 301, 304–306
implementação e, 307–308
operações em registros e, 306–307
referências a campos de registros, 305–306
- Tipos **struct**, 125–126, 333–334
- Tokens, 192–193
- Tokens literais inteiros, 194–195
- Trade-offs* no projeto de linguagens, 43–45
- Transbordamento**, 357–358
- Transbordamento negativo**, 357–358
- Transformador de predicado**, 177–178
- Transparência referencial**, 351–353, 685–686
- Tratadores**, 202–203, 213–214
- Tratamento de eventos**, 668–675
com Java, 669–675
interfaces gráficas com o usuário e, 668–670
- Tratamento de exceções**
ausência em linguagens, 642–643
conceitos básicos de, 641–644
continuação e, 644–645
definido, 34–35
em Ada, 645–654
em C++, 653–659
em Java, 659–668
erros de faixas de índices e, 641
finalização e, 645–646
operações de entrada e saída e, 640–641
questões de projeto e, 643–647
reinício e, 644–645
término e, 644–645
vantagens do, 642–644
- Trimming*, 96–97
- Tuplas**, 122–123, 291–292, 421
- Turbo Pascal, 98–99
- typedef**, 333–334
- U**
- undef**, 167–168
- Unicode, 73–74, 273–274
- Unidades genéricas**, 453–454
- Unificação**, 732
- Uniões**, 308–312
avaliação e, 311–312
declarações, 333–334
discriminadas *versus* uniões livres, 308–310
implementação e, 311–312
questões de projeto e, 308–309
tipos união em Ada, 309–311
- UNIX**
características do, 51–53
comandos do interpretador de comandos do, 31–32
linguagens de interpretação de comandos e, 276–277
modificadores inseguros, 320–321
primeiras versões do, 99–101
- USE** (grupo de usuários de computadores), 74–75
- Uso de apelidos**, 34–36
- V**
- Valor inteiro com sinal, 270–271
- Valores**
agregados, 284, 285, 289–291
atributos, 159–161
valores do lado direito (*r-values*), 232–233
valores do lado esquerdo (*l-values*), 231–232
- Valores **iniciais** de variável de laço, 388–389
- Valores não instanciados**, 736–737
- Valores retornados, subprogramas e, 460–462
- Valores **terminais** de variáveis de laço, 388–389
- van Rossum, Guido, 122–123
- Variáveis, 230–233
endereço de, 231–233
escopo e, 245–255
nomes, 231–232
tipo de, 232–233
valor de, 232–234

- variáveis de dois ponteiros, 232–233
 variáveis de referência, 232–233
 variáveis estáticas, 241–243
 variáveis não locais, 245–246
 vinculando a atributos, 233–235
- Variáveis anônimas**, 312–313
- Variáveis de classe**, 242–243, 548–549
- Variáveis de instância**, 548–549
- Variáveis de vinculação**, 692
- Variáveis dinâmicas da pilha**, 242–244
- Variáveis dinâmicas do monte**, 312–313
- Variáveis dinâmicas do monte explícitas**, 243–245
- Variáveis dinâmicas do monte implícitas**, 244–246
- Variáveis dinâmicas do monte perdidas**, 316–318
- Variáveis `int`, 34–35, 326–327
- Variáveis locais**, subprogramas, 425–428
- Variáveis locais dinâmicas da pilha**
 desvantagens de, 425–426
 encadeamentos dinâmicos e, 480–481
 exemplo sem recursão, 479–482
 implementando subprogramas com, 475–484
 recursão e, 481–484
 registros de ativação complexos e, 476–480
- Variáveis locais estáticas, 425–427
- Variável variante restrita**, 309–310
- Variável de laço**, 388–389
- Variável especial**, 72–73
- VAX (série de microcomputadores), 28–30
- Vazamento de memória**, 316–318
- Verificação de tipos**, 326–329
 importância da, 34–35
 linguagens orientadas a objetos e, 551–553
 métodos de passagem de parâmetros e, 440–443
 Smalltalk e, 555–557
- Verificação dinâmica de tipos**, 328–329
- Vienna Definition Language (VDL), 164–165
- Vinculação**, 233–246
 de atributos a variáveis, 233–235
 inferência de tipos e, 240–241
 variáveis dinâmicas da pilha e, 242–244
 variáveis dinâmicas do monte explícitas e, 243–245
 variáveis dinâmicas do monte implícitas e, 244–246
 variáveis estáticas e, 241–243
 vinculação dinâmica de tipos, 233–237, 577–579
 vinculação estática de tipos, 233–236
 vinculações de armazenamento e tempo de vida, 241
- Vinculação ad hoc**, 450–451
- Vinculação dinâmica
 C# e, 574–575
 C++ e, 566–569
 Java e, 572–573
- linguagens orientadas a objetos e, 554–555
 programação orientada a objetos e, 548–550
 Ruby e, 583–584
- Vinculação estática**, 554–555
- Vinculação profunda**, 450–451
- Vinculação rasa**, 450–451
- Vinculações de índices, categorias de matrizes e e, 286–290
- virtual**, 566
- Visual BASIC, 32–33, 87
- Visual BASIC.NET, 42–43, 87
- von Neumann, John, 37–38
- W**
- Wall, Larry, 118–119, 392–393, 430–431
- Weinberger, Peter, 118–119
- Wexelblat, Richard, 118–119
- Wheeler, David J., 62–63
- Whitaker, William, 104–105
- Widgets*, 669–670
- Wilkes, Maurice V., 62–63
- Wirth, Niklaus, 97–98, 486–487
- World Wide Web Consortium (W3C), 128–129
- X**
- Xerox, 108–109
- XHTML, 42–43, 250–251
- XML, 42–43
- XSLT, 128–129
- Y**
- yacc, 154–155
- Z**
- Zuse, Konrad, 59–61

