

# LINGUAGENS DE PROGRAMAÇÃO

Robert W. Sebesta

5<sup>a</sup> Edição



Bookman®

Copyrighted material

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

CONCEITOS DE  
LINGUAGENS DE  
PROGRAMAÇÃO

5<sup>a</sup> Edição

ROBERT W. SEBESTA

University of Colorado, Colorado Springs.

Tradução:

JOSÉ CARLOS BARBOSA DOS SANTOS

Consultoria, supervisão e revisão técnica desta edição:

JOÃO CARLOS DE ASSIS RIBEIRO DE OLIVEIRA

*Mestre em Informática pela PUC/RJ.*

*Professor do Departamento de Ciência da Computação da UFJF.*

Reimpressão 2006



2003

Copyrighted material

Obra originalmente publicada sob o título  
*Concepts of programming languages 5/e*  
© Addison Wesley Higher Education, 2002  
Publicado conforme acordo com Pearson Education, Inc.,  
publicando sob o selo Addison Wesley Higher Education,

ISBN 0-201-75295-6

Capa:  
MÁRIO RÖHNELT

Preparação do original:  
CLÓVIS VICTÓRIA JR.  
DENISE NOWACZYK  
AMANDA RAMOS FRANCISCO

Supervisão editorial:  
ARYSINHA JACQUES AFFONSO

Editoração eletrônica e filmes:  
GRAFLINE EDITORA GRÁFICA

Muitos dos nomes usados por fabricantes e revendedores para identificar seus produtos são reivindicados como marcas registradas. Quando os editores sabiam da reivindicação, os nomes foram impressos com destaque.

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

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

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

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

SAC 0800 703-3444

IMPRESSO NO BRASIL  
PRINTED IN BRAZIL

Hidden page

Hidden page

Hidden page

Capítulo 2, ele pode ser lido isoladamente, independentemente dos outros capítulos. O Capítulo 3 aborda o principal método formal de descrição da sintaxe das linguagens de programação, chamado BNF. Segue-se a descrição das gramáticas de atributos, que têm papel proeminente no projeto de compiladores. A difícil tarefa da descrição semântica é então explorada, incluindo breve introdução aos três métodos mais usuais: semântica operacional, axiomática e denotacional.

O Capítulo 4, novo, introduz a análise léxica e sintática. Esse capítulo é direcionado aos colegas que não mais necessitam um curso de projeto de compiladores em seus currículos. O restante do livro não depende do material desse capítulo. Os capítulos de 5 a 14 descrevem em detalhe os itens de projeto para as principais construções das linguagens imperativas. Em cada caso, as escolhas de projeto são apresentadas e avaliadas.

Especificamente, o Capítulo 5 cobre as várias características das variáveis, o Capítulo 6 sobre tipos de dados e o Capítulo 7 apresenta expressões e instruções de atribuição. O Capítulo 8 descreve as instruções de controle, os Capítulos 9 e 10 abordam subprogramas e suas implementações e o Capítulo 11 examina as facilidades para abstração de dados. O Capítulo 12 trata das características das linguagens que suportam a programação orientada a objetos (herança e vinculação dinâmica de métodos). O Capítulo 13 aborda as unidades de programas concorrentes e o Capítulo 14 a manipulação de exceções. Os dois últimos capítulos (15 e 16) descrevem os dois mais importantes paradigmas alternativos da programação: programação funcional e programação lógica. O Capítulo 15 apresenta uma introdução à linguagem Scheme, incluindo a descrição de algumas de suas funções primitivas, formas especiais e formas funcionais, bem como alguns exemplos de funções simples escritas em Scheme. Breves introduções ao COMMON LISP, ML e Haskell são apresentadas para ilustrar algumas espécies diferentes de linguagens funcionais. O Capítulo 16 introduz a programação lógica com a linguagem Prolog.

---

## Para o professor

---

No curso de linguagem de programação de nível júnior, na Universidade de Colorado, em Colorado Springs, o livro é usado da seguinte maneira: tipicamente, cobrimos os Capítulos 1 e 3 em detalhe. O Capítulo 2 exige pouco tempo de exposição por não abranger conteúdo técnico difícil. Entretanto, os alunos acham interessante e benéfica a sua leitura. Uma vez que nenhum material dos capítulos subsequentes depende do Capítulo 2, ele pode, conforme observado anteriormente, ser pulado integralmente. O Capítulo 4 não é coberto, uma vez que é requerido dos alunos um curso de projeto de compiladores. Os capítulos de 5 a 9 e 11 devem ser relativamente fáceis para estudantes que possuam experiência em C++, Ada ou Java. Os Capítulos 10, 12, 13 e 14 são mais desafiadores e exigem exposições mais detalhadas.

Os Capítulos 15 e 16 são inteiramente novos para a maioria dos estudantes de nível júnior. É bom que os processadores das linguagens Scheme e Prolog estejam disponíveis para os estudantes que deverão aprender o material desses capítulos. Existe material suficiente nesses dois capítulos para permitir que os estudantes se envolvam com alguns programas simples. Os cursos de graduação provavelmente não serão capazes de cobrir os dois últimos capítulos detalhadamente. Os cursos de pós-graduação, contudo, ao saltarem partes dos primeiros capítulos sobre linguagens imperativas, serão capazes de discutir completamente as linguagens não imperativas.

Hidden page

Hidden page

# Sumário

<b>Capítulo 1</b>	<b>Aspectos Preliminares .....</b>	<b>15</b>
1.1	Motivos para Estudar os Conceitos de Linguagens de Programação	16
1.2	Dominios de Programação	19
1.3	Critérios de Avaliação da Linguagem	22
1.4	Influências sobre o Projeto da Linguagem	33
1.5	Categorias de Linguagem	36
1.6	Custo/Benefício no Projeto da Linguagem	36
1.7	Métodos de Implementação	37
1.8	Ambientes de Programação	44
<b>Capítulo 2</b>	<b>Evolução das Principais Linguagens de Programação .....</b>	<b>47</b>
2.1	A Linguagem Plankalkül de Zuse	48
2.2	Programação de Hardware Mínima: Pseudocódigos	51
2.3	O IBM 704 e o FORTRAN	53
2.4	Programação Funcional: LISP	58
2.5	O Primeiro Passo Rumo à Sofisticação: ALGOL 60	63
2.6	Informatizando Registros Comerciais: COBOL	69
2.7	O Início do Compartilhamento de Tempo ( <i>timesharing</i> ): BASIC	73
2.8	Tudo para Todos: PL/I	76
2.9	Duas Primeiras Linguagens Dinâmicas: APL e SNOBOL	79
2.10	A Origem da Abstração de Dados: SIMULA 67	80
2.11	Projeto Ortogonal: ALGOL 68	82
2.12	Algumas Descendentes Importantes dos ALGOLs	84
2.13	Programação Baseada na Lógica: Prolog	90
2.14	O Maior Esforço de Projeto da História: Ada	91
2.15	Programação Orientada a Objeto: Smalltalk	96
2.16	Combinando Recursos Imperativos e Orientados a Objeto: C++	99
2.17	Programando a World Wide Web: Java	102
<b>Capítulo 3</b>	<b>Descrevendo a Sintaxe e a Semântica .....</b>	<b>109</b>
3.1	Introdução	110
3.2	O Problema Geral de Descrever a Sintaxe	111
3.3	Métodos Formais para Descrever a Sintaxe	112
3.4	Gramáticas de Atributos	126
3.5	Descrevendo o Significado dos Programas: Semântica Dinâmica	131

<b>Capítulo 4</b>	<b>Análise Léxica e Sintática .....</b>	<b>153</b>
4.1	Introdução 154	
4.2	Análise Léxica 155	
4.3	O Problema da Análise Sintática 158	
4.4	Análise Descendente Recursiva 161	
4.5	Análise Baixo-Cima 166	
<b>Capítulo 5</b>	<b>Nomes, Vinculações, Verificação de Tipos e Escopos .....</b>	<b>175</b>
5.1	Introdução 176	
5.2	Nomes 176	
5.3	Variáveis 179	
5.4	O Conceito de Vinculação 181	
5.5	Verificação de Tipos 189	
5.6	Tipificação Forte 189	
5.7	Compatibilidade de Tipos 191	
5.8	Escopo 193	
5.9	Escopo e Tempo de Vida 200	
5.10	Ambientes de Referenciamento 201	
5.11	Constantes Nomeadas 203	
5.12	Inicialização de Variáveis 204	
<b>Capítulo 6</b>	<b>Tipos de Dados .....</b>	<b>211</b>
6.1	Introdução 212	
6.2	Tipos de Dados Primitivos 213	
6.3	Tipos Cadeia de Caracteres 216	
6.4	Tipos Ordinais Definidos pelo Usuário 221	
6.5	Tipos Matriz 225	
6.6	Matrizes Associativas 237	
6.7	Tipos Registro 238	
6.8	Tipos União 243	
6.9	Tipos Conjunto 248	
6.10	Tipos Ponteiro 250	
<b>Capítulo 7</b>	<b>Expressões e Instruções de Atribuição .....</b>	<b>267</b>
7.1	Introdução 268	
7.2	Expressões Aritméticas 269	
7.3	Operadores Sobrecarregados 276	
7.4	Conversões de Tipo 278	
7.5	Expressões Relacionais e Booleanas 281	
7.6	Avaliação Curto-Círcuito 283	
7.7	Instruções de Atribuição 284	
7.8	Atribuição de Modo Misto 288	
<b>Capítulo 8</b>	<b>Estruturas de Controle no Nível da Instrução .....</b>	<b>293</b>
8.1	Introdução 294	
8.2	Instruções Compostas 295	
8.3	Instruções de Seleção 296	
8.4	Instruções Iterativas 306	
8.5	Desvio Incondicional 318	
8.6	Comandos Protegidos 320	
8.7	Conclusões 322	

<b>Capítulo 9</b>	<b>Subprogramas .....</b>	<b>329</b>
9.1	Introdução 330	
9.2	Fundamentos dos Subprogramas 330	
9.3	Questões de Projeto Referentes aos Subprogramas 335	
9.4	Ambientes de Referência Locais 336	
9.5	Métodos de Passagem de Parâmetros 337	
9.6	Parâmetros que São Nomes de Subprograma 355	
9.7	Subprogramas Sobre carregados 357	
9.8	Subprogramas Genéricos 358	
9.9	Compilação Separada e Independente 362	
9.10	Questões de Projeto Referentes a Funções 363	
9.11	Acessando Ambientes Não-Locais 364	
9.12	Operadores Sobre carregados Definidos pelo Usuário 367	
9.13	Co-Rotinas 367	
<b>Capítulo 10</b>	<b>Implementando Subprogramas .....</b>	<b>375</b>
10.1	A Semântica Geral das Chamadas e dos Retornos 376	
10.2	Implementando Subprogramas FORTRAN 77 376	
10.3	Implementando Subprogramas em Linguagens Assemelhadas ao ALGOL 379	
10.4	Blocos 397	
10.5	Implementando o Escopo Dinâmico 399	
10.6	Implementando Parâmetros que São Nomes de Subprograma 403	
<b>Capítulo 11</b>	<b>Tipos de Dados Abstratos .....</b>	<b>409</b>
11.1	O Conceito de Abstração 410	
11.2	Encapsulamento 411	
11.3	Introdução à Abstração de Dados 412	
11.4	Questões de Projeto 415	
11.5	Exemplos de Linguagens 416	
11.6	Tipos de Dados Abstratos Parametrizados 425	
<b>Capítulo 12</b>	<b>Suporte para Programação Orientada a Objeto .....</b>	<b>431</b>
12.1	Introdução 432	
12.2	Programação Orientada a Objeto 432	
12.3	Questões de Projeto Referentes às Linguagens Orientadas a Objeto 436	
12.4	Visão Geral da Smalltalk 440	
12.5	Introdução à Linguagem Smalltalk 441	
12.6	Exemplo de Programas Smalltalk 451	
12.7	Recursos em Grande Escala da Smalltalk 456	
12.8	Avaliação da Smalltalk 458	
12.9	Suporte para Programação Orientada a Objeto em C++ 459	
12.10	Suporte para Programação Orientada a Objeto em Java 466	
12.11	Suporte para Programação Orientada a Objeto em Ada 95 469	
12.12	Suporte para Programação Orientada a Objeto em Eiffel 472	
12.13	O Modelo de Objetos de Java Script 474	
12.14	Implementação de Construções Orientadas a Objeto 477	
<b>Capítulo 13</b>	<b>Concorrência .....</b>	<b>483</b>
13.1	Introdução 484	
13.2	Introdução à Concorrência no Nível de Subprograma 487	
13.3	Semáforos 491	
13.4	Monitores 495	
13.5	Passagem de Mensagens 500	
13.6	Concorrência em Ada 95 509	
13.7	Linhos de Execução Paralela Java 511	
13.8	Concorrência no Nível de Comando 516	

<b>Capítulo 14</b>	<b>Manipulação de Exceções .....</b>	<b>521</b>
14.1	Introdução à Manipulação de Exceções	522
14.2	Manipulação de Exceções em PL/I	527
14.3	Manipulação de Exceções em Ada	532
14.4	Manipulação de Exceções em C++	537
14.5	Manipulação de Exceções em Java	541
<b>Capítulo 15</b>	<b>Linguagens de Programação Funcionais .....</b>	<b>551</b>
15.1	Introdução	552
15.2	Funções Matemáticas	553
15.3	Fundamentos das Linguagens de Programação Funcionais	555
15.4	A Primeira Linguagem de Programação Funcional: LISP	556
15.5	Uma Introdução à Scheme	559
15.6	COMMON LISP	574
15.7	ML	576
15.8	Haskell	577
15.9	Aplicações das Linguagens Funcionais	580
15.10	Uma Comparação entre as Linguagens Funcionais e Imperativas	581
<b>Capítulo 16</b>	<b>Linguagens de Programação Lógicas .....</b>	<b>585</b>
16.1	Introdução	586
16.2	Uma breve Introdução ao Cálculo de Predicados	586
16.3	Cálculo de Predicados e Demonstração de Teoremas	589
16.4	Uma Visão Geral da Programação Lógica	591
16.5	As Origens do Prolog	593
16.6	Os Elementos Básicos do Prolog	593
16.7	Deficiências do Prolog	606
16.8	Aplicações da Programação Lógica	611
16.9	Conclusões	613
<b>Bibliografia .....</b>		<b>617</b>
<b>Índice .....</b>		<b>623</b>

# Capítulo 1

## Aspectos Preliminares



### Konrad Zuse

Konrad Zuse projetou uma série de computadores eletromecânicos entre 1936 e 1944 na Alemanha. Em 1945, projetou uma linguagem de programação algorítmica completa, a Plankalkül, jamais implementada, e sua descrição completa nem mesmo foi publicada até 1972.

- 1.1** Motivos para Estudar os Conceitos de Linguagens de Programação
- 1.2** Domínios de Programação
- 1.3** Critérios de Avaliação da Linguagem
- 1.4** Influências sobre o Projeto da Linguagem
- 1.5** Categorias de Linguagem
- 1.6** Custo/Benefício no Projeto da Linguagem
- 1.7** Métodos de Implementação
- 1.8** Ambientes de Programação

Antes de iniciarmos nossa exposição dos conceitos de linguagens de programação, devemos considerar alguns aspectos. Primeiro, discutiremos alguns motivos pelos quais os estudantes de ciência da computação e os desenvolvedores de software profissionais devem estudar os conceitos gerais de projeto e de avaliação das linguagens. Essa discussão é valiosa para os que acreditam ser o conhecimento funcional de uma ou de duas linguagens de programação suficiente para os cientistas da computação. Os principais domínios de programação serão descritos brevemente. Em seguida, uma vez que o livro avalia recursos de linguagem, apresentaremos uma lista de critérios por meio dos quais se pode fazer julgamentos. As duas principais influências sobre o projeto da linguagem, sobre a arquitetura de máquina e sobre as metodologias de projeto de programa serão, então, discutidas. Depois, descreveremos algumas das principais relações custo/benefício que devem ser consideradas durante o projeto da linguagem.

Uma vez que este livro também trata da implementação de linguagens de programação, este capítulo inclui uma visão geral das abordagens mais comuns à implementação. Por fim, descreveremos brevemente alguns exemplos de ambientes de programação e discutiremos seu impacto na produção de software.

## **1.1 Motivos para Estudar os Conceitos de Linguagens de Programação**

---

É natural que os estudantes imaginem como se beneficiarão do estudo dos conceitos de linguagens de programação. Afinal de contas, uma grande quantidade de outros tópicos da ciência da computação merece um estudo sério. O que apresentamos a seguir é o que acreditamos ser uma lista obrigatória dos benefícios potenciais relativos ao estudo de conceitos de linguagens.

- *Aumento da capacidade de expressar idéias.* Acredita-se que a profundidade de nossa capacidade intelectual seja influenciada pelo poder expressivo da linguagem em que comunicamos nossos pensamentos. Os que possuem uma compreensão limitada da linguagem natural são limitados na complexidade de expressar seus pensamentos, especialmente em termos de profundidade de abstração. Em outras palavras, é difícil para as pessoas conceberem estruturas que não podem descrever, verbalmente ou por escrito. Programadores inscritos no processo de desenvolver softwares vêm-se similarmente embaralhados. A linguagem na qual desenvolvem o software impõe limites quanto aos tipos de estruturas de controle, de estruturas de dados e de abstrações que eles podem usar; assim, as formas de algoritmos possíveis de serem construídas também são limitadas.

O conhecimento de uma variedade mais ampla de recursos de linguagens de programação reduz essas limitações no desenvolvimento de software. Os programadores podem aumentar a variedade de seus processos intelectuais de desenvolvimento de software aprendendo novas construções de linguagem.

Pode-se argumentar que aprender as capacidades de outras linguagens não ajudará um programador obrigado a usar uma linguagem sem essas capacidades. Esse argumento não se sustenta, porém, porque freqüentemente as facilidades da linguagem podem ser simuladas em outras linguagens que não suportam esses recursos diretamente.

Por exemplo, depois de ter aprendido as funções de manipulação de matrizes do FORTRAN 90 (ANSI, 1992), um programador C++ (Stroustrup, 1997), seria levado naturalmente a construir subprogramas para oferecer essas operações. O mesmo é verdadeiro em relação a muitas outras construções complexas discutidas neste livro.

O estudo dos conceitos das linguagens de programação forma uma apreciação dos recursos valiosos da linguagem e encoraja os programadores a usá-los.

O fato de muitos recursos das várias linguagens poderem ser simulados em outras não diminui significativamente a importância de projetar linguagens com o melhor conjunto de recursos. Sempre é melhor usar um recurso cujo projeto foi integrado na linguagem do que usar uma simulação desse, o qual, muitas vezes, é menos elegante e mais desajeitado em uma linguagem que não o suporta.

- *Maior embasamento para a escolha de linguagens apropriadas.* Muitos programadores profissionais tiveram pouca educação formal em ciência da computação; ao contrário, aprenderam a programar sozinhos ou por meio de programas de treinamento dentro da própria organização. Tais programas freqüentemente ensinam uma ou duas linguagens diretamente pertinentes ao trabalho da organização. Muitos outros programadores receberam seu treinamento formal em um passado distante. As linguagens que aprenderam não são mais usadas, e muitos recursos agora disponíveis não eram amplamente conhecidos. O resultado disso é que muitos programadores, quando lhes é dada a possibilidade de escolha das linguagens para um novo projeto, continuam a usar aquela com a qual estão mais familiarizados, mesmo que ela seja pouco adequada ao novo projeto. Se tais programadores estivessem mais familiarizados com as outras linguagens disponíveis, especialmente com os seus recursos particulares, estariam em uma posição melhor para fazerem uma escolha consciente.
- *Capacidade aumentada para aprender novas linguagens.* A programação de computadores é uma disciplina jovem, e as metodologias de projeto, as ferramentas de desenvolvimento de software e as linguagens de programação ainda estão em um estágio de contínua evolução. Isso torna o desenvolvimento de software uma profissão excitante, mas também significa que a aprendizagem contínua é fundamental. O processo de aprender uma nova linguagem de programação pode ser extenso e difícil, especialmente para alguém que esteja à vontade com somente uma ou com duas linguagens e que jamais examinou os conceitos em geral. Assim que for adquirida uma completa compreensão dos conceitos fundamentais das linguagens, será mais fácil ver como esses estão incorporados ao projeto da linguagem aprendida.

Por exemplo, programadores que entendem o conceito de abstração de dados terão mais facilidade para aprender como construir tipos de dados abstratos em Java (Gosling et al., 1996) do que aqueles que não estão absolutamente familiarizados com tal exigência. O mesmo fenômeno ocorre nas linguagens naturais. Quanto mais você conhece a gramática de sua língua nativa, mais fácil achará aprender uma segunda língua natural. Além disso, aprender uma segunda língua também tem o efeito colateral benéfico de ensiná-lo mais a respeito de seu primeiro idioma.

Por fim, é essencial que os programadores ativos conheçam o vocabulário e os conceitos fundamentais das linguagens de programação, para que possam ler e entender seus manuais e sua literatura de vendas de linguagens e de compiladores.

- *Entender melhor a importância da implementação.* Ao aprender os conceitos de linguagens de programação, tanto é interessante como necessário tocar nas questões de implementação que afetam esses conceitos. Em alguns casos, a compreensão das questões de implementação leva a um entendimento do porquê das linguagens serem projetadas daquela maneira. Isso, por sua vez, leva à capacidade de usar uma linguagem de modo mais inteligente, como ela foi projetada. Podemos nos tornar melhores programadores ao entendermos as escolhas que podemos fazer entre as construções de linguagens de programação e as consequências das opções.

Certos tipos de erro de programa somente podem ser encontrados e corrigidos por um programador que conheça certos detalhes de implementação relacionados. Outro benefício de entender as questões de implementação é que isso nos permite visualizar como um computador executa várias construções da linguagem. Esta última afirmação, por sua vez, fomenta o entendimento da eficiência relativa de construções alternativas a serem escolhidas para o programa. Por exemplo, programadores que sabem pouco da implementação da recursão, muitas vezes, não sabem que um algoritmo recursivo é tipicamente muito mais lento do que um iterativo equivalente.

- *Aumento da capacidade de projetar novas linguagens.* Para um estudante, a possibilidade de vir a projetar uma nova linguagem de programação no futuro pode parecer remota. Porém, a maioria dos programadores profissionais ocasionalmente projeta linguagens de um tipo ou de outro. Por exemplo, muitos dos sistemas exigem que o usuário interaja de alguma maneira, mesmo que somente para introduzir dados e comandos. Em situações simples, somente alguns valores de dados são introduzidos, e a linguagem do formato de entrada é trivial. Por outro lado, pode ser necessário que o usuário percorra diversos níveis de menus e introduza uma variedade de comandos, como no caso de um processador de texto. Nesses sistemas, a interface com o usuário é um problema complexo de projeto. A sua forma é projetada pelo desenvolvedor de sistemas, e os critérios para julgá-la são semelhantes aos usados para julgar o projeto de uma linguagem de programação. Um exame crítico das linguagens de programação, portanto, ajudará no projeto desses sistemas complexos e, mais comumente, ajudará os usuários a examinar e a avaliar esses produtos.
- *Avanço global da computação.* Por fim, há uma visão global da computação que pode justificar o estudo dos conceitos das linguagens de programação. Embora normalmente seja possível determinar o motivo pelo qual uma linguagem particular de programação tornou-se popular, nem sempre é claro, pelo menos em retrospectiva, que as linguagens mais populares são as melhores disponíveis. Em alguns casos, pode-se concluir que uma linguagem se tornou popular porque aqueles com capacidade de optar ainda não estavam familiarizados com os conceitos de linguagens de programação.

Por exemplo, muitas pessoas acreditam que teria sido melhor se o ALGOL 60 (Backus et al., 1962) tivesse substituído o FORTRAN no início da década de 60, porque aquele é mais elegante e tem instruções de controle muito melhores do que este, entre outras razões. O fato disso não ter acontecido se deve parcialmente aos programadores e aos gerentes de desenvolvimento de software daquela época, cuja maioria não entendia claramente o projeto conceitual do ALGOL 60. Eles achavam sua descrição difícil de ler (o que era verdade) e ainda mais difícil

de entender. Não apreciaram os benefícios da estrutura em bloco, da recursão e das instruções de controle bem estruturadas, de modo que deixaram de ver os benefícios do ALGOL 60 em relação ao FORTRAN.

Obviamente, muitos outros fatores contribuíram para a falta de aceitação do ALGOL 60, conforme veremos no Capítulo 2. Entretanto, o fato dos usuários de computador geralmente não terem consciência dos benefícios da linguagem desempenhou um papel importante.

Em geral, se aqueles que escolhem as linguagens forem melhor informados, talvez linguagens melhores se sobreponham mais rapidamente às ruins.

## 1.2 Domínios de Programação

Os computadores são usados em uma infinidade de diferentes áreas, desde o controle de usinas elétricas nucleares à armazenagem de registros de talões de cheques pessoais. Por causa dessa grande diversidade no seu espaço, linguagens de programação com metas muito diferentes têm sido desenvolvidas. Nesta seção, discutiremos brevemente algumas das áreas de aplicações de computadores e suas linguagens associadas.

### 1.2.1 Aplicações Científicas

Os primeiros computadores digitais, que surgiram na década de 40, eram usados e, de fato, foram inventados para aplicações científicas. Tipicamente, as aplicações científicas têm estruturas de dados simples, mas exigem um grande número de computações aritméticas com números reais. As estruturas de dados mais comuns são as matrizes (*arrays*)<sup>\*N. de T. Array:</sup>; as estruturas de controle mais comuns são os laços de contagem e as seleções. As linguagens de programação de alto nível, inventadas para aplicações científicas, foram projetadas para suprir essas necessidades. A concorrente delas foi a linguagem de montagem (*assembly*); desse modo, a eficiência era a primeira preocupação. A primeira linguagem para aplicações científicas foi o FORTRAN. O ALGOL 60 e a maioria de suas descendentes também se destinam a serem usadas nessa área, ainda que tenham sido projetadas também para outras áreas relacionadas. Para algumas aplicações científicas cuja eficiência é a principal preocupação, como aquelas comuns nas décadas de 50 e 60, nenhuma linguagem subsequente é significativamente melhor do que o FORTRAN.

<sup>\*N. de T. Array:</sup> (1) arranjo ordenado. Em computação, arranjo de elementos de memória em um ou em diversos níveis ou planos; matriz (coleção de dados similares armazenados sob o mesmo nome) ou estrutura ordenada de elementos acessíveis, referenciados por números, usada para conter tabelas ou conjunto de dados relacionados e do mesmo tipo. (2) (programação) Um conjunto de itens de dados de tipos idênticos, distinguidos por seus índices (ou "subscritos"). O número de dimensões que uma matriz pode ter depende da linguagem, mas usualmente ele é ilimitado. Uma variável comum simples ("escalar") poderia ser considerada como uma matriz zero-dimensional. Uma referência a um elemento de matriz é escrita como  $A[i,j,k]$ , em que  $A$  é o nome da matriz e  $i, j$  e  $k$  são os índices. A linguagem C é peculiar em termos de que cada índice é escrito em colchetes separados, p. ex.  $A[i][j][k]$ . Isso expressa o fato de que, em C, uma matriz  $N$ -dimensional é, de fato, um vetor, sendo que cada um de seus elementos é uma matriz  $N-1$  dimensional. Os elementos de uma matriz são armazenados contiguamente.

### 1.2.2 Aplicações Comerciais

O uso de computadores para aplicações comerciais iniciou-se na década de 50. Equipamentos especiais foram desenvolvidos para tal propósito, juntamente com linguagens especiais. A primeira linguagem de alto nível bem-sucedida para negócios foi o COBOL (ANSI, 1985), que apareceu em 1960. Ela ainda é a mais comumente usada para essas aplicações. As linguagens comerciais são caracterizadas por facilidades para produzir relatórios elaborados, por maneiras precisas de descrever e por armazenar números decimais e textos, além da capacidade de especificar operações aritméticas decimais.

Com o advento dos microcomputadores, surgiram novas maneiras de usar computadores para fazer negócios, especialmente de pequeno porte. Duas ferramentas específicas que podem ser usadas em pequenos computadores, os sistemas de planilhas eletrônicas e os sistemas de bancos de dados, foram desenvolvidas para os negócios e agora são amplamente usadas.

Há poucos desenvolvimentos nas linguagens de aplicação comercial além do COBOL. Portanto, este livro não discute as linguagens de aplicação comercial, a não ser para apresentar a história do desenvolvimento do COBOL no Capítulo 2.

### 1.2.3 Inteligência Artificial

A inteligência artificial (IA) é uma área abrangente das aplicações de computador caracterizada pelo uso de computações simbólicas em vez de numéricas. Computação simbólica significa que símbolos, que consistem em nomes no lugar de números, são manipulados. Além disso, a computação simbólica é feita de maneira mais conveniente com listas encadeadas de dados em vez de matrizes. Esse tipo de programação, às vezes, requer mais flexibilidade do que outros domínios de programação. Por exemplo, em algumas aplicações de IA a capacidade de criar e de executar segmentos de código durante a execução é conveniente.

A primeira linguagem de programação desenvolvida para aplicações de IA amplamente utilizada foi a funcional LISP (McCarthy et al., 1965), que surgiu em 1959. A maioria das aplicações de IA foi escrita em LISP ou em uma de suas parentes próximas. No início da década de 70, surgiu uma abordagem alternativa para tais aplicações — a programação lógica, usando a linguagem Prolog (Clocksin e Mellish, 1997). O Scheme, um dialeto do LISP, e o Prolog são apresentados 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 computador são coletivamente conhecidos como seu **software básico** que é usado quase continuamente e, portanto, deve ter eficiência na execução. Portanto, uma linguagem para tal domínio deve oferecer uma execução rápida. Além disso, deve ter recursos de baixo nível que permitam ao software fazer interface com os dispositivos externos a serem escritos.

Nas décadas de 60 e 70, alguns fabricantes de computadores como, por exemplo, a IBM, a Digital e a Burroughs (agora UNISYS), desenvolveram linguagens de alto nível especiais orientadas para a máquina, ou seja, para o software de seus equipamentos. Para os computadores de grande porte da linguagem de montagem da IBM, a linguagem era a PL/S, um dialeto da PL/I; para a Digital, era a BLISS, em uma linguagem em um nível logo acima; para a Burroughs, era a Extended ALGOL.

O sistema operacional UNIX foi escrito quase inteiramente em C (ANSI, 1989), o que o tornou relativamente fácil de portar ou de mover, para máquinas diferentes. Algumas das características do C tornam boa a sua aplicação em programação de sistemas. Ela é de baixo nível, sua execução é eficiente e não sobrecarrega o usuário com muitas restrições de segurança. Os programadores de sistemas freqüentemente são excelentes e não acreditam precisar dessas restrições. Alguns, entretanto, acham a linguagem C muito perigosa para ser usada em sistemas grandes e importantes.

### **1.2.5 Linguagens de Scripting**

As linguagens de scripting desenvolveram-se lentamente no decorrer dos últimos 25 anos. Tais linguagens são usadas colocando-se uma lista de comandos, chamados de *script*, em um arquivo para serem executados. A primeira, chamada de *sh* (de *shell*), iniciou-se como uma pequena coleção de comandos interpretados como chamadas aos subprogramas do sistema que executavam funções de utilidade como, por exemplo, gerenciamento e filtragem simples de arquivo. A essa base foram adicionadas variáveis, instruções de fluxo de controle, funções e várias outras capacidades. O resultado é uma linguagem de programação completa. Uma das mais poderosas e conhecidas delas é a *ksh* (Bolsky e Korn, 1995), desenvolvida por David Korn no *Bell Laboratories*.

A *awk* é outra linguagem de scripting, desenvolvida por Al Aho, Brian Kernighan e Peter Wienberger no *Bell Laboratories* (Aho et al., 1988). A *awk* começou como uma linguagem de geração de relatórios, mas, depois, passou a ser usada para propósitos mais gerais. A *tc1* é uma linguagem de scripting extensível, desenvolvida por John Ousterhout na Universidade da Califórnia em Berkeley (Ousterhout, 1994). A *tc1* agora está combinada com a *tk*, uma linguagem que fornece um método para construir aplicações X Window. A linguagem Perl, desenvolvida por Larry Wall, era originalmente uma combinação da *sh* e da *awk* (Wall et al., 2000). A Perl desenvolveu-se significativamente desde seu início, e tornou-se uma linguagem de programação poderosa, ainda que um tanto primitiva. Embora ela ainda seja freqüentemente chamada de linguagem de scripting, preferimos imaginá-la como uma linguagem de programação estranha, mas completa. Desde o advento da World Wide Web, a popularidade da Perl cresceu drasticamente, principalmente porque ela é quase ideal para programação de *Common Gateway Interface (CGI)*\*.

A JavaScript (Flanagan, 1998) é uma linguagem de scripting desenvolvida pela Netscape para uso tanto em servidores Web como em navegadores. Ela tem crescido, tanto como linguagem, como em popularidade nos últimos anos.

As linguagens de scripting, de maneira geral, têm contribuído pouco para o desenvolvimento de linguagens de programação mais convencionais. Porém, a Perl e a Java Script têm diversos recursos interessantes que discutiremos posteriormente neste livro.

### **1.2.6 Linguagens de Propósitos Especiais**

Uma grande quantidade de linguagens para propósitos especiais surgiu no decorrer dos últimos 40 anos. Elas variam da RPG, usada para produzir relatórios comerciais, à APT, usada para instruir ferramentas de máquina programáveis, à GPSS, usada para simulação de sistemas. Este livro não discute as linguagens de propósitos especiais, principalmente

\*N. de T. CGI: Definição de como o servidor da Web e o navegador se comunicam para permitir ao autor da página fornecer conteúdo dinâmico em função da solicitação dos usuários.

por causa de sua estreita aplicabilidade e da dificuldade de compará-las com outras linguagens.

### 1.3 Critérios de Avaliação da Linguagem

Conforme observou-se, o propósito deste livro é examinar cuidadosamente os conceitos fundamentais das várias construções e das capacidades das linguagens de programação. Também avaliaremos esses recursos, concentrando-nos em seu impacto sobre o processo de desenvolvimento (inclusive manutenção) de software. Para levarmos isso a efeito, precisaremos de um conjunto de critérios de avaliação. Porém, uma lista de critérios é necessariamente controversa, porque é virtualmente impossível conseguir até mesmo dois cientistas da computação que concordem com o valor de determinada característica de linguagem em relação a outras. Apesar dessas diferenças, a maioria dos cientistas da computação concordaria que os critérios discutidos nas subseções seguintes são importantes.

Algumas das características que influenciam os mais importantes critérios são mostradas na Tabela 1.1; os mesmos são discutidos nas seções seguintes.

#### 1.3.1 Legibilidade

Um dos critérios mais importantes para julgar uma linguagem de programação é a facilidade com que os programas podem ser lidos e entendidos. Antes de 1970, o desenvolvimento de software era muito imaginado em termos da escrita do código. Na década de 70, entretanto, o conceito de ciclo de vida do software (Booch, 1987) foi desenvolvido; a codificação foi relegada a um papel muito menos importante, e a manutenção foi reconhecida como uma parte importante do ciclo, especialmente em termos de custo. Uma vez que a facilida-

**TABELA 1.1** Critérios de Avaliação da Linguagem e as Características que os Afetam

Característica	Critérios		
	Legibilidade	Capacidade de Escrita (Writability)	Confiabilidade
Simplicidade/ortogonalidade	•	•	•
Estruturas de controle	•	•	•
Tipos de dados e estruturas	•	•	•
Projeto da sintaxe	•	•	•
Supporte para abstração		•	•
Expressividade		•	•
Verificação de tipos			•
Manipulação de exceções			•
Apelido (Aliasing)* restrito			•

\*N. de T. Aliasing: determinação de um nome alternativo.

de de manutenção é determinada, em grande parte, pela legibilidade dos programas, ela tornou-se uma medida importante da qualidade dos programas e das linguagens.

A legibilidade deve ser considerada no contexto do domínio do problema. Por exemplo, se um programa que descreve uma computação tiver sido escrito em uma linguagem não-projetada para esse uso, o programa pode ser antinatural e enrolado, tornando-o incomumente difícil de ser lido.

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

### 1.3.1.1 Simplicidade Global

A simplicidade global de uma linguagem de programação afeta fortemente sua legibilidade. Antes de mais nada, uma linguagem com um grande número de componentes básicos é mais difícil de ser aprendida do que uma com poucos desses componentes. Os programadores que precisam usar uma linguagem grande tendem a aprender um subconjunto dela e ignorar seus outros recursos. Esse padrão de aprendizagem, às vezes, é usado para desviarse do grande número de componentes da linguagem, mas tal argumento não é válido. Ocorrerão problemas de legibilidade sempre que o autor do programa tiver aprendido um subconjunto diferente daquele com o qual o leitor está familiarizado.

Uma segunda característica que complica uma linguagem de programação é a multiplicidade de recursos — mais de uma maneira de realizar uma operação particular. Por exemplo, em C, o usuário pode incrementar uma variável inteira simples de quatro maneiras diferentes:

```
cont = cont + 1  
cont += 1  
cont++  
++cont
```

Não obstante as duas últimas instruções tenham significados ligeiramente diferentes entre elas e de todas as outras em alguns casos, todas as quatro têm o mesmo significado quando usadas como expressões independentes. Essas variações serão discutidas no Capítulo 7.

Um terceiro problema potencial é a sobrecarga\* (*overloading*) de operador, na qual um único símbolo tem mais de um significado. Embora seja um recurso útil, pode levar a uma reduzida legibilidade se for permitido aos usuários criar suas próprias sobrecargas e não as constituírem criteriosamente. Por exemplo, é bem aceitável sobreclarregar + e usá-lo tanto para adição de números inteiros como para números reais. Aliás, tal sobreclara simplifica uma linguagem ao reduzir o número de operadores. Porém, suponhamos que o programador tenha definido que + seja usado entre operandos de matrizes unidimensionais para significar a soma de todos os elementos de ambas as matrizes. Uma vez que o significado da adição de vetores é bastante diferente deste último, isso tornaria o programa mais confuso tanto para o autor como para seus leitores. Um exemplo ainda mais extremo de confusão do programa seria um usuário definir + entre dois operandos de vetor para significar a diferença entre seus respectivos primeiros elementos. A sobreclara de operadores será discutida adicionalmente no Capítulo 7.

A simplicidade nas linguagens, evidentemente, pode ser levada muito longe. Por exemplo, a forma e o significado da maioria das instruções da linguagem de montagem são

\*N. de T. Sobrecarga: tradução literal de *overloading* que, em programação, é a possibilidade de usar o mesmo nome para mais de uma variável ou procedimento, exigindo que o compilador diferencie, baseando-se no contexto.

modelos de simplicidade, como você poderá ver quando considerar as instruções que aparecem na próxima seção. Essa mesma simplicidade, entretanto, torna os programas em linguagem de montagem menos legíveis. Uma vez que lhes faltam instruções de controle mais complexas, suas estruturas são menos evidentes; o fato de suas instruções serem simples exige um número bem maior do que o necessário para programas equivalentes escritos em uma linguagem de alto nível. Esses mesmos argumentos aplicam-se ao caso menos extremo das linguagens de alto nível com controle e com construções de estruturação de dados inadequados.

### 1.3.1.2 Ortogonalidade

Ortogonalidade em uma linguagem de programação significa que um conjunto relativamente pequeno de construções primitivas pode ser combinado em um número relativamente pequeno de maneiras para construir as estruturas de controle e de dados da linguagem. Além disso, toda combinação possível de primitivas é legal e significativa. Por exemplo, considere os tipos de dados. Suponhamos que uma linguagem tenha quatro tipos de dados primitivos, inteiro, real, real com precisão dupla e caractér, e dois operadores de tipo, matriz e ponteiro. Se os dois operadores de tipo puderem ser aplicados a si mesmos e aos quatro tipos de dados primitivos, um grande número de estruturas de dados poderá ser definido. Porém, se não for permitido aos ponteiros apontar para matrizes, muitas dessas possibilidades seriam eliminadas.

O significado de um recurso de linguagem ortogonal é livre de contexto de sua aparência em um programa (o nome ortogonal vem do conceito matemático de vetores ortogonais, independentes um do outro). A ortogonalidade parte de uma simetria de relações entre primitivas. Os ponteiros devem ser capazes de apontar para qualquer tipo de variável ou estrutura de dados. A falta de ortogonalidade acarreta exceções às regras da linguagem.

Podemos ilustrar o uso da ortogonalidade como um conceito de projeto, comparando um aspecto das linguagens de montagem dos computadores de grande porte da IBM com a série VAX de superminicomputadores. Consideramos uma única situação simples: adicionar valores inteiros de 32 bits que residem na memória ou nos registradores e substituir um dos dois valores pela soma. Os computadores de grande porte da IBM têm duas instruções para essa finalidade sob as formas

```
A    Reg1, célula_de_memória  
AR   Reg1, Reg2
```

em que Reg1 e Reg2 representam registradores. A semântica delas é:

$$\begin{aligned} \text{Reg1} &\leftarrow \text{conteúdo(Reg1)} + \text{conteúdo(célula\_de\_memória)} \\ \text{Reg1} &\leftarrow \text{conteúdo(Reg1)} + \text{conteúdo(Reg2)} \end{aligned}$$

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

```
ADDL operando_1, operando_2
```

cuja semântica é

$$\text{operando}_2 \leftarrow \text{conteúdo(operando}_1) + \text{conteúdo(operando}_2)$$

Nesse caso, qualquer um dos operandos pode ser um registrador ou uma célula de memória.

O projeto da instrução VAX é ortogonal pelo fato de uma única operação poder usar registradores ou células de memória como operandos. São duas maneiras de especificar operandos que podem ser combinadas de todas as maneiras. O projeto IBM não é ortogonal. Somente duas combinações de operandos são legais em quatro possibilidades, e ambas

exigem diferentes instruções, A e AR. O projeto IBM é mais restrito e, portanto, menos fácil de ser escrito. Por exemplo, você não pode adicionar dois valores e armazenar a soma em uma localização da memória. Além disso, ele é mais difícil de ser aprendido por causa das restrições e da instrução adicional.

A ortogonalidade está estreitamente relacionada à simplicidade: quanto mais ortogonal é o projeto de uma linguagem, menos exceções as regras da linguagem exigirão. Menos exceções significam um grau mais elevado de regularidade no projeto, o que torna a linguagem mais fácil de ser aprendida, lida e entendida. Qualquer um que tenha aprendido uma parte significativa da língua inglesa pode atestar a dificuldade de entender suas muitas exceções à regra (por exemplo, i antes de e, exceto depois de c).

Como exemplos da falta de ortogonalidade em uma linguagem de alto nível, manifestada como exceções à regra, consideremos as seguintes regras em C. Embora a linguagem C possua dois tipos de dados estruturados, matrizes e registros (`structs`), registros podem ser retornados de funções, mas matrizes não. Um membro de uma estrutura pode ser qualquer tipo de dado, exceto `void` ou uma estrutura do mesmo tipo. Um elemento de matriz pode ser qualquer tipo de dado, exceto `void` ou uma função. Parâmetros são passados por valor, a menos que sejam matriz, em cujo caso são, com efeito, passados por referência (porque o aparecimento do nome de uma matriz isoladamente, ou seja, sem nenhum colchete, em um programa em C é interpretado como o endereço do primeiro elemento da matriz). Uma expressão de adição simples, como

`a + b`

usualmente significa que os valores de `a` e `b` são trazidos da memória e adicionados. Porém, se acontecer de `a` ser um ponteiro, o valor de `b` pode ser modificado antes que a adição se desenvolva. Por exemplo, se `a` apontar para um valor que tenha dois bytes de extensão, o valor de `b` será multiplicado por 2 antes que a adição se desenvolva. O tipo de `a`, que é o contexto esquerdo de

`+ b`

forçará o valor de `b` a ser modificado antes que ele seja adicionado a `a`.

Muita ortogonalidade também pode causar problemas. Talvez a linguagem de programação mais ortogonal seja o ALGOL 68 (van Wijngaarden et al., 1969). Toda construção de linguagem em ALGOL 68 tem um tipo, e não há nenhuma restrição para ele. Além disso, a maioria das construções produz valores. Essa liberdade de combinação permite construções extremamente complexas. Por exemplo, uma condicional pode aparecer como o lado esquerdo de uma atribuição, juntamente com declarações e com outras várias instruções, contanto que o resultado seja uma localização. Essa forma extrema de ortogonalidade acarreta uma complexidade desnecessária. Além disso, uma vez que as linguagens exigem um grande número de primitivas, um elevado grau de ortogonalidade resultará em uma explosão de combinações. Sendo assim, mesmo que as combinações sejam simples, seu elevado número leva à complexidade.

Portanto, simplicidade em uma linguagem é, pelo menos em parte, o resultado de uma combinação de um número relativamente pequeno de construções primitivas e do uso limitado do conceito de ortogonalidade.

Alguns acreditam que as linguagens funcionais oferecem uma boa combinação de simplicidade e de ortogonalidade. Uma linguagem funcional, como por exemplo o LISP, é uma linguagem em que as computações são feitas principalmente aplicando funções a determinados parâmetros. Em comparação, nas linguagens imperativas como C, C++ e Java, as computações normalmente são especificadas com variáveis e com instruções de atribuição. As linguagens funcionais oferecem potencialmente a maior simplicidade global porque

podem realizar tudo com uma única construção, a chamada à função, a qual pode ser combinada com outras chamadas a funções de maneira simples. Essa elegância simples é a razão pela qual alguns pesquisadores de linguagens são atraídos para as linguagens funcionais como a primeira alternativa para as linguagens não-funcionais complexas como o C++.

Outros fatores, como a eficiência, porém, têm impedido que as linguagens funcionais tornem-se mais populares.

### 1.3.1.3 Instruções de Controle

A revolução da programação estruturada da década de 70 foi, em parte, uma reação à má legibilidade causada pelas limitadas instruções de controle de algumas das linguagens das décadas de 50 e 60. Em particular, reconheceu-se amplamente que o uso indiscriminado das instruções *goto* reduz criticamente a legibilidade do programa.

Um programa que pode ser lido de cima a baixo é muito mais fácil de entender do que o que exige ao leitor pular de uma instrução a outra não-adjacente para seguir a ordem de execução. Em certas linguagens, entretanto, instruções *goto* que desviam para cima, às vezes, são necessárias; por exemplo, elas constroem laços WHILE em FORTRAN 77. Restringir instruções *goto* das seguintes maneiras pode tornar os programas mais legíveis:

- Elas devem preceder seus alvos, exceto quando usadas para formar laços.
- Seus alvos nunca devem estar muito distantes.
- Seu número deve ser limitado.

Faltavam, às versões do BASIC e do FORTRAN disponíveis no início da década de 70, as instruções de controle que permitem fortes restrições ao uso de *gotos*, de modo que escrever programas altamente legíveis nessas linguagens era difícil. A maioria das linguagens de programação projetadas desde o final da década de 60, tem incluído instruções suficientes, de forma que a necessidade da instrução *goto* foi quase eliminada. Portanto, o projeto da estrutura de controle de uma linguagem agora é um fator menos importante na legibilidade do que no passado.

### 1.3.1.4 Tipos de Dados e Estruturas

A presença de facilidades adequadas para definir tipos de dados e estruturas de dados em uma linguagem é outro auxílio significativo para a legibilidade. Por exemplo, suponhamos que um tipo numérico seja usado para um sinalizador porque não há nenhum tipo booleano na linguagem. Nessa linguagem, poderíamos ter uma atribuição como

```
soma_e_muito_grande=1
```

cujo significado não é claro, enquanto que em uma linguagem com tipos booleanos, teríamos

```
soma_e_muito_grande=true
```

cujo significado é perfeitamente claro. Similarmente, tipos de dados registro (*record*) constituem um método para representar registros de empregados mais legível do que um conjunto de vetores similares, um para cada item de dados em um registro de empregado, o

método exigido em uma linguagem sem registros. Por exemplo, no FORTRAN 77, uma matriz de registros de empregados poderia ser armazenada nos seguintes vetores:

```
CHARACTER (LEN = 30) NOME (100)
INTEGER IDADE (100), NUMERO_EMPREGADO (100)
REAL SALARIO (100)
```

Depois, um empregado particular é representado pelos elementos desses quatro vetores com o mesmo valor de índice.

### 1.3.1.5 Considerações sobre a Sintaxe

A sintaxe ou a forma dos elementos de uma linguagem têm um efeito significativo sobre a legibilidade dos programas. O que apresentamos a seguir são três exemplos de opções de projeto sintático que afetam a legibilidade:

- *Formas identificadoras.* Restringir os identificadores a tamanhos muito pequenos prejudica a legibilidade. Se os identificadores puderem ter seis caracteres no máximo, como no FORTRAN 77, muitas vezes não é possível usar nomes conotativos para as variáveis. Um exemplo mais extremo é o BASIC (ANSI, 1978b) do American National Standards Institute (ANSI), na qual um identificador poderia consistir somente de uma única letra ou de uma única letra seguida de um dígito. Outras questões de projeto referentes a formas identificadoras serão discutidas no Capítulo 5.
- *Palavras especiais.* A aparência do programa e, desse modo, a sua legibilidade são fortemente influenciadas pelas formas das palavras especiais de uma linguagem (por exemplo, **while**, **class** e **for**). Especialmente importante é o método para formar instruções compostas ou grupos de instrução principalmente em construções de controle. Diversas linguagens usam pares coincidentes de palavras ou de símbolos especiais para formar grupos. O Pascal exige pares **begin-end** para formar grupos em todas as construções de controle, exceto a instrução **repeat**, na qual elas podem ser omitidas (um exemplo da falta de ortogonalidade do Pascal). A linguagem C usa chaves para a mesma finalidade. Ambas as linguagens sofrem porque os grupos de instrução são sempre encerrados da mesma maneira, o que torna difícil determinar qual grupo está sendo finalizado quando um **end** ou **}** aparece. O FORTRAN 90 e a Ada tornam isso mais claro, usando uma sintaxe de fechamento distinta para cada tipo de grupo de instrução. Por exemplo, a Ada usa **end if** para finalizar uma construção de seleção, e **end loop** para finalizar uma construção de laço. Esse é um exemplo do conflito entre a simplicidade resultante de um número menor de palavras reservadas, como no Pascal, e a maior legibilidade que pode resultar do uso de um número maior de palavras reservadas, como na Ada.

Outra questão importante é se as palavras especiais de uma linguagem podem ser usadas como nomes para variáveis de programa. Se puderem, os programas resultantes podem ser muito confusos. Por exemplo, no FORTRAN 90, palavras especiais como **DO** e **END** são nomes de variáveis legais, de forma que o aparecimento de tais palavras em um programa pode conotar ou não algo especial.

- *Forma e significado.* Projetar instruções, a fim de que sua aparência indique, pelo menos, parcialmente sua finalidade, é um auxílio evidente para a legibilidade. A semântica, ou o significado, deve seguir diretamente da sintaxe ou da forma. Em alguns casos, esse princípio é violado por duas construções de linguagem idênti-

cas ou similares quanto à aparência, mas com significados diferentes, dependendo, talvez, do contexto. Em C, por exemplo, o significado da palavra reservada **static** depende do contexto de seu aparecimento. Se for usada na definição de uma variável dentro de uma função, significa que a variável é criada no momento da compilação. Se for usada na definição de uma variável fora de todas as funções, significa que esta é visível somente no arquivo em que sua definição aparece; ou seja, ela não é exportada desse arquivo.

Uma das principais reclamações a respeito dos comandos shell do UNIX (Kernighan e Pike, 1984) é que a aparência deles nem sempre sugere sua função. Por exemplo, o comando UNIX **grep** pode ser decifrado somente pelo conhecimento prévio, ou talvez pela habilidade e pela familiaridade com o editor UNIX, **ed**. Sua aparência não tem conotação alguma para os iniciantes UNIX. (No **ed**, o comando **/expressão\_regular/** procura por uma subcadeia que coincide com a expressão regular. Precedê-lo com **g** irá torná-lo um comando global, especificando que o escopo da procura é todo o arquivo que está sendo editado. Colocar um **p** depois do comando especificará que as linhas com subcadeias coincidentes sejam impressas. Assim, **g/expressão\_regular/p**, que evidentemente pode ser abreviado para **grep**, imprime todas as linhas de um arquivo que contenham subcadeias que coincidem com a expressão regular.

### 1.3.2 Capacidade de Escrita (*Writability*)

Capacidade de escrita é uma medida de quão facilmente uma linguagem pode ser usada para criar programas para um domínio de problema escolhido. A maioria das características da linguagem que afeta a legibilidade também afeta a capacidade de escrita. Isso se segue diretamente do fato de que escrever um programa exige uma leitura freqüente da parte que já foi escrita pelo programador.

Como acontece com a legibilidade, a capacidade de escrita deve ser considerada no contexto do domínio de problema-alvo de uma linguagem. Simplesmente, não é razoável comparar a capacidade de escrita de duas linguagens no domínio de uma aplicação particular quando uma foi projetada para essa aplicação e a outra não. Por exemplo, a capacidade de escrita do COBOL (ANSI, 1985) e a da APL (Gilman e Rose, 1976) são drasticamente diferentes para criar um programa capaz de lidar com estruturas de dados bidimensionais, para as quais a APL é ideal. Suas capacidades de escrita também são diferentes na produção de relatórios financeiros com formatos complexos, para os quais o COBOL foi projetado.

As subseções seguintes descrevem os fatores mais importantes que influenciam a capacidade de escrita de uma linguagem.

#### 1.3.2.1 Simplicidade e Ortogonalidade

Se uma linguagem tiver um grande número de diferentes construções, alguns programadores podem não estar familiarizados com todas elas. Isso pode levar ao uso inadequado de alguns recursos e ao desuso de outros que podem ser ou mais elegantes ou mais eficientes (ou ambos) do que aqueles usados. Pode, até mesmo, ser possível, como foi observado por Hoare (1973), usar recursos desconhecidos accidentalmente, com resultados bizarros. Portanto, um número menor de construções primitivas e um conjunto consistente de regras para combiná-las (isto é, ortogonalidade) é muito melhor do que, simplesmente, ter um

número grande de primitivas. Um programador pode projetar uma solução para um problema complexo depois de aprender somente um conjunto simples de construções primitivas.

Por outro lado, demasiada ortogonalidade pode resultar em prejuízo para a capacidade de escrita. Erros ao escrever programas podem não ser detectados, uma vez que quase todas as combinações de primitivas são legais. Isso pode levar a absurdos no código que não podem ser descobertos pelo compilador.

### 1.3.2.2 Suporte para Abstração

Colocando brevemente, **abstração** significa a capacidade de definir e, depois, de usar estruturas ou operações complicadas de uma maneira que permita ignorar muitos dos detalhes. A abstração é um conceito fundamental no projeto de linguagens de programação contemporâneas. Isso é um reflexo do papel central que a abstração desempenha nas modernas metodologias de projeto de programas. O grau de abstração permitido por uma linguagem de programação e a naturalidade de sua expressão são, por conseguinte, muito importantes para sua capacidade de escrita. As linguagens de programação podem suportar duas categorias distintas de abstração: processo e dados.

Um exemplo simples de abstração do processo é o uso de um subprograma para implementar um algoritmo de classificação exigido diversas vezes em um programa. Sem este último, o código de classificação teria de ser replicado em todos os lugares em que fosse necessário; isso tornaria o programa muito mais longo e mais tedioso de ser escrito. O mais importante, se o subprograma não fosse usado, é que o código que usou o subprograma de classificação seria atravancado com detalhes do algoritmo de classificação, obscurecendo grandemente o fluxo e o intento global desse código.

Como um exemplo de abstração de dados, considere uma árvore binária que armazena dados inteiros em seus vértices e que seria normalmente implementada em FORTRAN 77 como três vetores paralelos de números inteiros, em que dois dos inteiros são usados como subscritos<sup>\*</sup> para especificar os vértices descendentes. Em C++ e em Java, esses três podem ser implementados usando-se uma abstração de um vértice da árvore na forma de uma classe simples com dois ponteiros e um número inteiro. A naturalidade desta última representação torna muito mais fácil de escrever um programa com árvores binárias em tais linguagens do que escrever em FORTRAN 77. É uma simples questão do domínio de solução de problemas da linguagem estar mais próximo do domínio do problema.

O suporte global para abstração, evidentemente, é um fator importante na capacidade de escrita de uma linguagem.

### 1.3.2.3 Expressividade

Expressividade, em uma linguagem, pode referir-se a diversas características diferentes. Na linguagem APL, por exemplo, significa que há operadores muito poderosos permitindo que uma grande quantidade de computação seja realizada com um programa muito pequeno. Mais comumente, significa que uma linguagem tem formas relativamente convenientes, em vez de desajeitadas, de especificar computações. Por exemplo, em C, a notação `cont++` é

\*N. de T. Subscrito: subscript. É um método para referenciar dados em uma tabela. Por exemplo, na tabela Tabpreco, a instrução para referenciar um preço específico pode ser tabpreco(item), onde item é variável de subscrito.

mais conveniente e mais breve do que `cont = cont + 1`. Da mesma forma, o operador booleano `and then` em Ada é uma maneira conveniente de especificar uma avaliação em curto-circuito de uma expressão booleana. A inclusão da instrução `for` em Java torna mais fácil a escrita de laços de contagem do que com o uso de `while`, também possível. Tudo isso aumenta a capacidade de escrita de uma linguagem.

### 1.3.3 Confiabilidade

Diz-se que um programa é **confiável** se ele se comportar de acordo com suas especificações sob todas as condições. As subseções seguintes descrevem diversos recursos de linguagem que exercem um efeito significativo sobre a confiabilidade de programas.

#### 1.3.3.1 Verificação de Tipos

**Verificar tipos** é, simplesmente, testar se existem erros de tipo em determinado programa, ou pelo compilador ou durante a execução do programa. A verificação de tipos é um fator importante na confiabilidade da linguagem. Uma vez que a verificação de tipos em tempo de execução é dispendiosa, a verificação em tempo de compilação é mais desejável. Além disso, quanto mais cedo forem detectados erros em um programa, menos dispendioso será para fazer os reparos necessários. O projeto da Ada exige verificações dos tipos de quase todas as variáveis e expressões em tempo de compilação, exceto quando o usuário declara explicitamente que a verificação de tipos deve ser suspensa. Isso virtualmente elimina os erros de tipo durante a execução de programas escritos em Ada. Os tipos e a verificação destes serão descritos nos Capítulos 5 e 6.

Um exemplo de como a falta de verificação de tipos, em tempo de compilação ou em tempo de execução, levam a incontáveis erros de programa é o uso de subprogramas como parâmetros na linguagem C original (Kernighan e Ritchie, 1978). Nessa linguagem, o tipo de um argumento em uma chamada a função não é verificado para determinar se ele coincide com o do parâmetro formal correspondente na função. Uma variável `inteira` pode ser usada como um argumento em uma chamada a uma função que espera um tipo `real` como seu parâmetro formal, e nem o compilador, nem o sistema em tempo de execução detectarão a incoerência. Isso acarreta problemas, cuja fonte, muitas vezes, é difícil de determinar-se (em resposta a esse e a outros problemas semelhantes, os sistemas UNIX incluem um programa utilitário chamado `lint` que verifica se existem problemas em programas C). Os subprogramas e a passagem de parâmetros serão discutidos no Capítulo 9.

#### 1.3.3.2 Manipulação de Exceções

A capacidade de um programa de interceptar erros em tempo de execução (bem como outras condições incomuns detectadas pelo programa), pôr em prática medidas corretivas e, depois, prosseguir é um grande auxílio para a confiabilidade. Tal facilidade da linguagem é chamada de **manipulação de exceções**<sup>1</sup>. Ada, C++ e Java incluem grandes capacidades de manipular exceções, mas essas facilidades praticamente não existem em muitas linguagens mais populares, como o C e o FORTRAN. A manipulação de exceções será discutida no Capítulo 14.

<sup>1</sup>N. de T. Manipulação de exceções: também chamada de tratamento de exceções.

### 1.3.3.3 Apelidos

Definido livremente, **apelidos** é ter dois ou mais métodos, ou nomes, distintos para fazer referência à mesma célula da memória. Agora é amplamente aceito que os apelidos são um recurso perigoso em uma linguagem de programação. A maioria das linguagens de programação permite algum tipo de apelido — por exemplo, membros de união e ponteiros definidos para apontar para a mesma variável em C. Em ambos os casos, duas diferentes variáveis de programa podem referir-se à mesma célula da memória. Alguns tipos de apelidos, descritos nos Capítulos 5 e 9, podem ser proibidos pelo projeto de uma linguagem.

Em algumas linguagens, o apelido é usado para superar deficiências nas facilidades de abstração de dados. Outras linguagens o restringem muito para aumentarem sua confiabilidade.

### 1.3.3.4 Legibilidade e Capacidade de Escrita

Tanto a legibilidade como a capacidade de escrita influenciam a confiabilidade. Um programa escrito em uma linguagem que não suporta maneiras naturais de expressar os algoritmos exigidos usará, necessariamente, métodos não-naturais. Estes últimos têm menos probabilidade de estarem corretos para todas as situações possíveis. Quanto mais fácil é escrever um programa, mais probabilidade ele tem de ser correto.

A legibilidade afeta a confiabilidade tanto nas fases de escrita como nas de manutenção no ciclo de vida. Programas de difícil leitura complicam também sua escrita e sua modificação.

## 1.3.4 Custo

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

Primeiro, há o custo do treinamento de programadores para usar a linguagem. Essa é uma função da simplicidade e da ortogonalidade da linguagem e da experiência dos programadores. Embora linguagens mais poderosas não sejam, necessariamente, mais difíceis de aprender, elas freqüentemente o são.

Em segundo lugar, há o custo para escrever programas na linguagem. Essa é uma função da capacidade de escrita, a qual depende de sua proximidade de propósito com a aplicação particular. Os esforços originais para projetar e implementar linguagens de alto nível foram motivados pelo desejo de diminuir os custos da criação de software.

Tanto o custo para treinar programadores como para escrever programas em uma linguagem podem ser significativamente reduzidos em um bom ambiente de programação. Os ambientes de programação serão discutidos na Seção 1.7.

Em terceiro lugar, há o custo para compilar programas na linguagem. Um grande empecilho para os primeiros usos da Ada era o custo proibitivamente elevado para rodar os compiladores Ada de primeira geração. Esse problema foi diminuído pelo surgimento de compiladores melhores.

Em quarto lugar, o custo para executar programas é grandemente influenciado pelo projeto da linguagem. Se ela exigir muitas verificações de tipos durante a execução, proibirá a execução rápida de código, independentemente da qualidade do compilador. Ainda que a eficiência de execução fosse a principal preocupação no projeto das primeiras linguagens, considera-se isso menos importante agora.

Pode ser feita uma análise simples de custo/benefício entre custo de compilação e velocidade de execução do código compilado. Otimização é o nome dado para a coleção de métodos que os compiladores podem usar para diminuir o tamanho e/ou aumentar a velocidade de execução do código produzido. Se ocorrer pouca ou nenhuma otimização, a compilação pode ser feita muito mais rapidamente do que se houver um esforço significativo para produzir código otimizado. O esforço de compilação extra resulta em uma execução de código mais rápida. A escolha entre as duas alternativas é determinada pelo ambiente no qual o compilador será usado. Em um laboratório para estudantes de programação iniciantes que usam um bocado de tempo de compilação, mas pouco de execução de código (seus programas são pequenos e eles devem executar corretamente somente uma vez), pouca ou nenhuma otimização deve ser feita. Em um ambiente de produção, em que programas completos são executados muitas vezes, é melhor pagar o custo extra para otimizar o código.

O quinto fator é o custo do sistema de implementação da linguagem. Um dos fatores que explicam a rápida aceitação de Java é que sistemas compiladores/interpretadores estavam à disposição para ela logo depois que seu projeto foi lançado pela primeira vez. Uma linguagem de programação cujo sistema de implementação seja caro, ou rode somente em hardware caro, terá muito menos chance de tornar-se popular.

O sexto fator é o custo da má confiabilidade. Se o software falhar em um sistema crítico, como uma usina de energia nuclear ou uma máquina de raios X, o custo poderia ser muito elevado. As falhas de sistemas não-críticos também podem ser muito caras em termos de futuro comercial ou de ações judiciais em função de sistemas de software defeituosos.

A consideração final é o custo de manutenção dos programas, que inclui tanto correções como modificações para adicionar novas capacidades. O custo da manutenção de software depende de uma série de características da linguagem, mas principalmente da legibilidade. Uma vez que a manutenção freqüentemente é feita por pessoas que não o autor original do software, uma legibilidade ruim pode tornar a tarefa extremamente desafiadora.

A importância da manutenção de software não pode ser exagerada. Estima-se que, para grandes sistemas de software com tempos de vida relativamente longos, os custos de manutenção podem atingir de duas a quatro vezes os custos de desenvolvimento (Sommerville, 2000).

De todos os fatores que contribuem para os custos da linguagem, três são os mais importantes: desenvolvimento do programa, manutenção e confiabilidade. Uma vez que esses são funções da capacidade de escrita e da legibilidade, os dois últimos critérios de avaliação são, por sua vez, os mais importantes.

Obviamente, um grande número de critérios está disponível para avaliar linguagens de programação. Um, por exemplo, é a portabilidade ou a facilidade com que os programas podem ser mudados de uma implementação para outra. A portabilidade é mais fortemente influenciada pelo grau de padronização da linguagem. Algumas linguagens, como o BASIC, não são padronizadas, tornando os programas muito difíceis de serem mudados de uma implementação para outra. A padronização é um processo difícil que consome tempo. Uma comissão começou a trabalhar para produzir uma versão padrão do C++ em 1989. Foi aprovada só em 1998.

Generalidade (a aplicabilidade a uma ampla faixa de utilizações) e boa definição (a perfeição e a precisão do documento oficial que define a linguagem) são dois outros critérios.

A maioria dos critérios, especialmente a legibilidade, a capacidade de escrita e a confiabilidade não é nem precisamente definida, nem exatamente mensurável. Eles são

conceitos úteis, entretanto, e fornecem valiosas avaliações sobre o projeto e sobre as linguagens de programação.

Uma nota final sobre os critérios de avaliação: os critérios de projeto da linguagem são pesados diferentemente a partir de perspectivas diversas. Os implementadores da linguagem estão preocupados primeiramente com a dificuldade de implementar as construções e os recursos daquela. Os usuários da linguagem estão preocupados em primeiro lugar com a capacidade de escrita e depois com a legibilidade. Os projetistas provavelmente enfatizarão a elegância e a capacidade de atrair um uso generalizado. Essas características, às vezes, entram em conflito.

## 1.4 Influências sobre o Projeto da Linguagem

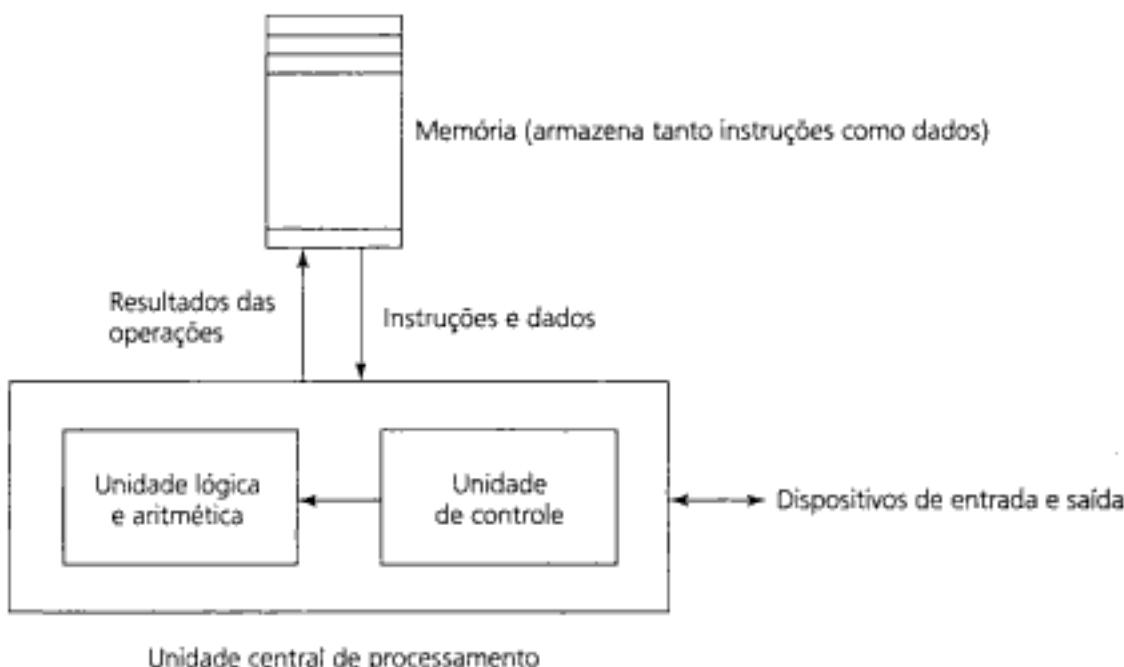
Além daqueles fatores descritos na Seção 1.3, vários outros influenciam no projeto básico das linguagens de programação. Os mais importantes são a arquitetura do computador e as metodologias de projeto do programa.

### 1.4.1 Arquitetura do Computador

A arquitetura básica dos computadores exerceu um efeito crucial sobre o projeto das linguagens. A maioria das mais populares dos últimos 45 anos foi projetada em função da arquitetura de computador prevalecente, chamada de arquitetura von Neumann, em homenagem a um de seus criadores, John von Neumann (pronuncia-se “fon Nóiman”). Essas linguagens são chamadas de *imperativas*. Em um computador de von Neumann, tanto os dados como os programas são armazenados na mesma memória. A unidade central de processamento (UCP), que executa realmente as instruções, é separada da memória. Portanto, as instruções e os dados devem ser canalizados (*piped*) ou transmitidos da memória para a UCP. Os resultados das operações na UCP devem ser novamente transferidos para a memória. Quase todos os computadores digitais construídos desde a década de 40 têm se baseado na arquitetura von Neumann. A estrutura global de um computador de von Neumann é mostrada na Figura 1.1 (ver p. 34).

Por causa da arquitetura von Neumann, os recursos centrais das linguagens imperativas são as variáveis, as quais modelam as células de memória, as instruções de atribuição, baseadas na operação de canalização (*piping*) e a forma iterativa de repetição, o método mais eficiente dessa arquitetura. Os operandos das expressões são canalizados da memória para a UCP, e o resultado da avaliação daquela é canalizado de volta para a célula da memória representada pelo lado esquerdo da atribuição. A iteração é rápida nos computadores de von Neumann porque as instruções são armazenadas em células adjacentes da memória. Essa eficiência desencoraja o uso da recursão para repetição, embora ela freqüentemente seja mais natural.

Conforme foi declarado anteriormente, uma linguagem funcional ou aplicativa é aquela cujo principal meio de fazer computações é aplicando funções a determinados parâmetros. A programação pode ser feita em uma linguagem funcional sem o tipo de variáveis usadas nas imperativas, sem instruções de atribuição e sem iteração. Ainda que muitos cientistas da computação tenham feito exposições sobre os inúmeros benefícios das linguagens funcionais,



**FIGURA 1.1** A arquitetura do computador de von Neumann.

como o LISP, é improvável que eles deixem de lado as imperativas até que um computador não-von Neumann seja projetado, permitindo a execução eficiente de programas em linguagens funcionais. Entre aqueles que lastimam tal fato, o mais eloquente tem sido John Backus, o projetista-chefe da versão original do FORTRAN (Backus, 1978).

As máquinas de arquitetura paralela que surgiram nos últimos 20 anos apresentam alguma promessa de agilizar a velocidade de execução de programas funcionais, mas, até agora, isso não tem sido suficiente para torná-las competitivas com programas imperativos. Aliás, ainda que existam maneiras elegantes de usar arquiteturas paralelas para executar programas funcionais, a maioria dessas é usada para programas imperativos, especialmente os escritos em dialetos do FORTRAN.

#### 1.4.2 Metodologias de Programação

O final da década de 60 e o início da década de 70 trouxeram uma análise intensa, em grande parte iniciada pelo movimento de programação estruturada, tanto do processo de desenvolvimento de software como do projeto de linguagens de programação.

Uma razão importante para essa pesquisa foi a mudança no importante custo de computação do hardware para o software, uma vez que os custos daquele diminuíram e os custos dos programadores se elevaram. Os aumentos de produtividade do programador foram relativamente pequenos. Além disso, problemas progressivamente maiores e mais complexos passaram a ser resolvidos por computador. Em vez de simplesmente resolver

conjuntos de equações para simular rastreamento de satélites, como no início da década de 60, passaram a ser feitos programas para tarefas grandes e complexas, como, por exemplo, controlar grandes instalações de refinaria de petróleo e oferecer sistemas de reservas de passagens aéreas no mundo inteiro.

As novas metodologias de desenvolvimento de software surgiram em consequência das pesquisas da década de 70 e foram chamadas de projeto *top-down* e refinamento passo a passo. As principais deficiências descobertas nas linguagens de programação foram a não-plenitude de verificação de tipos e a insuficiência das instruções de controle (que exigiam o uso excessivo de *gotos*).

No final dessa década, iniciou-se uma mudança das metodologias de projeto de programas orientadas para o processo para as orientadas a dados. Colocando de maneira simples, os métodos orientados a dados enfatizam o projeto de dados, concentrando-se no uso de tipos de dados abstratos para resolver problemas.

Para que a abstração de dados seja usada eficientemente no projeto de sistemas de software, ela deve ser suportada pelas linguagens voltadas para implementação. A primeira a oferecer um limitado suporte para abstração de dados foi a SIMULA 67 (Birtwistle et al., 1973), embora ela certamente não tenha sido levada à popularidade por causa disso. Os benefícios da abstração de dados não foram amplamente reconhecidos até o início da década de 70. Porém, a maioria das linguagens projetadas desde o final daquela década suporta abstração de dados que será discutida detalhadamente no Capítulo 11.

Os passos mais recentes na evolução do desenvolvimento de software orientado a dados, que se iniciou em meados da década de 80, é o projeto orientado a objeto. A metodologia orientada a objeto inicia-se com a abstração de dados, a qual encapsula o processamento com objetos de dados e oculta o acesso a eles, e adiciona herança e vinculação dinâmica de métodos. Herança é um conceito poderoso que aumenta muito a reutilização potencial dos softwares existentes, oferecendo, portanto, a possibilidade de significativos aumentos na produtividade de desenvolvimento de software. Esse é um fator importante no aumento de popularidade das linguagens orientadas a objeto. A vinculação dinâmica (tempo de execução) de métodos permite um uso mais flexível da herança.

A programação orientada a objeto desenvolveu-se juntamente com uma linguagem que suportou seus conceitos: a Smalltalk (Goldberg e Robson, 1989). Embora a Smalltalk não tenha sido tão usada como algumas outras linguagens, o suporte para programação orientada a objeto, agora, faz parte das linguagens de programação mais populares, inclusive a Ada 95 (AARM, 1995), o Java e o C++. Os conceitos orientados a objeto também chegaram à programação funcional em CLOS (Bobrow et al., 1988) e programação lógica em Prolog++. O suporte à linguagem para programação orientada a objeto será discutido detalhadamente no Capítulo 12.

A programação orientada ao processo é, em certo sentido, o oposto da programação orientada a dados. Mesmo que os métodos orientados a dados agora dominem o desenvolvimento de software, os métodos orientados ao processo não foram abandonados. Ao contrário, tem ocorrido muita pesquisa em programação orientada a processo nos últimos anos, especialmente na área da concorrência. Esses esforços de pesquisa trouxeram consigo a necessidade de facilidades de linguagem para criar e para controlar unidades de programa concorrentes. A Ada e o Java incluem essas capacidades. A concorrência será discutida detalhadamente no Capítulo 13.

## 1.5 Categorias de Linguagem

As linguagens de programação, muitas vezes, são categorizadas em quatro caixas: imperativas, funcionais, lógicas e orientadas a objeto. Já discutimos as características das linguagens imperativas e funcionais. Também descrevemos como as linguagens de programação orientadas a objeto mais populares desenvolveram-se a partir das imperativas. Apesar do paradigma de desenvolvimento de software orientado a objeto diferir bastante do paradigma orientado para procedimentos normalmente usado com as linguagens imperativas, as extensões a uma linguagem imperativa necessárias para suportar a programação orientada a objeto não são opressivas. Por exemplo, as expressões, as instruções de atribuição e as instruções de controle do C e do Java são quase idênticas; por outro lado, as matrizes, os subprogramas e a semântica da linguagem Java são muito diferentes em C.

Uma linguagem de programação lógica é um exemplo de linguagem baseada em regras. Em uma linguagem imperativa, um algoritmo é especificado com grandes detalhes, e a ordem de execução específica das instruções ou dos comandos deve ser incluída. Em uma linguagem baseada em regras, estas são especificadas sem nenhuma ordem particular, e o sistema de implementação deve escolher uma ordem de execução que produza o resultado desejado. Essa abordagem ao desenvolvimento de software é radicalmente diferente daquelas usadas com os outros três tipos de linguagens, e, evidentemente, exige um tipo de linguagem completamente diferente. O Prolog, a mais popular linguagem de programação lógica usada, e a programação lógica serão discutidas no Capítulo 16.

As linguagens de marcação<sup>\*</sup> (*markup*), como a HTML, às vezes, são confundidas com as de programação. Porém, as linguagens de marcação não especificam computações; ao contrário, elas descrevem a aparência geral de documentos. Porém, muitos dos critérios de projeto e de avaliação descritos neste capítulo também se aplicam às linguagens de marcação. Afinal de contas, é evidentemente importante que o código de marcação seja fácil de escrever e de ler. Entretanto, não discutiremos as linguagens de marcação neste livro.

## 1.6 Custo/Benefício no Projeto da Linguagem

Os critérios de avaliação das linguagens de programação descritos na Seção 1.3 fornecem uma estrutura para o projeto de linguagens. Infelizmente, essa estrutura é contraditória. Em seu precioso artigo sobre projeto de linguagens, Hoare (1973) afirma que “existem tantos critérios importantes, mas conflitantes, que a reconciliação e a satisfação dos mesmos é uma grande tarefa de engenharia”.

Dois critérios conflitantes são a confiabilidade e o custo de execução. Por exemplo, a definição da linguagem Ada exige que todas as referências aos elementos de matrizes sejam verificados para assegurar que o índice ou os índices estejam em suas faixas legais. Isso aumenta o custo de execução de programas Ada que contenham um grande número de referências aos elementos das matrizes. O C não exige verificação da faixa de índice, de modo que os programas em C executam mais rapidamente do que programas em Ada semanticamente equivalentes, não obstante os programas em Ada sejam mais confiáveis. Os projetistas da Ada trocaram a eficiência de execução pela confiabilidade.

<sup>\*</sup>N. de T. Marcação: *markup*. Um padrão para a definição de vínculos de hipertexto entre documentos.

Como outro exemplo de critérios conflitantes que levam diretamente a compensações no projeto, consideremos o caso da APL. Esta inclui um poderoso conjunto de operadores para manipular matrizes. Por causa do grande número de operadores, um número significativo de novos símbolos teve de ser incluído na APL para representá-los. Além disso, muitos operadores APL podem ser usados em uma única expressão longa e complexa. Um resultado desse elevado grau de expressividade é que, para aplicações que envolvem muitas operações com matrizes, a APL possui grande capacidade de escrita. De fato, uma enorme quantidade de computação pode ser especificada em um programa muito compacto. Outro resultado é que os programas em APL têm uma legibilidade muito ruim. Uma expressão compacta e concisa tem certa beleza matemática, mas é difícil para qualquer outra pessoa entender, a não ser para quem a escreveu. O conhecido autor Daniel McCracken observou certa vez que demorava quatro horas para ler um programa APL de quatro linhas (McCracken, 1970). Os projetistas da APL trocaram a legibilidade por capacidade de escrita.

O conflito entre flexibilidade e segurança é comum no projeto de uma linguagem. Os registros variantes em Pascal permitem que uma célula de memória contenha diferentes valores de tipo em diferentes momentos. Por exemplo, uma célula pode conter ou um ponteiro, ou um número inteiro. Então, pode-se fazer uma operação sobre um valor de ponteiro colocado nesse tipo de célula como se ele fosse um número inteiro, usando-se qualquer operação definida para valores inteiros. Isso fornece uma maneira de burlar a verificação de tipos da linguagem Pascal, que permite que um programa faça operações aritméticas nos ponteiros, o que pode ser, às vezes, conveniente. Porém, esse uso não-verificado das células de memória é, em geral, uma prática perigosa.

Abundam exemplos de conflitos entre os critérios de projeto (e de avaliação) de linguagens; alguns são sutis, outros são óbvios. Portanto, é evidente que a tarefa de escolher construções e recursos ao projetar uma linguagem de programação envolve uma coleção de meios-termos e de compensações.

## 1.7 Métodos de Implementação

Como foi descrito na Seção 1.4.1, dois dos principais componentes de um computador são sua memória interna e seu processador. A memória interna é usada para armazenar programas e dados. O processador é um conjunto de circuitos que garante a realização de um conjunto de operações primitivas, ou de instruções de máquina, como aquelas para operações aritméticas e lógicas. Na maioria dos computadores, algumas dessas instruções, que, às vezes, são chamadas de macroinstruções, são de fato implementadas com um conjunto de instruções de nível ainda mais baixo, chamadas microinstruções. Uma vez que as microinstruções nunca são vistas pelo software ou pelos programadores, usualmente elas não são incluídas em qualquer discussão sobre software. Portanto, elas não serão discutidas adicionalmente aqui.

A linguagem de máquina do computador é seu conjunto de macroinstruções. Na ausência de outro software de suporte, sua própria linguagem de máquina é a única que a maioria dos computadores “entendem”. Teoricamente, um computador pode ser projetado e construído com uma linguagem de alto nível particular como sua linguagem de máquina, mas seria muito complexo e caro. Além disso, ele seria altamente inflexível, porque seria difícil (ainda que não impossível) usá-lo com outras linguagens de alto nível. A opção de projeto de máquina mais prática implementa em hardware uma linguagem de nível muito

baixo que oferece as operações primitivas mais comumente necessárias e exige que o software de sistema crie uma interface com os programas de nível mais elevado.

Um sistema de implementação de linguagem não pode ser o único software em um computador. Também é necessário um grande conjunto de programas, chamado sistema operacional, o qual fornece primitivas de nível mais alto do que as da linguagem de máquina. Essas primitivas oferecem gerenciamento de recursos do sistema, operações de entrada e saída, um sistema de gerenciamento de arquivos, editores de texto e/ou programas, e uma variedade de outras funções comumente necessárias. Uma vez que os sistemas de implementação de linguagem precisam de muitas das facilidades do sistema operacional, eles se comunicam com o sistema operacional em vez de diretamente com o processador (na linguagem de máquina).

O sistema operacional e as implementações são dispostos em camadas sobre a interface da linguagem de máquina de um computador. Essas camadas podem ser imaginadas como computadores virtuais, que oferecem interfaces para o usuário em níveis mais altos. Por exemplo, um sistema operacional e um compilador C constituem um computador virtual C. Com outros compiladores, uma máquina pode transformar-se em outros tipos de computadores virtuais. A maioria dos computadores fornece diversas máquinas virtuais diferentes. Os programas de usuário formam outra camada no topo da camada das máquinas virtuais.

A visualização em camadas de um computador é mostrada na Figura 1.2 (ver p. 39).

Os sistemas de implementação das primeiras linguagens de programação de alto nível, construídos no final da década de 50, estavam entre os sistemas mais complexos daquele época. Na década de 60, esforços de pesquisa intensivos foram feitos para compreender e para formalizar o processo de construção dessas implementações de linguagem de alto nível. O maior sucesso desses esforços ocorreu na área de análise sintática, principalmente porque essa parte do processo de implementação é uma aplicação de partes da teoria dos autômatos<sup>\*</sup> e da teoria da linguagem formal que eram, então, bem-entendidas.

### 1.7.1 Compilação

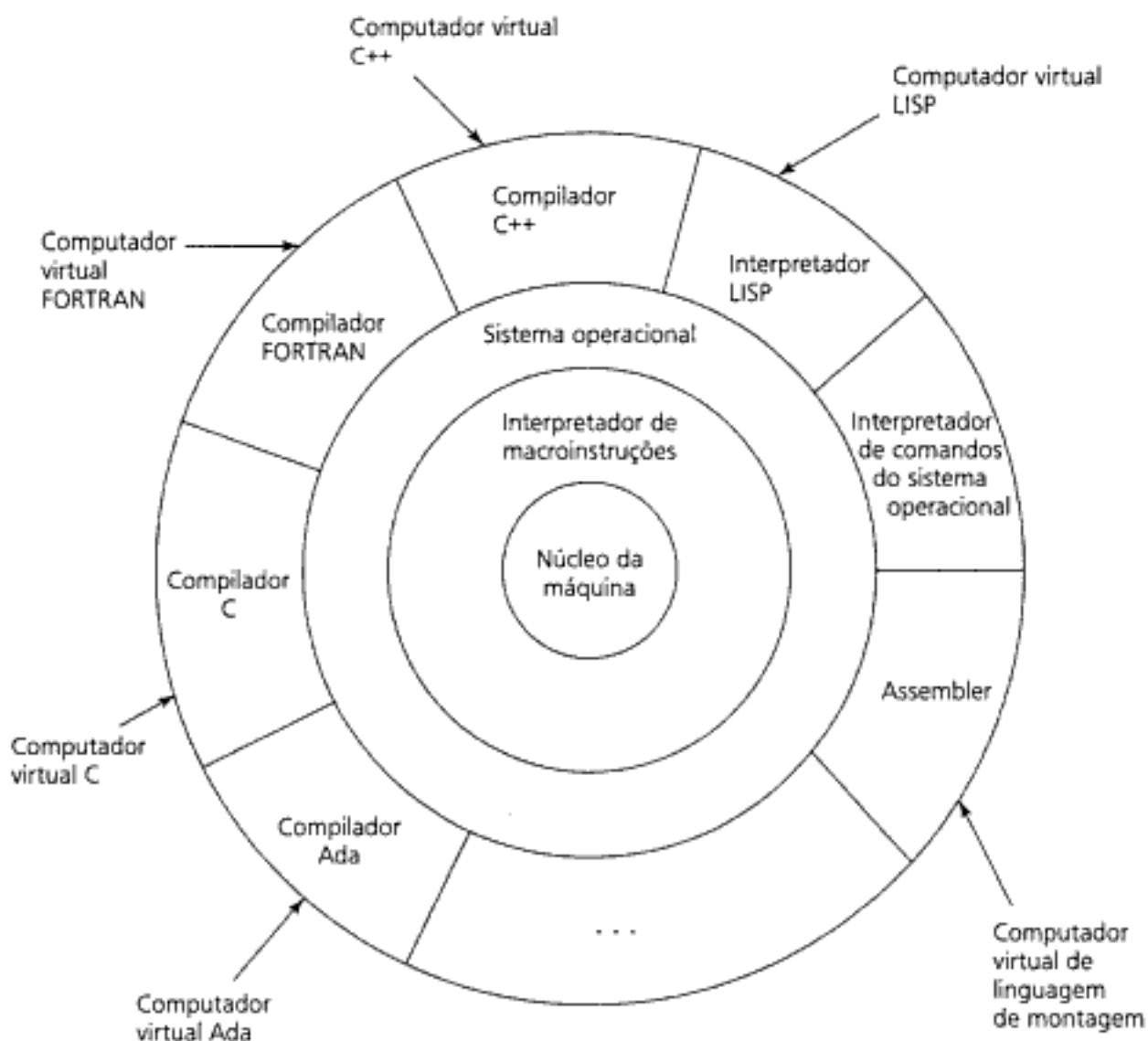
As linguagens de programação podem ser implementadas por meio de qualquer um dos três métodos gerais. Em um extremo, programas podem ser traduzidos para linguagem de máquina, a qual pode ser executada diretamente no computador. Isso é chamado de implementação **compilada**. Esse método tem a vantagem de uma execução de programa muito rápida, assim que o processo de tradução for concluído. A maioria das implementações de linguagens de produção como C, COBOL e Ada dá-se por meio de compiladores.

A linguagem que um compilador traduz é chamada de linguagem-fonte. O processo de compilação desenvolve-se em diversas etapas, sendo que as mais importantes são mostradas na Figura 1.3 (ver p. 40).

O analisador léxico reúne os caracteres do programa-fonte em unidades léxicas que são os identificadores, as palavras especiais, os operadores e os símbolos de pontuação. O analisador léxico ignora os comentários no programa-fonte, porque eles não têm nenhuma utilidade para o compilador.

O analisador sintático pega as unidades do analisador léxico e usa-as para construir estruturas hierárquicas chamadas árvores de análise (*parse trees*), as quais representam a

<sup>\*</sup>N. de T. Teoria dos autômatos: uma disciplina da ciência da computação aberta que diz respeito a um dispositivo abstrato denominado "autômato", o qual executa uma função específica computacional ou de reconhecimento.

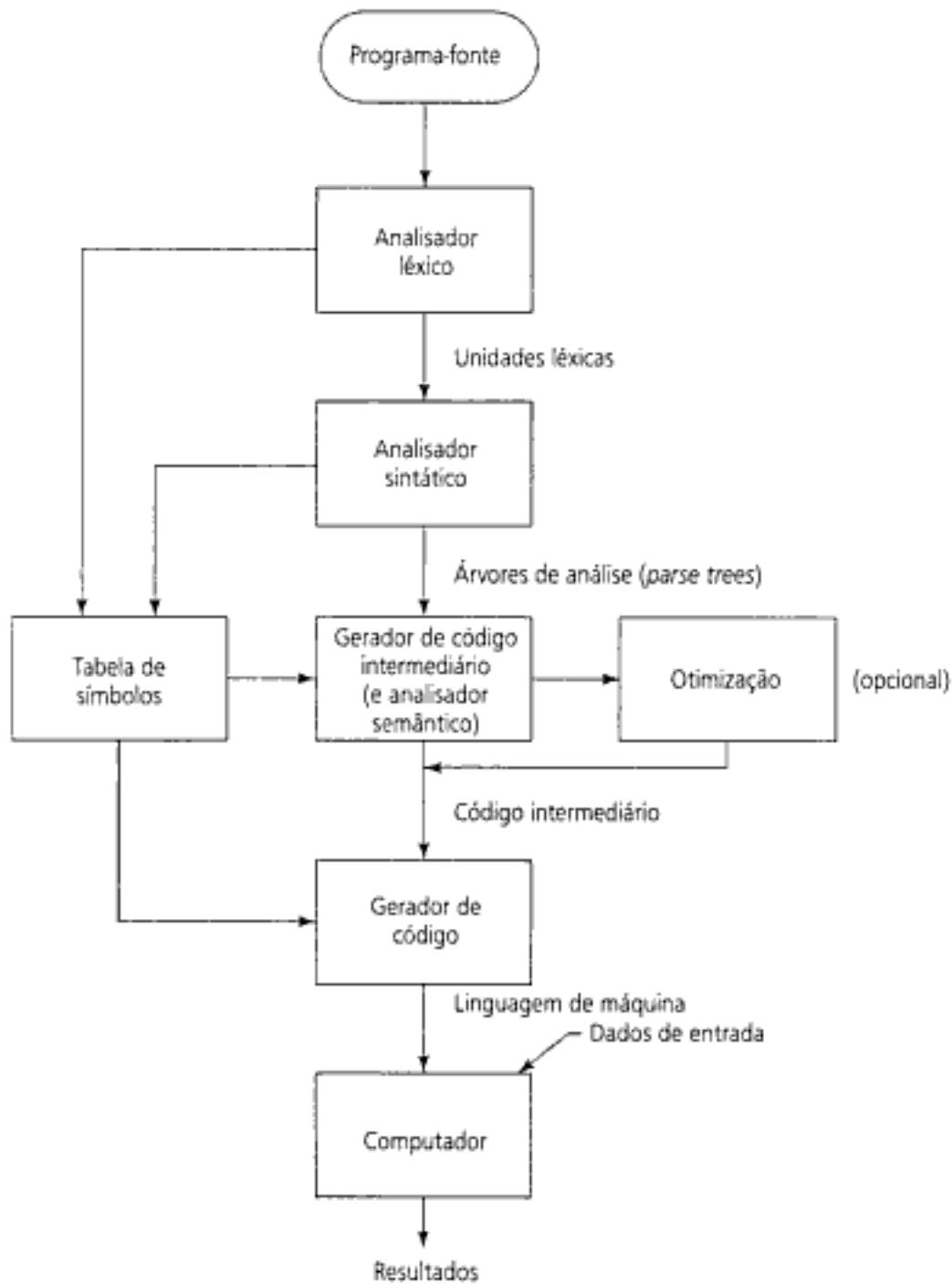


**FIGURA 1.2** Interface de computadores virtuais disposta em camadas, fornecida por um computador típico.

estrutura sintática do programa. Em muitos casos, nenhuma estrutura de árvore de análise real é construída; ao contrário, a informação que seria necessária para construí-la é gerada e usada. Tanto as unidades léxicas como as árvores de análise serão discutidas adicionalmente no Capítulo 3. A análise sintática é discutida no Capítulo 4.

O gerador de código intermediário produz um programa em uma linguagem diferente, no nível intermediário entre o programa-fonte e a saída final do compilador, o programa em linguagem de máquina (note que as palavras *linguagem* e *código* muitas vezes são usadas de maneira intercambiável). As linguagens intermediárias, às vezes, parecem-se muito com as linguagens de montagem e, de fato, algumas vezes são linguagem de montagem. Em outros casos, o código intermediário está em um nível bem mais alto do que o de montagem. O analisador semântico faz parte integralmente do gerador de código intermediário e verifica se há erros difíceis, se não impossíveis, de serem detectados durante a análise sintática, como por exemplo, erros de tipo.

Tanto a análise léxica como a análise sintática são discutidas em detalhe no Capítulo 4.

**FIGURA 1.3** O processo de compilação.

A otimização, que melhora os programas tornando-os menores ou mais rápidos, ou ambos, muitas vezes, é uma parte opcional da compilação. De fato, alguns compiladores são incapazes de fazer qualquer otimização significativa. Esse tipo de compilador seria usado em situações em que a velocidade de execução do programa traduzido é bem menos importante do que a velocidade de compilação. Um exemplo dessa situação é um laboratório de computação para programadores principiantes. Na maioria das situações comerciais e industriais, a velocidade de execução é mais importante do que a velocidade de compila-

ção, de modo que a otimização é rotineiramente desejável. Uma vez que muitos tipos de otimização não podem ser feitos em linguagem de máquina, a maioria das otimizações é feita no código intermediário.

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

A tabela de símbolos serve como um banco de dados para o processo de compilação. Seu principal conteúdo são informações sobre tipos e atributos de cada nome definido pelo usuário no programa. Essas informações são colocadas na tabela de símbolos pelos analisadores léxico e sintático e usadas pelo analisador semântico e pelo gerador de código.

Conforme afirmou-se acima, não obstante a linguagem de máquina gerada por um compilador possa ser executada diretamente no hardware, quase sempre ela deve ser executada juntamente com algum outro código. A maioria dos programas de usuário também exige programas do sistema operacional. Entre os mais comuns, estão aqueles para entrada e para saída de dados. O compilador cria chamadas a programas do sistema necessários quando o programa de usuário necessita deles. Antes que os programas em linguagem de máquina produzidos pelo compilador possam ser executados, os programas necessários do sistema operacional devem ser encontrados e vinculados ao do usuário. A operação de vinculação conecta o programa de usuário aos de sistema, colocando os endereços dos pontos de entrada dos programas de sistema nas chamadas a eles no de usuário. O código de usuário e o de sistema juntos, às vezes, são chamados de **módulo de carga ou imagem de executável**. O processo de coletar programas de sistema e vinculá-los aos programas de usuário é chamado de **vinculação e carregamento** ou, às vezes, apenas de **vinculação**. Ele é realizado por um programa de sistema chamado **linkeditor**.

Além dos programas de sistemas, os programas de usuário muitas vezes devem ser vinculados a programas de usuário compilados anteriormente, que residem em bibliotecas. Assim, o linkeditor não somente vincula algum dado programa aos programas de sistema, mas também o vincula a outros programas de usuário.

A execução de um programa em código de máquina em um computador com arquitetura von Neumann ocorre em um processo chamado **ciclo buscar-executar** (fetch-execute cycle). Como se afirmou na Seção 1.4.1, os programas residem na memória, mas são executados na UCP. Cada instrução a ser executada deve ser transferida da memória para o processador. O endereço da instrução seguinte a ser executada é mantido em um registrador chamado contador de programa. O ciclo buscar-executar pode ser descrito de maneira simples pelo seguinte algoritmo:

inicialize o contador de programa

**Repete** sempre

busque a instrução apontada pelo contador de programa

incremente o contador de programa para apontar para a instrução seguinte

decodifique a instrução

execute a instrução

**fim da repetição**

O passo “decodifique a instrução” no processo acima significa que esta é examinada para determinar a ação especificada por ela. A execução do programa encerra-se quando uma instrução de parada é encontrada, ainda que em um computador real uma instrução de parada raramente seja executada. Ao contrário, o controle transfere-se do sistema operacional para um programa de usuário para ser executado e depois retorna ao sistema operacional quando a execução completa-se. Em um computador em que mais de um programa de usuário pode estar na memória em determinado momento, esse processo é bem mais complexo.

A velocidade da conexão entre a memória de um computador e seu processador usualmente determina a velocidade do computador, porque, muitas vezes, as instruções podem ser executadas mais rapidamente do que podem ser transferidas para o processador para serem executadas. Essa conexão é chamada de **gargalo de von Neumann**; ele é o principal fator limitante na velocidade da arquitetura de computadores de von Neumann. O gargalo de von Neumann foi uma das principais motivações para a pesquisa e para o desenvolvimento de computadores paralelos.

### 1.7.2 Interpretação Pura

Na extremidade oposta dos métodos de implementação, os programas podem ser interpretados por outro programa chamado interpretador, sem nenhuma conversão. O programa interpretador age como uma simulação de software de uma máquina cujo ciclo buscar-executar lida com instruções de programa em linguagem de alto nível em vez de instruções de máquina. Essa simulação de software, evidentemente, fornece uma máquina virtual para a linguagem.

Essa técnica, chamada de **interpretação pura** ou, simplesmente, de interpretação, tem a vantagem de permitir uma fácil implementação de muitas operações de depuração do código-fonte, porque todas as mensagens de erro em tempo de execução podem referir-se a unidades do código. Por exemplo, se for considerado que um índice de matriz está fora da faixa, a mensagem de erro poderá facilmente indicar a linha da fonte e o nome da matriz. Por outro lado, esse método tem a séria desvantagem de que a execução é de 10 a 100 vezes mais lenta do que em sistemas compilados. A principal causa dessa lentidão é a decodificação das instruções de linguagem de alto nível, bem mais complexas do que as instruções em linguagem de máquina (embora possa haver um número menor de comandos do que de instruções em código de máquina equivalente). Portanto, a decodificação de comandos, em vez da conexão entre o processador e a memória, é o gargalo de um interpretador puro.

Outra desvantagem da interpretação pura é que ela freqüentemente exige mais espaço. Além do programa-fonte, a tabela de símbolos deve estar presente na interpretação.

Além disso, o programa-fonte deve ser armazenado em uma forma projetada para permitir fácil acesso e modificação, em vez de um tamanho mínimo.

A interpretação é um processo difícil em programas escritos em uma linguagem complicada, porque o significado de cada expressão e instrução deve ser determinado diretamente do programa-fonte em tempo de execução. Linguagens com estruturas mais simples prestam-se à interpretação pura. Por exemplo, a APL e a LISP, às vezes, são implementadas como sistemas interpretativos puros. A maioria dos comandos do sistema operacional, como por exemplo, o conteúdo dos scripts do shell do UNIX e dos arquivos .BAT do DOS, são implementados com interpretadores puros. Linguagens mais complexas, como o FORTRAN e o C, raramente



**FIGURA 1.4** Interpretação pura.

são implementadas com interpretadores puros. Embora não seja uma linguagem particularmente simples, o JavaScript é puramente interpretado.

O processo de interpretação pura é mostrado na Figura 1.4 (ver página anterior).

### 1.7.3 Sistemas de Implementação Híbridos

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

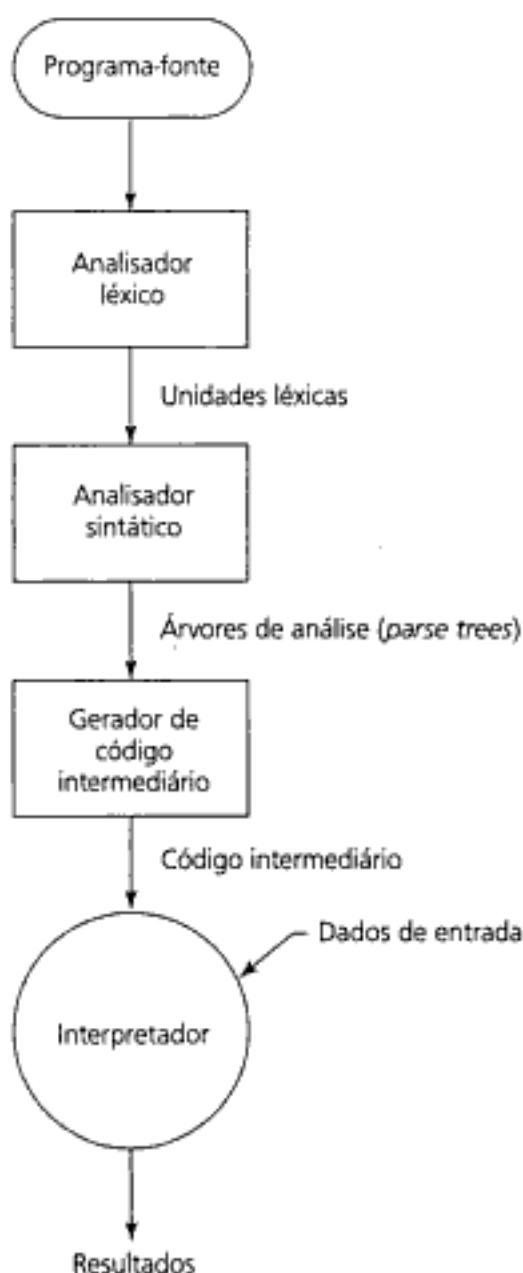
linguagem intermediária projetada para permitir fácil interpretação. Esse método é mais rápido do que a interpretação pura porque as instruções da linguagem fonte são decodificadas somente uma vez. Essas implementações são chamadas de **sistemas de implementação híbridos**.

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

A Perl é implementada com um sistema híbrido. Ela se desenvolveu a partir das linguagens interpretativas sh e awk, mas é parcialmente compilada para detectar erros antes da interpretação e para simplificar o interpretador.

As implementações iniciais de Java eram todas híbridas. Sua forma intermediária, chamada código de bytes, oferece portabilidade a qualquer máquina que tenha um interpretador de código de bytes e um sistema associado em tempo de execução. Juntos, eles são chamados de Máquina Virtual Java. Agora há sistemas que traduzem código de bytes Java para código de máquina para permitir uma execução mais rápida. Porém, os *applets*\* Java são sempre descarregados do servidor Web na forma de código de bytes.

Às vezes, um implementador pode oferecer tanto implementações compiladas como interpretadas para uma linguagem. Nesses casos, o interpretador é usado para desenvolver e para depurar programas. Então, depois que um estado (relativamente)



**FIGURA 1.5** Sistema de implementação híbrido.

\*N. de T. Applets: pequenas aplicações.

livre de erros é alcançado, os programas são compilados para aumentar sua velocidade de execução.

## 1.8 Ambientes de Programação

---

Um ambiente de programação é o conjunto de ferramentas usadas no desenvolvimento de software. Esse conjunto pode consistir em somente um sistema de arquivos, em um editor de texto, em um *linkeditor* e em um compilador. Ou pode incluir uma grande coleção de ferramentas integradas, cada uma das quais acessada por meio de uma interface uniforme. Nesse último caso, o desenvolvimento e a manutenção de software melhoram bastante. Portanto, as características de uma linguagem de programação não são a única medida da capacidade de desenvolvimento de software de um sistema. Descreveremos agora, brevemente, diversos ambientes de programação.

O UNIX é um dos mais antigos ambientes de programação, distribuído pela primeira vez na década de 70, construído em torno de um sistema operacional portável com compartilhamento de tempo (*time-sharing*). Ele fornece um amplo conjunto de poderosas ferramentas de apoio para a produção e para a manutenção de software em uma variedade de linguagens. No passado, o mais importante recurso ausente do UNIX era uma interface uniforme entre suas ferramentas. Isso o tornava difícil de aprender e de usar. Porém, o UNIX, agora, é freqüentemente usado por meio de uma interface gráfica que roda em cima dele. Um exemplo é o *Common Desktop Environment* (CDE).

O Borland C++ é um ambiente de programação que roda em microcomputadores IBM-PC e compatíveis. Ele oferece um compilador integrado, um editor, um depurador e um sistema de arquivos, em que todos os quatro são acessados por meio de uma interface gráfica. Um recurso conveniente desse tipo de ambiente é que, quando o compilador encontra um erro de sintaxe, ele pára e muda para o editor, deixando o cursor no ponto do programa-fonte em que o erro foi detectado.

O Smalltalk é uma linguagem e um ambiente de programação integrados, mas ela é mais elaborada do que o Borland C++. O Smalltalk foi o primeiro a fazer uso de um sistema de janelas e um dispositivo de indicação por mouse para oferecer ao usuário uma interface uniforme a todas as ferramentas. O Smalltalk será discutido no Capítulo 12.

O passo mais recente na evolução dos ambientes de desenvolvimento de software é representado pelo Microsoft Visual C++. É uma grande e elaborada coleção de ferramentas de desenvolvimento de software, todas usadas por uma interface provida de janelas. Esse sistema, juntamente com outros sistemas similares como o Visual BASIC, o Delphi e o JBuilder da Borland, oferecem maneiras simples de construir interfaces gráficas para os programas.

É evidente que a maior parte do desenvolvimento de software, pelo menos no futuro próximo, fará uso de ambientes de programação poderosos. Isso, sem dúvida, aumentará a produtividade do software e talvez elevará a sua qualidade de produção.

## RESUMO

---

O estudo das linguagens de programação é valioso por uma série de importantes razões: aumenta nossa capacidade de usar diferentes construções para escrever programas, possibilita-nos escolher linguagens para projetos de maneira mais inteligente e torna mais fácil a aprendizagem de novas linguagens.

Os computadores são usados em uma ampla variedade de domínios de solução de problemas. O projeto e a avaliação de uma linguagem de programação particular dependem muito do domínio em que eles devem ser usados.

Entre os critérios mais importantes para avaliar linguagens estão a legibilidade, a capacidade de escrita, a confiabilidade e o custo global. Esses serão a base na qual examinaremos e julgaremos os vários recursos de linguagem discutidos no restante do livro.

As principais influências sobre o projeto de uma linguagem têm sido a arquitetura de máquina e as metodologias de projeto de software.

Projetar uma linguagem de programação é, antes de mais nada, um feito de engenharia, na qual uma longa lista de compensações deve ser feita entre os recursos, as construções e as capacidades.

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

Os ambientes de programação tornaram-se partes importantes dos sistemas de desenvolvimento de software, nos quais a linguagem é apenas um dos componentes.

## QUESTÕES DE REVISÃO

1. Por que é útil que o programador tenha algum embasamento em projeto de linguagens, ainda que talvez ele jamais projete de fato uma linguagem de programação?
2. Como o conhecimento das características da linguagem de programação pode beneficiar toda a comunidade de computação?
3. Que linguagem de programação dominou a computação científica ao longo dos últimos 40 anos?
4. Que linguagem de programação dominou as aplicações comerciais ao longo dos últimos 40 anos?
5. Que linguagem de programação dominou a inteligência artificial ao longo dos últimos 40 anos?
6. Em qual linguagem o UNIX foi escrito?
7. Qual é a desvantagem de haver demasiados recursos em uma linguagem?
8. Como uma sobrecarga de operador definida pelo usuário prejudica a legibilidade de um programa?
9. Que exemplo pode ilustrar a falta de ortogonalidade no projeto do C?
10. Qual linguagem usou a ortogonalidade como principal critério de projeto?
11. Qual instrução de controle primitiva é usada para construir instruções de controle mais complicadas em linguagens em que elas não existem?
12. Qual problema de legibilidade é causado quando se usa a mesma palavra reservada de fechamento para mais de um tipo de instrução de controle?
13. Qual construção de uma linguagem de programação oferece abstração de processo?
14. O que significa um programa ser confiável?
15. Por que a verificação dos tipos e parâmetros de um subprograma é importante?
16. O que é apelido?
17. O que é manipulação de exceções?
18. Por que a legibilidade é importante para a capacidade de escrita?
19. Qual é o custo dos compiladores para determinada linguagem em relação ao projeto dessa linguagem?
20. Qual tem sido a mais forte influência no projeto de linguagens de programação ao longo dos últimos 45 anos?
21. Qual é o nome da categoria de linguagens de programação cuja estrutura é determinada pela arquitetura de computador de von Neumann?
22. Quais foram as duas deficiências de linguagem de programação descobertas em consequência da pesquisa em desenvolvimento de software na década de 70?
23. Quais são os três recursos fundamentais de uma linguagem de programação orientada a objeto?

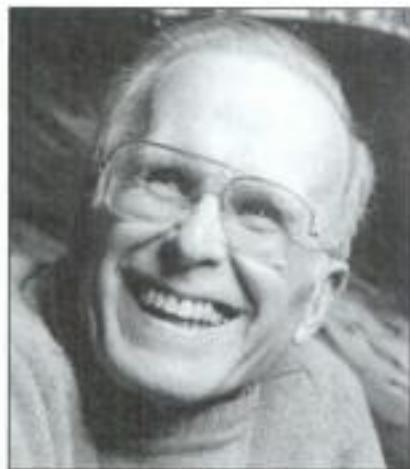
24. Qual linguagem foi a primeira a suportar os três recursos fundamentais da programação orientada a objeto?
25. Dê um exemplo de dois critérios de projeto de linguagem que estão em conflito direto um com o outro.
26. Quais são os três métodos gerais para implementar uma linguagem de programação?
27. O que produz uma execução de programa mais rápida: um compilador ou um interpretador puro?
28. Qual papel a tabela de símbolos desempenha em um compilador?
29. O que um linkeditor faz?
30. Por que o gargalo de von Neumann é importante?
31. Quais são as vantagens de implementar uma linguagem com um interpretador puro?
32. Qual desvantagem o UNIX tem como ambiente de desenvolvimento de software?

## ***PROBLEMAS***

1. Você acredita que nossa capacidade de pensar é influenciada por nossa linguagem? Sustente sua opinião.
2. Quais recursos de linguagens de programação específicas você conhece, cujos fundamentos lógicos são um mistério para você?
3. Quais argumentos você poderia levantar a favor da idéia de uma única linguagem para todos os domínios de programação?
4. Quais argumentos você poderia levantar contra a idéia de uma única linguagem para todos os domínios de programação?
5. Cite e explique outro critério pelo qual as linguagens podem ser julgadas (além daqueles apresentados neste capítulo).
6. Qual instrução de linguagem de programação comum, em sua opinião, é a mais prejudicial para a legibilidade?
7. O Modula-2 usa END para marcar o final de todas as instruções compostas. Quais os argumentos que podem ser levantados contra tal uso?
8. Algumas linguagens, notavelmente o C e o Java, fazem distinção entre maiúsculas e minúsculas nos identificadores. Quais são os prós e os contras nessa decisão de projeto?
9. Explique os diferentes aspectos do custo de uma linguagem de programação.
10. Quais são os argumentos para escrever programas eficientes não obstante o hardware ser relativamente barato?
11. Descreva algumas relações custo/benefício de projeto entre eficiência e segurança em algumas linguagens que você conhece.
12. Quais os principais recursos que uma linguagem de programação perfeita incluiria, em sua opinião?
13. A primeira linguagem de programação de alto nível que você aprendeu foi implementada com um interpretador puro, com um sistema de implementação híbrido ou com um compilador? (Talvez você não saiba isso sem pesquisa.)
14. Descreva as vantagens e as desvantagens de algum ambiente de programação que você usou.
15. Como as instruções de declaração de tipo para variáveis simples afetam a legibilidade de uma linguagem, considerando que algumas linguagens não as exigem?
16. Escreva uma avaliação de alguma linguagem de programação que conheça, usando os critérios descritos neste capítulo.
17. O Pascal usa o ponto e vírgula para separar instruções, enquanto o C usa-o para finalizar instruções. Qual dessas, em sua opinião, é a mais natural e a que tem menos probabilidade de resultar em erros de sintaxe? Sustente sua resposta.
18. Algumas linguagens, como o Pascal e o C, usam delimitadores em ambas as extremidades dos comentários. Outras linguagens, como o FORTRAN e a Ada, usam um símbolo ou um par de símbolos para indicar o início de um comentário e o final da linha para finalizá-lo. Discuta as vantagens e as desvantagens de cada opção de projeto com respeito aos nossos critérios.

## Capítulo 2

# Evolução das Principais Linguagens de Programação



### John Backus

John Backus, empregado da IBM, projetou o sistema de pseudocódigos Speedcoding para o computador 701 da IBM no início da década de 50. Entre 1954 e 1957, ele chefiou a equipe de projeto que produziu o FORTRAN, a partir do qual quase todas as linguagens imperativas desenvolveram-se. No período de 1958–1960, ele foi membro da equipe de projeto do ALGOL.

- 2.1** A Linguagem Plankalkül de Zuse
- 2.2** Programação de Hardware Mínima: Pseudocódigos
- 2.3** O IBM 704 e o FORTRAN
- 2.4** Programação Funcional: LISP
- 2.5** O Primeiro Passo Rumo à Sofisticação: ALGOL 60
- 2.6** Informatizando Registros Comerciais: COBOL
- 2.7** O Início do Compartilhamento de Tempo (*timesharing*): BASIC
- 2.8** Tudo para Todos: PL/I
- 2.9** Duas Primeiras Linguagens Dinâmicas: APL e SNOBOL
- 2.10** A Origem da Abstração de Dados: SIMULA 67
- 2.11** Projeto Ortogonal: ALGOL 68
- 2.12** Algumas Descendentes Importantes dos ALGOLs
- 2.13** Programação Baseada na Lógica: Prolog
- 2.14** O Maior Esforço de Projeto da História: Ada
- 2.15** Programação Orientada a Objeto: Smalltalk
- 2.16** Combinando Recursos Imperativos e Orientados a Objeto: C++
- 2.17** Programando a World Wide Web: Java

Este capítulo segue cronologicamente o desenvolvimento de uma coleção de linguagens de programação, explorando o ambiente em que cada uma foi projetada e se concentrando nas contribuições da linguagem e na motivação para seu desenvolvimento. Descrições globais da linguagem não serão incluídas; pelo contrário, discutiremos somente os novos recursos introduzidos por cada uma. De especial interesse são os recursos que mais influenciaram as linguagens subsequentes ou o campo da ciência da computação.

Este capítulo não inclui uma discussão em profundidade de quaisquer recursos ou conceitos de linguagens; isso é deixado para capítulos posteriores. Explicações breves e informais dos recursos bastarão para nossa jornada pelo desenvolvimento dessas linguagens.

A escolha de quais linguagens discutirmos aqui foi subjetiva, e muitos leitores notarão, insatisfeitos, a ausência de uma ou mais de suas favoritas. Porém, para mantermos a cobertura histórica em um tamanho razoável, foi necessário omitirmos diversas linguagens pelas quais algumas pessoas têm a mais alta consideração. As escolhas basearam-se em nossa estimativa da importância de cada uma para o desenvolvimento das linguagens e do mundo da computação como um todo. Também incluímos breves discussões de algumas destas que serão referenciadas posteriormente no livro.

Este capítulo inclui listagens de 11 exemplos de programas completos, cada um em uma linguagem diferente. Nenhum dos programas será descrito neste capítulo; eles pretendem simplesmente ilustrar o surgimento de programas nessas linguagens. Os leitores familiarizados com qualquer uma das linguagens imperativas comuns devem ser capazes de ler e de entender a maior parte do código existente nos programas, exceto aqueles em LISP, em COBOL e em Smalltalk. O exemplo em LISP será discutido no Capítulo 15; o exemplo em Smalltalk será discutido no Capítulo 12. O mesmo problema será resolvido por programas em FORTRAN, em ALGOL 60, em PL/I, em BASIC, em Pascal, em C, em Ada e em Java.

A Figura 2.1 (ver p. 49) é um gráfico da genealogia das linguagens de alto nível discutidas neste livro.

## 2.1 A Linguagem Plankalkül de Zuse

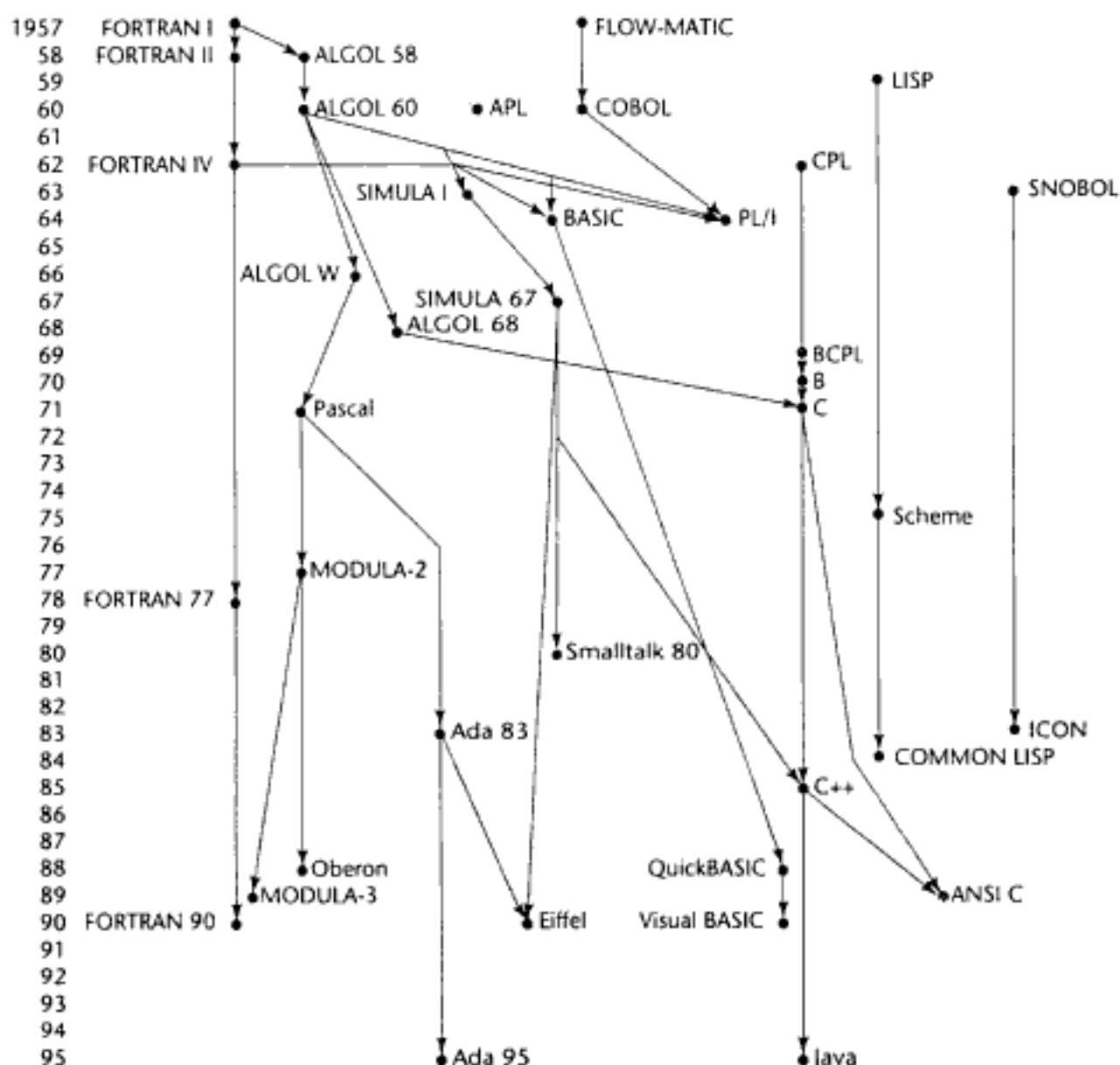
---

A primeira linguagem de programação discutida neste capítulo é altamente incomum sob vários aspectos. Por um lado, ela nunca foi implementada. Além disso, não obstante ter sido desenvolvida em 1945, sua descrição somente foi publicada em 1972. Em consequência da ignorância geral a respeito da linguagem, algumas de suas capacidades não apareceram em outras linguagens até 15 anos depois do desenvolvimento da Plankalkül.

### 2.1.1 Embasamento Histórico

Entre 1936 e 1945, o cientista alemão Konrad Zuse (pronuncia-se “Tzu-ze”) construiu uma série de computadores complexos e sofisticados a partir de relés eletromecânicos. Em 1945, a guerra destruiu todos, a não ser um de seus últimos modelos, o Z4; então, ele se mudou para uma remota aldeia bávara, Hinterstein, e os membros de seu grupo de pesquisas seguiram caminhos distintos.

Trabalhando sozinho, Zuse embarcou em um esforço para desenvolver uma linguagem para expressar computações, um projeto que ele iniciou em 1943 como proposta para sua dissertação de Ph.D. Ele chamou sua linguagem de Plankalkül, que significa cálculo de



**FIGURA 2.1** Genealogia das linguagens de programação de alto nível comuns.

programa. Em um manuscrito extenso datado de 1945, mas não-publicado até 1972 (Zuse, 1972), ele definiu a Plankalkül e escreveu algoritmos na linguagem para uma ampla variedade de problemas.

### 2.1.2 Visão Geral da Linguagem

A Plankalkül era notavelmente completa, com alguns de seus recursos mais avançados na área das estruturas de dados. O tipo mais simples era o bit único. A partir dele eram construídos tipos para números inteiros e reais. O tipo real usava uma notação de complemento de dois e o esquema de “bit oculto” atualmente usado para evitar armazenar o bit mais significativo da parte fracionária normalizada de um valor.

Além desses tipos escalares comuns, a Plankalkül incluía matrizes e registros. Os registros podiam usar recursão para incluir outros registros como elementos.

Ainda que a linguagem não tivesse nenhum *goto* explícito, ela incluía uma instrução iterativa semelhante ao **for** da Ada. Ela também tinha o comando *Fin* com um sobreescrito que indicava um salto para fora de um número especificado de aninhamentos de laços de iteração ou para o início de um novo ciclo de iteração. A Plankalkül incluía uma instrução de seleção, mas não permitia uma cláusula *else*.

Um dos recursos mais interessantes dos programas de Zuse era a inclusão de expressões matemáticas mostrando as relações entre variáveis de programa. Tais expressões declaravam o que seria verdadeiro durante a execução nos pontos do código em que apareciam e que são muito similares às asserções usadas atualmente na linguagem de programação Eiffel (Meyer, 1992) e na semântica axiomática, a qual será discutida no Capítulo 3.

O manuscrito de Zuse continha programas de complexidade bem maior do que qualquer um escrito antes de 1945. Inclusos, havia programas para classificar arranjos de números, para testar a conectividade de determinado grafo; para executar operações com números inteiros e reais, inclusive raiz quadrada; e para executar análise sintática de fórmulas lógicas com parênteses e com operadores em seis diferentes níveis de precedência. Talvez o mais notável tenha sido suas 49 páginas de algoritmos para jogar xadrez, um jogo em que ele não era especialista.

Se um cientista da computação tivesse encontrado a descrição da Plankalkül de Zuse no início da década de 50, o único aspecto da linguagem que teria atrapalhado sua implementação conforme era definida teria sido a notação. Cada instrução consistia em duas ou três linhas de código. A primeira linha era muito semelhante às instruções das linguagens contemporâneas. A segunda linha, que era opcional, continha os índices da matriz na primeira linha. É interessante notar que o mesmo método de indicar subscritos foi usado por Charles Babbage nos programas para seu *Analytical Engine* em meados do século XIX. A última linha de cada instrução da Plankalkül continha nomes de tipos para as variáveis mencionadas na primeira linha. Essa notação é muito assustadora quando vista pela primeira vez.

A instrução de atribuição dada como exemplo a seguir, que atribui o valor da expressão **A (4) + 1 para A (5)**, ilustra essa notação. A linha rotulada com **V** é para subscritos, e a com **S** é para tipos de dados. Nesse exemplo, **1.n** significa um número inteiro de **n** bits:

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

Somente podemos especular sobre a direção que o projeto de linguagens de programação e de desenvolvimento de computadores poderiam ter tomado se o trabalho de Zuse tivesse sido amplamente conhecido em 1945 ou mesmo em 1950. Também é interessante considerarmos como seu trabalho poderia ter sido diferente caso ele o tivesse feito em um ambiente pacífico rodeado por outros cientistas, em vez de na Alemanha em 1945 em virtual isolamento.

## 2.2 Programação de Hardware Mínima: Pseudocódigos

Os computadores que se tornaram disponíveis no final da década de 40 e no início da década de 50 eram bem menos usáveis do que os de hoje. Além de serem lentos, pouco confiáveis e caros, e possuírem memórias extremamente pequenas, eram difíceis de programar por causa da falta de software de apoio.

Não havia nenhuma linguagem de programação de alto nível ou mesmo linguagens de montagem, de forma que a programação era feita em código de máquina, o que, além de tedioso, está propenso a erros. Entre seus problemas está o uso de códigos numéricos para especificar instruções. Por exemplo, uma instrução ADD (Adicionar) era especificada pelo código 14 em vez de um nome textual conotativo, mesmo que somente uma única letra. Isso torna difícil a leitura dos programas. Um problema mais sério é o endereçamento absoluto, o qual dificulta a sua modificação. Por exemplo, suponhamos que temos um programa em linguagem de máquina armazenado na memória. Muitas das instruções dele referem-se a outras de suas localizações internas, usualmente para referenciar dados ou para indicar os alvos de instruções ramificadas. Inserir uma instrução em qualquer posição do programa que não no final invalidará a justeza de todas as instruções que se referem ao endereço além do ponto de inserção, porque estes últimos devem ser aumentados para dar espaço à nova instrução. Para fazer a adição corretamente, todas aquelas instruções que se referem aos endereços seguintes à adição devem ser encontradas e modificadas. Um problema similar ocorre com a exclusão de uma instrução. Nesse caso, entretanto, as linguagens de máquina freqüentemente incluem uma instrução "no operation" ("nenhuma operação") que pode substituir as excluídas, evitando, assim, o problema.

Tais problemas-padrão ocorrem com todas as linguagens de máquina e foram as principais motivações para inventar os *assemblers* e as linguagens de montagem. Além disso, a maioria dos problemas de programação daquela época eram numéricos e exigiam operações aritméticas com números reais e indexação de algum tipo para permitir o uso conveniente de matrizes. Nenhuma dessas capacidades, entretanto, foi incluída na arquitetura dos computadores do final da década de 40 e do início da década de 50. As deficiências naturalmente levaram ao desenvolvimento de linguagens de nível bem mais elevado.

### 2.2.1 Short Code

A primeira dessas novas linguagens, chamada de Short Code, foi desenvolvida por John Mauchly, em 1949, para o computador BINAC. A Short Code foi posteriormente transferida para o computador UNIVAC I e, durante muitos anos, foi um dos principais meios de programar essas máquinas. Ainda que pouco se saiba sobre a Short Code original porque sua descrição completa jamais foi publicada, um manual de programação para a versão UNIVAC I sobreviveu (Remington-Rand, 1952). Pode-se presumir, com segurança, que as duas versões eram muito similares.

O UNIVAC I tinha palavras que consistiam em 72 bits, agrupados como 12 bytes de seis bits. A Short Code consistia em versões codificadas de expressões matemáticas que tinham de ser avaliadas. Os códigos eram valores pares de bytes, e a maioria das equações encaixava-se em uma palavra. Alguns desses códigos eram:

01 -	06	valor absoluto	1n	(n+2) éssima potência
02 )	07 +		2n	(n+2) éssima raiz
03 =	08	pausa	4n	se <= n
04 /	09 (		58	impressão e tab

As variáveis ou localizações de memória eram nomeadas com códigos de pares de bytes, uma vez que deveriam ser usadas como constantes. Por exemplo, X0 e Y0 poderiam ser variáveis. A instrução

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

seria codificada em uma palavra como 00 X0 03 20 06 Y0. A inicial 00 era usada como um *enchimento*\* (*padding*) para preencher a palavra. Curiosamente, não havia nenhum código de multiplicação; esta última era indicada simplesmente colocando-se os dois operandos próximos um do outro, como na álgebra.

A Short Code não era traduzida para código de máquina; ao contrário, era implementada com um interpretador puro. Naquela época, esse processo era chamado de programação automática. Evidentemente, ele simplificava o processo de programação, mas à custa do tempo de execução. A interpretação da Short Code era, aproximadamente, 50 vezes mais lenta do que o código de máquina.

### 2.2.2 Speedcoding

Em outros lugares, sistemas interpretativos foram desenvolvidos e estendiam as linguagens de máquina para incluir operações com números reais. O sistema Speedcoding desenvolvido por John Backus para o IBM 701, é um exemplo desse sistema (Backus, 1954). O interpretador Speedcoding convertia efetivamente o 701 para uma calculadora virtual de números reais com três endereços. O sistema incluía pseudo-instruções para as quatro operações aritméticas em números reais, bem como para operações como raiz quadrada, seno, arco-tangente, exponenciação e logaritmo. Ramificações condicionais e incondicionais e conversões de entrada/saída também faziam parte da arquitetura virtual. Para obter uma idéia das limitações desses sistemas, considere que a memória útil restante depois de carregar o interpretador era de somente 700 palavras e que a instrução add demorava 4,2 milissegundos para executar. Por outro lado, o Speedcoding incluía a nova facilidade de incrementar automaticamente registradores de endereço que não apareceu em hardware até os computadores UNIVAC 1107 em 1962. Por causa desses recursos, a multiplicação de matrizes podia ser feita em 12 instruções de Speedcoding. Backus afirmava que problemas que demoraram duas semanas para rodar em código de máquina podiam ser programados em algumas horas, usando-se o Speedcoding.

### 2.2.3 O Sistema “Compilador” UNIVAC

Entre 1951 e 1953, uma equipe da UNIVAC chefiada por Grace Hopper desenvolveu uma série de sistemas “compiladores” chamados A-0, A-1 e A-2, que expandiam um pseudocódigo para código de máquina da mesma maneira que as macros são expandidas para linguagem de montagem. A fonte do pseudocódigo para esses “compiladores” ainda era bastante primitiva; não obstante, até mesmo isso melhorou bastante o código de máquina, porque tornava os programas-fonte muito mais curtos. Wilkes (1952), independentemente, sugeriu um processo similar.

\*N. de T. Enchimento: *padding*. Caracteres usados para preencher as partes não-utilizadas de uma estrutura de dados, tais como um campo ou uma mensagem de comunicações. Um campo pode ser preenchido com espaços brancos, zeros ou nulos.

### 2.2.4 Trabalho Relacionado

Outros meios de facilitar a tarefa de programar eram desenvolvidos mais ou menos na mesma época. Na Universidade de Cambridge, David J. Wheeler chegou a um método para usar blocos de endereços realocáveis para resolver parcialmente o problema do endereçamento absoluto (Wheeler, 1950). Posteriormente, Maurice V. Wilkes (também em Cambridge) ampliou a idéia para projetar um programa que poderia combinar sub-rotinas escolhidas e alocar áreas de armazenagem (Wilkes et al., 1951, 1957). Isso, de fato, foi um avanço importante e fundamental.

Também devemos mencionar que as linguagens de montagem, bem diferentes dos pseudocódigos mencionados, desenvolveram-se durante o início da década de 50. Porém, elas tiveram pouco impacto sobre o projeto de linguagens de alto nível.

## 2.3 O IBM 704 e o FORTRAN

Certamente um dos maiores avanços particulares na computação veio com a introdução do IBM 704 em 1954, em grande medida porque suas capacidades motivaram o desenvolvimento do FORTRAN. Pode-se argumentar que, se não tivesse sido a IBM com o 704 e o FORTRAN, logo depois teria surgido alguma outra organização com um computador similar e com uma linguagem de alto nível relacionada. Porém, a IBM foi a primeira a ter tanto a previsão como os recursos para incumbir-se desses desenvolvimentos.

### 2.3.1 Embasamento Histórico

Uma das principais razões pelas quais se tolerou os sistemas interpretativos desde o final da década de 40 até meados da década de 50 foi a falta de hardware para números reais nos computadores disponíveis. Todas as operações com números reais tinham de ser simuladas em software, o que consumia muito tempo. Uma vez que grande parte do tempo de processador era gasto no processamento de números reais em software, o *overhead*<sup>1</sup> de interpretação e a simulação de indexação eram relativamente insignificantes. Na medida em que a manipulação tinha de ser feita pelo software, a interpretação era uma despesa aceitável. Porém, muitos programadores daquela época jamais usavam sistemas interpretativos, preferindo a eficiência da linguagem de máquina codificada à mão. O anúncio do sistema IBM 704, possuindo tanto instruções de indexação como de números reais em hardware, marcou o fim da era interpretativa, pelo menos para a computação científica.

Ainda que muitas vezes o FORTRAN seja creditado como sendo a primeira linguagem de alto nível compilada, a questão de quem merece o crédito pela iniciativa de implementação é bastante aberta. Knuth e Pardo (1977) dão o crédito a Alick E. Glennie por seu compilador Autocode para o computador Manchester Mark I. Glennie desenvolveu o compilador em Fort Halstead, Royal Armaments Research Establishment, na Inglaterra. O compilador entrou em operação em setembro de 1952. Porém, de acordo com John Backus (Wexelblat,

<sup>1</sup>N. de T. Overhead: a quantidade de tempo de processamento utilizada pelo software de sistema, tal como o sistema operacional, monitor TP ou gerenciador de banco de dados.

1981, p. 26), o Autocode de Glennie era de tão baixo-nível e orientado à máquina que não deve ser considerado um sistema compilado. Backus dá o crédito a Laning e Zierler, do *Massachusetts Institute of Technology*.

O projeto de Laning e Zierler (Laning e Zierler, 1954) foi o primeiro sistema de conversão algébrica a ser implementado. Por algébrico, queremos dizer que ele convertia expressões aritméticas, usava chamadas de funções para funções matemáticas e incluía referências de variáveis subscritas. O sistema foi implementado no computador Whirlwind do MIT, na forma de protótipo experimental no verão de 1952, em uma forma mais utilizável em maio de 1953. O tradutor gerava uma chamada a sub-rotina para codificar cada fórmula ou expressão no programa. A leitura da linguagem-fonte era fácil, e as únicas instruções de máquina reais incluídas seriam para ramificação. Ainda que esse trabalho tenha precedido o trabalho no FORTRAN, ele nunca saiu do MIT.

Apesar desses trabalhos anteriores, a primeira linguagem de alto nível compilada bem aceita foi o FORTRAN. As subseções seguintes apresentam um relato desse importante desenvolvimento.

### 2.3.2 Processo de Projeto

Mesmo antes do sistema 704 ter sido anunciado em maio de 1954, os planos para o FORTRAN já haviam sido iniciados. Em novembro de 1954, John Backus e seu grupo na IBM haviam produzido o relatório intitulado "The IBM Mathematical FORMula TRANslating System: FORTRAN" (IBM, 1954). Esse documento descrevia a primeira versão do FORTRAN, à qual nos referimos como FORTRAN 0, antes de sua implementação. Ele também declarava destemidamente que o FORTRAN ofereceria a eficiência dos programas codificados à mão e a facilidade de programação dos sistemas de pseudocódigo interpretativos. Em outro rompante de otimismo, o documento declarava que o FORTRAN eliminaria os erros de codificação e o processo de depuração. Baseado nessa premissa, seu primeiro compilador incluía pouca verificação de erros de sintaxe.

O ambiente em que o FORTRAN foi desenvolvido era o seguinte: (1) os computadores ainda eram pequenos, lentos e relativamente pouco confiáveis; (2) seu principal uso era para computações científicas; (3) não existia nenhuma maneira eficiente de programar computadores; (4) por causa de seu elevado custo em comparação com o custo dos programadores, a velocidade do código-objeto gerado era a meta principal dos primeiros compiladores FORTRAN. As características das suas primeiras versões do FORTRAN decorrem diretamente desse ambiente.

### 2.3.3 Visão Geral do FORTRAN I

O FORTRAN 0 foi modificado durante o período de implementação, que se iniciou em janeiro de 1955 e prosseguiu até o lançamento do compilador em abril de 1957. A linguagem implementada, chamada FORTRAN I, é descrita no primeiro FORTRAN Programmer's Reference Manual, publicado em outubro de 1956 (IBM, 1956). O FORTRAN I incluía formatação de entrada/saída, nomes de variáveis de até seis caracteres (havia apenas dois no FORTRAN 0), sub-rotinas definidas pelo usuário, ainda que elas não pudessem ser compiladas separadamente, a instrução de seleção IF e a instrução DO loop.

O FORTRAN 0 incluía uma instrução lógica IF cuja expressão booleana usava operadores relacionais em suas formas algébricas — por exemplo, > para maior do que. A tabela de caracteres do 704 não incluía caracteres como >; assim, esses operadores relacionais

tiveram de ser deixados de lado. Uma vez que a máquina tinha uma instrução de ramificação de três caminhos baseada na comparação do valor existente em uma localização de armazenamento com o valor de um registrador, o IF lógico original era substituído pela seleção aritmética, cuja forma era

IF (expressão aritmética) N1, N2, N3

em que N1, N2 e N3 são rótulos de instruções. Se o valor da expressão for negativo, a ramificação será para N1; se for zero, será para N2; se for maior que zero, para N3. Essa instrução ainda faz parte do FORTRAN.

A forma da instrução (laço) iterativa do FORTRAN I era

DO N1 variável = primeiro\_valor, último\_valor

em que N1 era o rótulo da última instrução do laço, e a instrução na linha seguinte ao DO era a primeira.

Como acontece com a instrução IF, o 704 tinha uma única instrução para implementar DO. O DO do FORTRAN foi pensado dessa maneira uma vez que a instrução fora projetada para laços pós-testados. Um laço pré-testado poderia ter sido implementado no 704, mas isso exigiria uma instrução de máquina adicional, e, desde que a eficiência era a preocupação principal no projeto do FORTRAN, isso não foi feito.

Todos os comandos de controle do FORTRAN basearam-se em instruções do 704. Não está claro se os projetistas do 704 determinaram o projeto da instrução de controle do FORTRAN I ou se os projetistas dele sugeriram essas instruções aos desenvolvedores do 704.

Não havia instruções com tipificação de dados no FORTRAN I. Variáveis cujos nomes iniciavam-se com I, J, K, L, M e N eram implicitamente do tipo inteiro, e todas as outras eram implicitamente do tipo real. A escolha das letras para essa convenção baseou-se no fato de que, naquela época, números inteiros eram usados principalmente como subscritos, e os cientistas normalmente usavam o i, o j e o k para subscritos. Para serem generosos, os projetistas do FORTRAN lançaram as três letras adicionais.

A mais audaciosa reivindicação feita pelo grupo de desenvolvimento do FORTRAN durante o seu projeto foi que o código de máquina produzido pelo compilador seria quase tão eficiente quanto aquilo que poderia ser produzido manualmente. Isso, mais do que qualquer outra coisa, tornou céticos os usuários potenciais e impediu que houvesse um grande interesse no FORTRAN antes de seu lançamento real. Para surpresa de quase todo mundo, entretanto, o grupo de desenvolvimento do FORTRAN quase atingiu sua meta em termos de eficiência. A maior parte dos 18 trabalhadores-ano de esforço usados para construir o primeiro compilador foram gastos em otimização, e os resultados foram notavelmente eficazes.

O sucesso inicial do FORTRAN é mostrado pelos resultados de uma pesquisa feita em abril de 1958. Naquela época, quase metade do código escrito para os 704 ainda era feita em FORTRAN, apesar do extremo ceticismo da maior parte do mundo da programação a apenas um ano antes.

#### 2.3.4 Visão Geral do FORTRAN II

O compilador FORTRAN II foi distribuído na primavera de 1958. Ele resolveu muitos dos problemas existentes no sistema de compilação FORTRAN I e acrescentou alguns recursos importantes à linguagem, sendo o mais destacado a compilação independente de sub-rotinas. Sem a compilação independente, qualquer mudança em um programa exige que o

programa inteiro seja recompilado. A falta de capacidade de compilação independente do FORTRAN I, combinada com a má confiabilidade do 704, impuseram uma virtual restrição ao tamanho dos programas para aproximadamente 300 a 400 linhas (Wexelblat, 1981, p. 68). Programas mais longos tinham pouca chance de serem compilados completamente antes que ocorresse uma falha de máquina. A capacidade de incluir versões em linguagem de máquina, de subprogramas previamente compilados, abreviava consideravelmente o processo de compilação.

### 2.3.5 FORTRAN IV, FORTRAN 77 e FORTRAN 90

Um FORTRAN III foi desenvolvido, mas jamais distribuído popularmente. O FORTRAN IV, porém, tornou-se uma das linguagens de programação mais usadas em seu tempo. Ela evoluiu ao longo do período de 1960 a 1962 e foi a versão padrão até 1978, quando o relatório FORTRAN 77 (ANSI, 1978a) foi lançado. O FORTRAN IV obteve um ganho de qualidade em relação ao FORTRAN II sob muitos aspectos. Entre suas muito importantes adições havia as declarações de tipo explícitas para variáveis, uma construção lógica IF e a capacidade de passar subprogramas como parâmetros a outros subprogramas.

O FORTRAN 77 mantém a maioria dos recursos do FORTRAN IV e adiciona manipulação de cadeias de caracteres, instruções lógicas de controle de laço e um IF com a cláusula opcional ELSE.

O FORTRAN 90 é o nome da versão mais recente do FORTRAN (ANSI, 1992). O FORTRAN 90 é drasticamente diferente do FORTRAN 77. As mudanças mais significativas são descritas nos parágrafos seguintes.

Um conjunto de funções é incorporado para operações com matrizes. Elas incluem DOTPRODUCT, MATMUL, TRANSPOSE, MAXVAL, MINVAL, PRODUCT e SUM, cujos significados são evidentes a partir de seus nomes. Eis apenas algumas das funções mais úteis disponíveis.

As matrizes podem ser dinamicamente alocadas e desalocadas sob comando se tiverem sido declaradas para serem ALLOCATABLE. Essa é uma mudança radical em relação aos FORTRANs anteriores, os quais tinham somente dados estáticos. Uma forma de registros, chamados tipos derivados, é incluída. Ponteiros também fazem parte do FORTRAN 90.

Novas instruções de controle foram adicionadas: CASE é uma instrução de seleção múltipla, EXIT é usada para sair prematuramente de um laço, e CYCLE é usada para transferir o controle para a base de um laço, mas não para fora.

Subprogramas podem ser recursivos e também ter parâmetros opcionais e de palavra-chave.

Adicionou-se uma facilidade modular que é semelhante aos pacotes da Ada. Módulos podem conter declarações de dados e subprogramas, cada um dos quais pode ser PRIVATE ou PUBLIC para regular o acesso externo.

Um novo conceito que foi incluído na definição do FORTRAN 90 é o de remover recursos de linguagem de versões anteriores. Embora o FORTRAN 90 inclua todos os recursos do FORTRAN 77, ele tem duas listas de recursos que poderão ser eliminados nas versões futuras do FORTRAN. A lista de recursos obsoletos inclui os que podem ser eliminados na próxima versão do FORTRAN depois da 90. Na lista, há coisas como as instruções IF aritméticas e GOTO atribuídas. A lista de recursos desaprovados tem itens que podem ser eliminados na segunda versão subsequente depois da 90. Nessa lista, estão incluídas as instruções COMMON, EQUIVALENCE e GOTO computadas, bem como as instruções de função.

### 2.3.6 Avaliação

A equipe do FORTRAN original imaginava o projeto da linguagem somente como um prelúdio necessário para a tarefa crítica de projetar o tradutor. Além disso, nunca lhes ocorreu que o FORTRAN seria usado em computadores não-fabricados pela IBM. De fato, eles foram obrigados a construir compiladores FORTRAN para outras máquinas IBM somente porque o sucessor do 704, o 709, foi anunciado antes do compilador 704 FORTRAN ser lançado. O efeito que o FORTRAN teve no uso dos computadores, juntamente com o fato das linguagens de programação subsequentes terem uma dívida com o FORTRAN, são, realmente, impressionantes à luz das modestas metas de seus projetistas.

Um dos recursos do FORTRAN I e de todos os seus sucessores, exceto a versão 90, que permite compiladores altamente otimizados, é que os tipos e o armazenamento de todas as variáveis são fixados antes da execução. Nenhuma nova variável ou espaço pode ser alocado em tempo de execução. Isso é um sacrifício da flexibilidade em favor da simplicidade e da eficiência. Ele elimina a possibilidade de subprogramas recursivos e dificulta a implementação de estruturas de dados que crescem ou mudam de forma dinamicamente. Obviamente, os tipos de programas que eram construídos na época do desenvolvimento das primeiras versões do FORTRAN eram principalmente numéricos por natureza, além de simples em comparação com os projetos de software recentes. Portanto, o sacrifício não foi grande.

O sucesso global do FORTRAN é difícil de exagerar: ele mudou drasticamente e para sempre o uso dos computadores. Isso se deve, é claro, em grande parte, ao fato de ser a primeira linguagem de alto nível muito usada. Em comparação com os conceitos e com as linguagens desenvolvidas mais tarde, as primeiras versões do FORTRAN padecem em várias maneiras, como se poderia esperar. Afinal de contas, os automóveis Ford Modelo T de 1910 não podem ser comparados sob todos os aspectos com os Ford Mustangs de 2001. Ainda assim, apesar das insuficiências do FORTRAN, o impulso do enorme investimento em software, entre outros fatores, o manteve entre as mais usadas de todas as linguagens de alto nível.

Alan Perlis, um dos projetistas do ALGOL 60, disse a respeito do FORTRAN em 1978, "o FORTRAN é a língua franca<sup>1</sup> do mundo da computação. Ela é a linguagem das ruas no melhor sentido da palavra, não no sentido prostituído da palavra. E ela sobreviveu e sobreviverá porque tornou-se uma parte notavelmente útil de um comércio muito vital" (Wexelblat, 1981, p. 161).

A seguir, apresentamos um exemplo de programa FORTRAN 90:

```
C EXEMPLO DE PROGRAMA FORTRAN 90
C ENTRADA: UM NÚMERO INTEIRO, COMP_LIS, EM QUE COMP_LIS
C           É MENOR QUE 100, SEGUIDO DE COMP_LIS VALORES
C SAÍDA:   O NÚMERO DE VALORES DE ENTRADA QUE SÃO MAiores DO QUE
C           A MÉDIA DE TODOS OS VALORES DE ENTRADA
          INTEGER INTLIST(99)
          INTEGER COMP_LIS, CONTADOR, SOMA, MEDIA, RESULTADO
          RESULTADO = 0
          SOMA = 0
          READ *, COMP_LIS
```

<sup>1</sup>N. de T. Língua Franca: [(latim) literalmente, língua dos franceses] a língua comercial do Levante — uma mistura da língua dos povos da região e de comerciantes estrangeiros. Qualquer linguagem híbrida ou outra linguagem usada em uma ampla região como a língua comercial comum entre pessoas de diferentes idiomas.

```

        IF ((COMP_LIS .GT. 0) .AND.
            (COMP_LIS .LT. 100)) THEN
C  LEIA DADOS DE ENTRADA EM UM VETOR E COMPUTE SUA SOMA
    DO 10 CONTADOR = 1, COMP_LIS
        READ *, INTLIST(CONTADOR)
        SOMA = SOMA + INTLIST(CONTADOR)
10     CONTINUE
C  COMPUTE A MÉDIA
        MEDIA = SOMA / COMP_LIS
C  CONTE OS VALORES QUE SÃO MAIORES DO QUE A MÉDIA
    DO 20 CONTADOR = 1, COMP_LIS
        IF (INTLIST(CONTADOR) .GT. MEDIA) THEN
            RESULTADO = RESULTADO + 1
        END IF
20     CONTINUE
C  IMPRIMA O RESULTADO
        PRINT *, 'NUMERO DE VALORES > QUE A MEDIA:', RESULTADO
    ELSE
        PRINT *, 'ERRO: VALOR DO COMPRIMENTO DA LISTA NÃO É
                    VÁLIDO'
    END IF
    STOP
END

```

## 2.4 Programação Funcional: LISP

A primeira linguagem de programação funcional foi inventada para oferecer recursos de linguagem para processamento de listas, cuja necessidade surgiu a partir das primeiras aplicações na área da inteligência artificial (IA).

### 2.4.1 O Início da Inteligência Artificial e o Processamento de Listas

O interesse na IA começou a surgir em meados da década de 50 em uma série de lugares. Parte desse interesse surgiu da Lingüística, parte da Psicologia e parte da Matemática. Os lingüistas estavam preocupados com o processamento da linguagem natural. Os psicólogos, por sua vez, preocuparam-se em modelar a armazenagem e a recuperação de informações humanas, juntamente com outros processos fundamentais do cérebro. Os matemáticos queriam mecanizar certos processos inteligentes, como, por exemplo, a demonstração de teoremas. Todas essas investigações chegaram à mesma conclusão: algum método precisava ser desenvolvido para permitir que os computadores processassem dados simbólicos em listas encadeadas. Na época, quase toda a computação fundamentava-se em dados numéricos armazenados em matrizes.

O conceito de processamento de listas foi desenvolvido por Allen Newell, J.C. Shaw e Herbert Simon, publicado pela primeira vez em um documento clássico que descreve um dos primeiros programas de IA, o Logical Theorist, e uma linguagem na qual ele podia ser implementado (Newell e Simon, 1956). A linguagem, chamada IPL-I (*Information Processing*

*Language I*), jamais foi implementada. A versão seguinte, a IPL-II, foi implementada em um computador Rand Corporation Johnniac. O desenvolvimento da IPL prosseguiu até 1960, quando se publicou a descrição da IPL-V (Newell e Tonge, 1960). O baixo nível das IPL impidiu seu uso generalizado. Elas eram, de fato, linguagens de montagem para um computador hipotético, implementadas por meio de interpretadores, nos quais eram incluídas instruções para processamento de listas. A primeira implementação ocorreu na obscura máquina Johnniac, outro fator que impidiu a popularização das linguagens IPL.

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

A IBM interessou-se pela IA em meados da década de 50 e escolheu a demonstração de teoremas como uma área de demonstração. Na época, o projeto FORTRAN ainda estava a caminho. O elevado custo do compilador FORTRAN I convenceu a IBM de que seu processamento de listas devia estar anexado ao FORTRAN, em vez de permanecer sob a forma de uma nova linguagem. Assim, o *FORTRAN List Processing Language* (FLPL) foi projetado e implementado como uma extensão ao FORTRAN. O FLPL acabou sendo usado para construir um demonstrador de teoremas para geometria plana, a qual era então considerada a área mais fácil para demonstração mecânica de teoremas.

#### 2.4.2 Projeto do LISP

John McCarthy, do MIT, assumiu um cargo de verão no *Information Research Department* da IBM em 1958. Sua meta para a estação era investigar as computações simbólicas e desenvolver um conjunto de requisitos para fazê-las. Como uma área de problema experimental, ele escolheu a diferenciação de expressões algébricas. A partir desse estudo surgiu uma lista de requisitos de linguagem. Entre eles, estavam os métodos de fluxo de controle de funções matemáticas: recursão e expressões condicionais. A única linguagem de alto nível na época, o FORTRAN, não tinha nenhuma delas.

Outro requisito que surgiu da investigação sobre a diferenciação simbólica foi a necessidade de listas alocadas dinamicamente e de algum tipo de desalocação implícita de listas abandonadas. McCarthy simplesmente não permitiria que seu elegante algoritmo para diferenciação fosse atravancado com instruções de desalocação explícita.

Tornou-se claro para McCarthy que uma nova linguagem era necessária uma vez que o FLPL não suportava recursão, expressões condicionais, alocação de armazenagem dinâmica ou desalocação implícita.

Quando McCarthy retornou ao MIT no outono de 1958, ele e Marvin Minsky formaram o MIT AI Project, com financiamento do *Research Laboratory for Electronics*. O primeiro esforço importante do projeto foi produzir um sistema para processamento de listas. Ele se destinava a ser usado inicialmente para implementar um programa proposto por McCarthy, chamado *Advice Taker*. Tal aplicação impulsionou o desenvolvimento da linguagem de processamento de listas, o LISP. A primeira versão desta, às vezes, é chamada de LISP pura porque trata-se de uma linguagem funcional. Na seção seguinte, descrevemos o desenvolvimento da LISP pura.

#### 2.4.3 Visão Geral da Linguagem

##### 2.4.3.1 Estruturas de Dados

A LISP pura tem somente dois tipos de estruturas de dados: átomos e listas. Átomos são ou símbolos, sob a forma de identificadores, ou literais numéricas. O conceito de armazenar

informações simbólicas em listas encadeadas é natural e foi usado na IPL-II. Essas estruturas permitem inserções e exclusões em qualquer ponto, operações que, então, se imaginava serem partes necessárias do processamento de listas. Na forma em que se desenvolveu, entretanto, o LISP raramente exige tais operações.

As listas são especificadas, delimitando-se seus elementos com parênteses. Listas simples, cujos elementos restringem-se a átomos, têm a forma do exemplo

$(A \ B \ C \ D)$

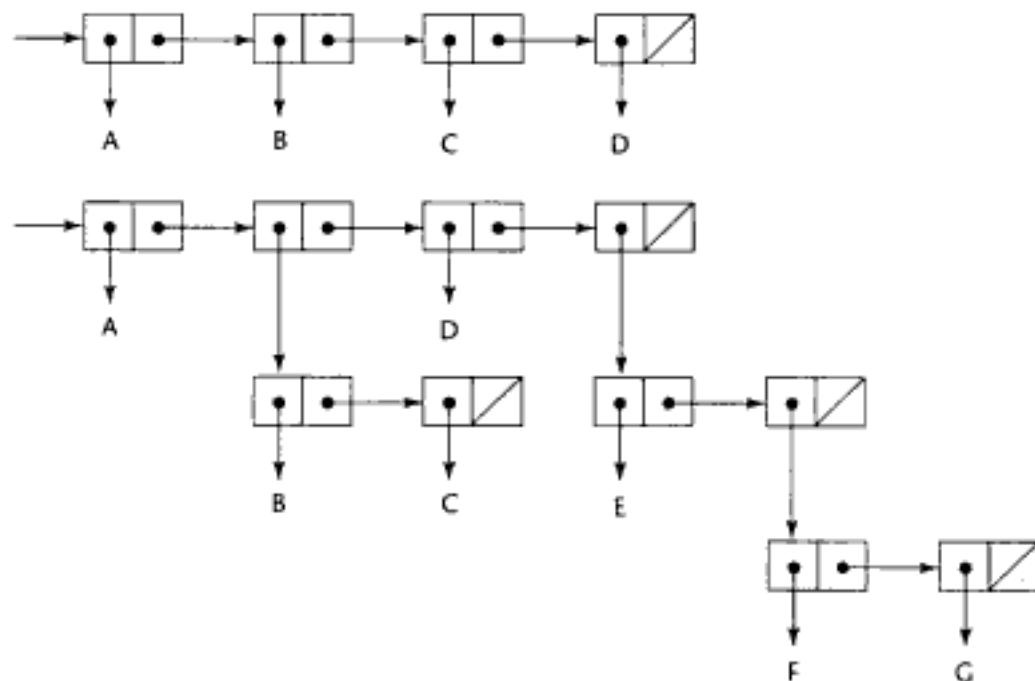
Estruturas de listas aninhadas também são especificadas por parênteses. Por exemplo, a lista

$(A \ (B \ C) \ D \ (E \ (F \ G) \ ) \ )$

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  $(E \ (F \ G))$ , cujo segundo elemento é a sublista  $(F \ G)$ .

Internamente, as listas usualmente são armazenadas como estruturas de lista com vínculo simples, em que cada vértice tem dois ponteiros e representa um elemento da lista. Um vértice para um átomo tem seu primeiro ponteiro apontando para alguma representação do átomo, como, por exemplo, seu símbolo ou seu valor numérico. Um vértice para um elemento da sublista tem seu primeiro ponteiro apontando para o primeiro vértice da sublista. Em ambos os casos, o segundo ponteiro de um vértice aponta para o elemento seguinte da lista que é referenciada por meio de um ponteiro ao seu primeiro elemento.

As representações internas das duas listas mostradas anteriormente são descritas na Figura 2.2. Note que os elementos de uma lista são mostrados horizontalmente. Seu último elemento não tem sucessor, de modo que seu vínculo é NIL. As sublistas são mostradas com a mesma estrutura.



**FIGURA 2.2** Representação interna de duas listas LISP.

#### 2.4.3.2 Processos na Programação Funcional

O LISP foi projetado como uma linguagem de programação funcional. Toda computação em um programa funcional é realizada por meio da aplicação de funções a argumentos. Nem as instruções de atribuição, nem as variáveis que abundam nos programas em linguagem imperativa são necessárias nos programas em linguagem funcional. Além disso, processos iterativos podem ser especificados com chamadas a função recursivas, tornando os laços desnecessários. Esses conceitos básicos da programação funcional tornam-na significativamente diferente da programação em uma linguagem imperativa.

#### 2.4.3.3 A Sintaxe do LISP

O LISP é muito diferente das linguagens imperativas porque é uma linguagem de programação funcional e porque os programas em LISP parecem bem diferentes daqueles em Java ou em C. Por exemplo, a sintaxe do C é uma complicada mistura de inglês e álgebra, enquanto que a do LISP é um modelo de simplicidade. O código de programa e os dados têm exatamente a mesma forma: listas entre parênteses. Consideremos novamente a lista

(A B C D)

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

#### 2.4.4 Avaliação

O LISP dominou as aplicações de IA durante um quarto de século e ainda é a linguagem mais usada nesse ramo. Grande parte dos motivos que fizeram com que o LISP fosse considerado muito ineficiente foram eliminados. Muitas implementações contemporâneas são compiladas, e o código resultante muito mais rápido que a execução do código fonte em um interpretador. Além de seu sucesso na IA, o LISP foi o pioneiro na programação funcional, que se mostrou uma vívida área de pesquisa em linguagens de programação. Como afirmamos no Capítulo 1, muitos pesquisadores de linguagens de programação acreditam que a programação funcional é uma abordagem muito melhor ao desenvolvimento de software do que o uso de linguagens imperativas.

Durante a década de 70 e início da década de 80, um grande número de diferentes dialetos do LISP foi desenvolvido e usado, o que levou ao familiar problema da portabilidade. Para corrigir a situação, uma versão padrão chamada COMMON LISP foi desenvolvida (Steele, 1984).

A Scheme, um dialeto do LISP, e a programação funcional serão discutidas detalhadamente no Capítulo 15. A seguir, apresentamos um exemplo de programa em LISP:

```
; Função de exemplo LISP
; O código seguinte define uma função predicada LISP
; que pega duas listas como argumentos e retorna True
; se as duas listas forem iguais, e NIL (falso)
; caso contrário.
(DEFUN listas_iguais (lis1 lis2))
(COND
  ( (ATOM lis1) (EQ lis1 lis2))
  ( (ATOM lis2) NIL)
```

```
    ( (listas_iguais (CAR lis1) (CAR lis2) )
      (listas_iguais (CDR lis1) (CDR lis2) ) )
    (T NIL)
  )
)
```

## 2.4.5 Dois Descendentes do LISP

Dois dialetos do LISP agora são bastante usados, a COMMON LISP e a Scheme. Ambos serão brevemente discutidos nas subseções seguintes.

### 2.4.5.1 Scheme

A linguagem Scheme surgiu no MIT em meados da década de 70 (Sussman e Steele, 1975). Ela é caracterizada por seu tamanho pequeno, pelo uso exclusivo de escopo estático (discutido no Capítulo 6) e pelo tratamento que dá às funções como entidades de primeira classe. Como entidades de primeira classe, as funções da Scheme podem ser os valores das expressões e dos elementos de listas; elas podem ser atribuídas a variáveis, passadas como parâmetros e retornadas como os valores de aplicações da função. As versões anteriores do LISP não ofereciam todas essas capacidades, nem faziam uso do escopo estático.

Como uma linguagem pequena com sintaxe e semântica simples, a Scheme adapta-se bem a aplicações educacionais, como, por exemplo, cursos de programação funcional e introduções gerais à programação. Conforme foi mencionado antes, a Scheme será mais bem descrita no Capítulo 15.

### 2.4.5.2 COMMON LISP

A COMMON LISP (Steele, 1984) é fruto de um esforço para combinar os recursos de diversos dialetos do LISP desenvolvidos no início da década de 80, inclusive a Scheme, em uma única linguagem. Sendo essa amálgama, a COMMON LISP é uma linguagem grande e complexa. Sua base, entretanto, é o LISP puro; assim, sua sintaxe, suas funções primitivas e sua natureza fundamental vêm dessa linguagem.

Reconhecendo a flexibilidade oferecida pelo escopo dinâmico, bem como a simplicidade do escopo estático, a COMMON LISP permite ambos. O escopo padrão para as variáveis é o estático, mas quando se declara uma variável como **special**, ela terá seu escopo definido dinamicamente.

A COMMON LISP tem um grande número de tipos de dados e de estruturas, inclusive registros, matrizes, números complexos e cadeias de caracteres. Ela também tem uma forma de pacotes para modularizar conjuntos de funções e de dados que oferecem controle de acesso.

A COMMON LISP será descrita detalhadamente no Capítulo 15.

## 2.4.6 Linguagens Relacionadas

A ML (MetaLanguage) foi projetada originalmente na década de 80 por Robin Milner na Universidade de Edinburgo como uma metalinguagem para um sistema de verificação de programas chamado *Logic for Computable Functions* (LCF) (Milner et al., 1990). A ML é, primeiramente, uma linguagem funcional, mas ela também suporta programação imperati-

va. Na ML, as funções são mais gerais do que nas linguagens imperativas: elas são rotineiramente passadas como parâmetros e podem ser polimórficas, significando que podem pegar parâmetros de vários tipos em diferentes chamadas. De maneira diversa do LISP e da Scheme, o tipo de toda variável e de expressão na ML pode ser determinado no momento da compilação. Os tipos são associados a objetos em vez de a nomes. Os tipos e as expressões são inferidos a partir do contexto da expressão, conforme discutiremos no Capítulo 7.

Diferentemente do LISP e da Scheme, a ML não usa a sintaxe funcional com parênteses que se originou com as expressões lambda. Em vez disso, a sintaxe da ML assemelha-se a das linguagens imperativas, como Java e C++.

A Miranda foi desenvolvida por David Turner na Universidade de Kent em Canterbury, Inglaterra, no início da década de 80 (Turner, 1986). Essa linguagem baseia-se parcialmente nas linguagens ML, SALS e KRC. A Haskell (Hudak e Fasel, 1992) baseia-se, em grande parte, na Miranda. Como a Miranda, ela é uma linguagem puramente funcional, não tendo qualquer variável e nenhuma instrução de atribuição. Outra característica marcante da Haskell é o uso de uma avaliação preguiçosa. Nenhuma expressão é avaliada até que seu valor seja exigido. Isso leva a algumas capacidades surpreendentes.

Tanto a ML como a Haskell serão discutidas brevemente no Capítulo 15.

## 2.5 O Primeiro Passo Rumo à Sofisticação: ALGOL 60

O ALGOL 60 influenciou fortemente as linguagens de programação subsequentes e, portanto, é de fundamental importância em qualquer revisão histórica das linguagens.

### 2.5.1 Embasamento Histórico

O ALGOL 60 passou a existir como resultado dos esforços para projetar uma linguagem universal. No final de 1954, o sistema algébrico de Laning e Zierler estava em operação há mais de um ano, e o primeiro relatório sobre o FORTRAN havia sido publicado. O FORTRAN tornou-se uma realidade em 1957, quando diversas outras linguagens de alto nível estavam sendo desenvolvidas. As mais notáveis entre elas eram a IT, projetada por Alan Perlis na Carnegie Tech, e duas para computadores UNIVAC, a MATH-MATIC e a UNICODE. A proliferação de linguagens tornou difícil a comunicação entre os usuários. Além disso, as novas linguagens desenvolviam-se em torno de arquiteturas únicas, algumas para computadores UNIVAC e outras para máquinas da série IBM 700. Em resposta a essa proliferação, diversos grupos de grandes usuários de computador nos Estados Unidos, incluindo o SHARE (o grupo de usuários científicos da IBM) e o USE (UNIVAC Scientific Exchange, o grupo de usuários científicos UNIVAC em grande escala), submeteram uma petição à Association for Computing Machinery (ACM) em 10 de maio de 1957, para formarem um comitê de estudos e para recomendar ações voltadas à criação de uma linguagem de programação universal. Ainda que o FORTRAN pudesse ter sido um candidato, ele não poderia se tornar universal, porque, na época, era propriedade somente da IBM.

Anteriormente, em 1955, a GAMM (sigla alemã de Sociedade para Matemática Aplicada e Mecânica) também formou um comitê para projetar uma linguagem algorítmica universal e independente de máquina para ser usada em todos os tipos de computadores. Seu desejo deveu-se, em grande parte, ao medo que os europeus tinham de serem dominados pela IBM. No final de 1957, entretanto, o surgimento de diversas linguagens de alto

nível nos Estados Unidos convenceu o subcomitê da GAMM de que seu esforço tinha de ser ampliado para incluir os americanos, e uma carta-convite foi enviada à ACM. Em abril de 1958, depois que Fritz Bauer, da GAMM, apresentou a proposta formal à ACM, os dois grupos concordaram oficialmente em participar de um projeto conjunto de elaboração de uma linguagem.

### 2.5.2 Primeiro Processo de Projeto

A GAMM e a ACM concluíram que o esforço de projeto conjunto devia ser feito em um encontro em que cada grupo enviaría quatro membros. O encontro, realizado em Zurique de 27 de maio a 1º de junho de 1958, iniciou-se com as seguintes metas para a nova linguagem:

- A sintaxe da linguagem deveria estar o mais próximo possível da notação matemática padrão, e os programas nela escritos deveriam ser legíveis e com pouca explicação adicional.
- Deveria ser possível usar a linguagem para a descrição de processos computacionais em publicações.
- Os programas na nova linguagem deveriam ser mecanicamente traduzíveis para linguagem de máquina.

A primeira meta indica que a nova linguagem seria usada para programação científica, que era a principal área de aplicação de computadores naquela época. A segunda era algo inteiramente novo nos negócios da computação. A última meta é uma necessidade óbvia para qualquer linguagem de programação.

Dependendo de como é visto, o encontro de Zurique produziu resultados momentâneos ou argumentos infundáveis. De fato, ele fez ambos. O encontro em si envolveu inúmeras concessões, tanto entre pessoas como entre os dois lados do Atlântico. Em alguns casos, as concessões não foram tanto sobre grandes questões, mas sim sobre esferas de influência. A questão de usar-se uma vírgula (o método europeu) ou um ponto (o método americano) para separar as casas decimais é um exemplo.

### 2.5.3 Visão Geral do ALGOL 58

A linguagem projetada no encontro de Zurique foi chamada de *International Algorithmic Language* (IAL). Sugeriu-se durante o projeto que seu nome deveria ser ALGOL, de ALGOrithmic Language, mas houve rejeição porque não refletia o escopo internacional do comitê. Durante o ano seguinte, entretanto, o nome foi mudado para ALGOL, e a linguagem subsequente tornou-se conhecida como ALGOL 58.

De muitas maneiras, o ALGOL 58 descendeu do FORTRAN, o que é muito natural. Ele generalizou muitos dos recursos deste e adicionou diversas construções e conceitos novos. Algumas das generalizações tinham a ver com a meta de não ligar a linguagem a qualquer máquina particular, e outras tentativas de tornar a linguagem mais flexível e poderosa. Uma combinação rara de simplicidade e de elegância surgiu desse esforço.

O ALGOL 58 formalizou o conceito de tipo de dados, não obstante somente variáveis que não eram números reais exigissem declaração explícita. Essa linguagem agregou a idéia de instruções compostas, que a maioria das linguagens subsequentes incorporou. Alguns recursos do FORTRAN que foram generalizados: permitiu-se que os identificadores tives-

sem qualquer tamanho, ao contrário da restrição do FORTRAN de seis ou menos caracteres; qualquer número de dimensões de matriz, diferentemente da limitação daquele a não mais de três; o limite inferior de subscrito de matriz podia ser especificado pelo programador, ao passo que no FORTRAN ele era implicitamente 1; foram permitidas instruções de seleção aninhadas, o que não acontecia nesta linguagem.

O ALGOL 58 adotou o operador de atribuição de uma maneira bastante inusitada. Zuse usava a forma

expressão => variável

para a instrução de atribuição na Plankalkül. Embora esta última ainda não tivesse sido publicada, alguns dos membros europeus do comitê do ALGOL 58 estavam familiarizados com ela. O comitê interessou-se pela forma de atribuição da Plankalkül, mas, devido aos argumentos sobre as limitações de caracteres, o símbolo maior que (>) foi substituído por dois pontos (:). Depois, em grande parte pela insistência dos americanos, a instrução inteira voltou-se para a forma

variável := expressão

Os europeus preferiam a forma oposta.

#### 2.5.4 Recepção do Relatório ALGOL 58

A publicação do relatório ALGOL 58 (Perlis e Samelson, 1958) em dezembro de 1958 foi recebida com muito entusiasmo. Nos Estados Unidos, viu-se a nova linguagem mais como um conjunto de idéias para o projeto de linguagens de programação do que como um padrão universal. De fato, o relatório ALGOL 58 não pretendia ser um produto acabado, mas, ao contrário, um documento preliminar para ser discutido internacionalmente. Não obstante, três grandes esforços de projeto e de implementação usaram o relatório como base. Na Universidade de Michigan, nasceu a linguagem MAD (Arden et al., 1961). O U.S. Naval Electronics Group produziu a linguagem NELIAC (Huskey et al., 1963). Na System Development Corporation, projetou-se e implementou-se a JOVIAL (Shaw, 1963). JOVIAL, uma sigla de *Jules' Own Version of the International Algebraic Language*, representa a única linguagem baseada no ALGOL 58 a ser bastante usada (Jules era Jules I. Schwartz, um dos projetistas da JOVIAL). A JOVIAL popularizou-se porque foi a linguagem científica oficial da Força Aérea dos Estados Unidos durante um quarto de século.

O restante da comunidade de computação dos Estados Unidos não foi assim tão gentil com ela. A princípio, tanto a IBM como seu principal grupo de usuários científicos, o SHARE, pareceram abraçar o ALGOL 58. A IBM iniciou uma implementação pouco depois que o relatório foi publicado, e o SHARE formou um subcomitê, o SHARE IAL, para estudar a linguagem. O subcomitê subsequentemente recomendou que a ACM padronizasse o ALGOL 58 e que a IBM o implementasse para todos os computadores da série 700. O entusiasmo teve, entretanto, curta duração. Na primavera de 1959, tanto a IBM como o SHARE, em função de sua experiência com o FORTRAN, concluíram que já haviam enfrentado muitos gastos e problemas para iniciarem-se em uma nova linguagem, tanto em termos de usarem os compiladores de nova geração como em termos de treinarem os usuários na nova linguagem e convencê-los a usá-la. Em meados de 1959, tanto a IBM como o SHARE haviam investido tanto no FORTRAN que decidiram mantê-lo como a linguagem científica para as máquinas da série IBM 700, abandonando, portanto, o ALGOL 58.

### 2.5.5 Projeto do ALGOL 60

Durante 1959, o ALGOL 58 foi freneticamente debatido tanto na Europa como nos Estados Unidos. Publicou-se grande número de modificações sugeridas e de adições no *ALGOL Bulletin* europeu e na *Communication of the ACM*. Um dos mais importantes eventos de 1959 foi a apresentação do trabalho do comitê de Zurique na *International Conference on Information Processing*, porque foi lá que Backus apresentou sua nova notação para descrever a sintaxe das linguagens de programação, que ficou conhecida mais tarde como BNF (de Backus-Naur form — forma de Backaus-Naur). A BNF será descrita detalhadamente no Capítulo 3.

Em janeiro de 1960, realizou-se o segundo encontro sobre o ALGOL, em Paris. A tarefa consistia em debater as 80 sugestões formalmente submetidas para consideração. Peter Naur, da Dinamarca, envolvera-se no desenvolvimento do ALGOL, mesmo não tendo sido membro do grupo de Zurique. Foi Naur quem criou e publicava o *ALGOL Bulletin*. Ele gastou um tempo considerável estudando o documento de Backus que apresentou a BNF e concluiu que esta última deveria ser usada para descrever formalmente os resultados do encontro de 1960. Depois de fazer algumas mudanças relativamente pequenas na BNF, ele redigiu uma descrição da nova linguagem proposta em BNF e entregou-a aos membros do grupo de 1960 no início do encontro.

### 2.5.6 Visão Geral do ALGOL 60

Ainda que o encontro de 1960 tenha durado somente seis dias, as modificações feitas no ALGOL 58 foram drásticas. Entre os desenvolvimentos mais novos estavam os seguintes:

- O conceito de estrutura em bloco foi introduzido. Isso permitia que o programador localizasse partes de programas ao introduzir novos ambientes de dados ou escopos.
- Permitiram-se dois meios diferentes de passar parâmetros a subprogramas: por valor e por nome.
- Permitiu-se que os procedimentos fossem recursivos. A descrição do ALGOL 58 não esclarecia essa questão. Note que, não obstante a recursão ser nova nas linguagens imperativas, a LISP já oferecia funções recursivas em 1959.
- As matrizes dinâmicas na pilha (*stack-dynamic array*) foram permitidas. Uma matriz dinâmica na pilha é aquela para a qual as faixas de subscrito são especificadas por variáveis, de modo que o tamanho da matriz é definido no momento em que a memória é alocada, o que acontece quando sua declaração é alcançada durante a execução. As matrizes dinâmicas na pilha serão discutidas detalhadamente no Capítulo 6.

Diversos recursos que poderiam ter um impacto dramático sobre o sucesso ou fracasso da linguagem foram propostos, mas rejeitados. Os mais importantes entre eles foram as instruções de entrada e saída com formatação, que foram omitidas porque imaginava-se que eram demasiadamente dependentes da máquina.

\*N. de T. Stack: Pilha. Lista linear com a disciplina de acesso em que o último a entrar é o primeiro a sair.

O relatório ALGOL 60 foi publicado em maio de 1960 (Naur, 1960). Uma série de ambigüidades ainda permaneceu na descrição da linguagem, e um terceiro encontro ficou programado para abril de 1962 em Roma para abordar os problemas. Nesse encontro, o grupo tratou somente de problemas; não se permitiu adições. Os resultados foram publicados sob o título "Revised Report on the Algorithmic Language ALGOL 60" (Backus et al., 1962).

### 2.5.7 Avaliação do ALGOL 60

Sob alguns aspectos, o ALGOL 60 foi um grande sucesso; sob outros, um obscuro fracasso. Essa versão obteve sucesso ao tornar-se, quase imediatamente, o único meio formal aceitável para comunicar algoritmos na literatura e, ao longo de 20 anos, a única linguagem para publicar algoritmos. Toda linguagem de programação imperativa projetada desde 1960 deve alguma coisa ao ALGOL. De fato, a maioria é uma descendente direta ou indireta; os exemplos são PL/I, SIMULA 67, ALGOL 68, C, Pascal, Ada, C++ e Java.

O esforço de projeto do ALGOL 58/ALGOL 60 inclui uma longa lista de "primeiros". Foi a primeira vez que um grupo internacional tentou projetar uma linguagem de programação, a primeira projetada para ser independente de máquina. Também foi a primeira cuja sintaxe descreveu-se formalmente. Esse bem-sucedido uso do formalismo BNF iniciou diversos campos importantes na ciência da computação: linguagens formais, teoria de análise e projeto de compiladores. Finalmente, a estrutura do ALGOL 60 afetou a arquitetura de máquina. No exemplo mais gritante disso, usou-se uma extensão dessa linguagem como a linguagem de sistemas de uma série de computadores de grande porte, as máquinas Burroughs B5000, B6000 e B7000, projetadas com uma pilha em hardware para implementar eficientemente a estrutura em blocos e os procedimentos recursivos da linguagem.

No outro lado da moeda, o ALGOL 60 jamais conseguiu um uso generalizado ou até mesmo significativo nos Estados Unidos. Mesmo na Europa, esta nunca se tornou a linguagem predominante. Há uma série de motivos para a falta de aceitação. Por um lado, alguns dos recursos do ALGOL 60 ficaram flexíveis demais; eles tornavam o entendimento difícil e a implementação ineficiente. O melhor exemplo é o método de passar parâmetros por nome a subprogramas, que será explicado no Capítulo 9. As dificuldades para implementar o ALGOL 60 são evidenciadas pela afirmação de Rutishauser em 1967, segundo o qual poucas implementações incluíam a linguagem ALGOL 60 completa (Rutishauser, 1967, p. 8).

A falta de instruções de entrada e saída na linguagem era outro importante motivo para sua falta de aceitação. A entrada/saída dependente de implementação tornava os programas difíceis de serem portados para outros computadores.

Uma das contribuições mais importantes para a ciência da computação associada ao ALGOL 60, a BNF, também foi um fator em sua falta de aceitação. Ainda que a BNF agora seja considerada um meio elegante de descrição da sintaxe, para o mundo de 1960 ela parecia estranha e complicada.

Por fim, embora houvesse muitos outros problemas, o entrincheiramento do FORTRAN entre os usuários e a falta de suporte por parte da IBM provavelmente foram os fatores mais importantes para que o ALGOL 60 não tenha disseminado seu uso.

O esforço do ALGOL 60 jamais completou-se, em termos de que as ambigüidades e as obscuridades sempre fizeram parte da descrição da linguagem (Knuth, 1967).

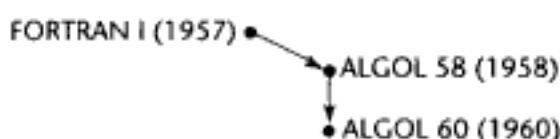
Veja, a seguir, um exemplo de programa em ALGOL 60.

```

comment Programa de Exemplo ALGOL 60
Entrada: Um número inteiro, complis, em que complis é menor
          do que 100, seguido de complis valores inteiros
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 complis, contador, soma, media, resultado;
  soma := 0;
  resultado := 0;
  readint (complis);
  if (complis > 0) ∧ (complis < 100) then
    begin
comment Leia a entrada em um vetor e compute a média;
    for contador := 1 step 1 until complis do
      begin
        readint (intlist[contador]);
        soma := soma + intlist[contador]
      end;
comment Compute a média;
    media := soma / complis;
comment Conte os valores de entrada que são > do que a média;
    for contador := 1 step 1 until complis do
      if intlist[contador] > media
        then resultado := resultado + 1;
comment Imprima o resultado;
    printstring("O número de valores > média é:");
    printint (resultado)
    end
  else
    printstring ("Erro - tamanho da lista de entrada não é
                  legal");
  end
end

```

A linhagem do ALGOL 60 é mostrada na Figura 2.3.



**FIGURA 2.3** Genealogia do ALGOL 60.

## 2.6 Informatizando Registros Comerciais: COBOL

A história do COBOL é realmente estranha. Apesar de ter sido usada mais do que qualquer outra linguagem de programação, o COBOL teve pouco efeito sobre o projeto das linguagens subsequentes, exceto em relação à PL/I. Talvez ela ainda seja a linguagem de uso mais generalizado, ainda que seja difícil ter certeza. Talvez o motivo mais importante pelo qual o COBOL tenha tido pouca influência é que poucos tentaram projetar uma nova linguagem para aplicações comerciais desde que ele surgiu. Isso pode muito bem ser um tributo a quão bem as capacidades do COBOL satisfazem as necessidades de sua área de aplicação. Outra razão é que ocorreu um grande crescimento na computação comercial ao longo dos últimos 15 anos nos pequenos negócios e muito pouco desenvolvimento de software foi levado a efeito nessa área. Ao contrário, a maioria dos softwares é comprada como pacotes de prateleira para várias aplicações comerciais gerais.

### 2.6.1 Embasamento Histórico

A origem do COBOL é bastante semelhante à do ALGOL 60, uma vez que foi projetado por um comitê de pessoas que se reuniam por períodos de tempo relativamente curtos. A situação da computação comercial naquela época, em 1959, era semelhante à situação da computação científica diversos anos antes, quando o FORTRAN começou a ser projetado. Uma linguagem compilada para aplicações comerciais, a FLOW-MATIC, foi implementada em 1957, mas pertencia a um fabricante, a UNIVAC, e atendia apenas os computadores dessa empresa. Outra linguagem, a AIMACO, era usada pela Força Aérea dos Estados Unidos, mas consistia somente de uma pequena variação da FLOW-MATIC. A IBM projetara uma linguagem de programação para aplicações comerciais, a COMTRAN (COMMercial TRANslator), mas ela ainda não havia sido implementada. Diversos outros projetos estavam sendo planejados.

### 2.6.2 FLOW-MATIC

As origens da FLOW-MATIC merecem, pelo menos, uma breve discussão, porque ela foi a principal progenitora do COBOL. Em dezembro de 1953, Grace Hopper, da Remington-Rand UNIVAC, produziu uma proposta, de fato, profética. Ela sugeriu que "programas matemáticos deveriam ser escritos em notação matemática, programas de processamento de dados deveriam ser escritos com instruções em inglês" (Wexelblat, 1981, p. 16). Infelizmente, em 1953, seria impossível convencer não-programadores que um computador poderia ser construído para entender palavras inglesas. Foi somente em 1955 que uma proposta semelhante teve alguma esperança de financiamento pela administração da UNIVAC e, mesmo assim, houve necessidade de um sistema protótipo para fazer o convencimento final. Parte desse processo de venda envolveu compilar e rodar um pequeno programa, primeiro usando palavras-chave inglesas, depois usando palavras-chave francesas e depois usando palavras-chave alemãs. Essa demonstração foi considerada notável pela administração da UNIVAC e constituiu-se no fator primordial na aceitação da proposta de Hopper.

### 2.6.3 Projeto do COBOL

A primeira reunião formal visando a uma linguagem comum para aplicações comerciais, patrocinada pelo Departamento de Defesa, foi realizada no Pentágono nos dias 28 e 29 de maio de 1959 (exatamente um ano depois do encontro sobre o ALGOL em Zurique). O grupo chegou ao consenso que a linguagem, então chamada CBL (de *Common Business Language*), deveria ter algumas características gerais. Ela deveria usar o inglês o máximo possível, não obstante alguns defenderem uma notação mais matemática. Seria fácil de usar, mesmo à custa de ser menos poderosa, a fim de ampliar a base daqueles que poderiam programar computadores. Além de facilitar o uso, acreditava-se que a opção pelo inglês permitiria que os gerentes lessem os programas. Por fim, o projeto não deveria ser demasiadamente restringido pelos problemas de sua implementação.

A preocupação com a necessidade de apressar o passo para a criação dessa nova linguagem universal predominou na reunião, uma vez que muito trabalho já estava sendo feito para criar novas linguagens comerciais. Além das linguagens existentes, a RCA e a Sylvania estavam trabalhando em aplicações comerciais. Se uma linguagem universal não fosse projetada logo, sua aceitação posterior seria bem mais difícil. Baseando-se nisso, foi concluído que deveria haver um estudo rápido das linguagens existentes. Para tal tarefa, formou-se o *Short Range Committee*.

Houve as primeiras decisões para separar as instruções da linguagem em duas categorias — descrição de dados e operações executáveis — e fazer com que as instruções das duas categorias residissem em diferentes partes do programa. Um dos grandes debates do *Short Range Committee* concentrou-se na inclusão de subscritos. Muitos membros do comitê argumentaram que os subscritos eram demasiadamente complexos para o pessoal do processamento de dados, com pouca desenvoltura em matemática. Argumentos semelhantes giraram em torno da dúvida sobre a inclusão ou não de expressões aritméticas. O relatório final do *Short Range Committee*, concluído em dezembro de 1959, descrevia uma linguagem mais tarde chamada de COBOL 60.

As especificações de linguagem para o COBOL 60, publicadas pelo Escritório de Imprensa do Governo em abril de 1960 (Departamento de Defesa, 1960), foram descritas como “iniciais.” Versões revisadas foram publicadas em 1961 e 1962 (Departamento de Defesa, 1961, 1962). A linguagem foi padronizada pelo grupo American National Standards Institute (ANSI) em 1968. O ANSI padronizou as duas revisões seguintes em 1974 e 1985. A linguagem continua a evoluir atualmente.

### 2.6.4 Avaliação

O COBOL deu origem a uma série de conceitos novos, alguns dos quais apareceram, por fim, em outras linguagens. Por exemplo, o verbo *DEFINE* do COBOL 60 foi a primeira construção em linguagem de alto nível para macros. O que é mais importante, as estruturas de dados hierárquicas, que surgiram pela primeira vez na Plankalkül, foram implementadas pela primeira vez no COBOL. E elas foram incluídas em, virtualmente, toda linguagem imperativa projetada desde então. O COBOL também foi a primeira linguagem a permitir que nomes fossem verdadeiramente conotativos, porque permitia tanto nomes longos (até 30 caracteres) como caracteres conectores de palavras (traços).

Acima de tudo, a divisão de dados é a parte forte de projeto do COBOL, enquanto que a divisão de procedimentos é relativamente fraca. Toda variável é definida detalhadamente na divisão de dados, incluindo o número de dígitos decimais e a localização da vírgula decimal implícita. Os registros de arquivo também são descritos com esse nível de detalhe,

assim como as linhas a serem enviadas para uma impressora, o que torna o COBOL ideal para imprimir relatórios de contabilidade. Talvez a fragilidade mais importante da divisão de procedimentos situe-se em sua falta de funções. As versões do COBOL anteriores ao padrão de 1974 também não permitiam subprogramas com parâmetros.

Nosso comentário final sobre o COBOL: foi a primeira linguagem de programação cujo uso foi imposto pelo Departamento de Defesa (DoD). Essa obrigação veio depois de seu desenvolvimento inicial, porque o COBOL não foi projetado especificamente para o DoD. Apesar de seus méritos, ele provavelmente não teria sobrevivido sem essa obrigação. O mau desempenho dos primeiros compiladores simplesmente o tornava muito caro para ser usado. Por fim, é claro, as pessoas aprenderam mais sobre como projetar compiladores, e os computadores tornaram-se muito mais rápidos, mais baratos e tinham memórias muito maiores. Juntos, esses fatores transformaram o COBOL em um grande sucesso, dentro e fora do DoD. Seu surgimento levou à mecanização eletrônica da contabilidade, o que se constituiu em uma importante revolução sob todos os aspectos.

O exemplo seguinte é de um programa em COBOL. Ele lê um arquivo chamado BAL-FWD-FILE que contém informações de controle de estoque sobre determinado conjunto de itens. Entre outras coisas, cada registro de item inclui a quantidade que se tem atualmente em mãos (BAL-ON-HAND) e o ponto de reordenamento do item (BAL-REORDER-POINT). O ponto de reordenamento é o número limite de itens que se tem à mão, a partir do qual mais itens devem ser ordenados. O programa produz uma lista de itens que devem ser reordenados 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.

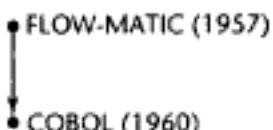
130-PRINT-REORDER-LINE.
    MOVE SPACE           TO REORDER-LINE.
```

```

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.

```

A linhagem do COBOL é mostrada na Figura 2.4.



**FIGURA 2.4** Genealogia do COBOL.

## 2.7 O Início do Compartilhamento de Tempo (*Timesharing*): BASIC

O BASIC (Mather e Waite, 1971) é outra linguagem de programação que desfrutou de um uso generalizado, mas obteve pouco respeito. À semelhança do COBOL, ele foi muito ignorado pelos cientistas da computação. Além disso, como o COBOL, em suas primeiras versões, o BASIC era deselegante e incluía somente um escasso conjunto de instruções de controle.

O BASIC foi muito popular nos microcomputadores no final da década de 70 e início da década de 80. Duas dessas principais características do BASIC levaram à disseminação de seu uso. Ele é fácil para os iniciantes aprenderem, especialmente aqueles que não estão voltados para a ciência, e seus dialetos menores podem ser implementados em computadores com memórias muito pequenas. Houve um ressurgimento no uso do BASIC como o Visual BASIC (Microsoft, 1991) no início da década de 90.

### 2.7.1 Processo de Projeto

O BASIC (*Beginner's All-purpose Symbolic Instruction Code*) foi projetado no *Dartmouth College* (agora, *Dartmouth University*) em New Hampshire, por dois matemáticos, John Kemeny e Thomas Kurtz, que estavam envolvidos, no início da década de 60, em produzir compiladores para uma variedade de dialetos do FORTRAN e do ALGOL 60. Seus alunos de ciências tiveram pouco trabalho para aprender ou para usar essas linguagens em seus trabalhos.

Porém, Dartmouth era, antes de mais nada, uma instituição de artes liberais, em que os alunos de ciências e de engenharia compunham somente cerca de 25% do corpo discente. Decidiu-se, na primavera de 1963, projetar uma nova linguagem, especialmente para estudantes de artes liberais. A nova linguagem usaria terminais como o método de acesso ao computador. As metas do sistema eram:

1. Deve ser fácil de aprender e de ser usada pelos estudantes das áreas não-científicas.

2. Deve ser agradável e amigável.
3. Deve oferecer rápido retorno para trabalho de casa.
4. Deve permitir acesso livre e privado.
5. Deve considerar o tempo de usuário mais importante do que o tempo de computador.

A última meta era, de fato, um conceito revolucionário. Ela baseava-se, pelo menos parcialmente, na crença de que os computadores iriam se tornar significativamente mais baratos à medida que o tempo passasse, o que, de fato, aconteceu.

A combinação da segunda, da terceira e da quarta metas levaram ao aspecto de tempo compartilhado do BASIC. Somente com acesso individual por meio de terminais por numerosos usuários simultâneos é que tais metas poderiam ser atingidas no início da década de 60.

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 iniciaram-se no outono de 1963. Às 4h do dia 1º de maio de 1964, o primeiro programa usando o BASIC de tempo compartilhado foi digitado e posto em execução. Em junho, o número de terminais no sistema cresceu para 11; no outono, subiu para 20.

### 2.7.2 Visão Geral da Linguagem

A versão original do BASIC era muito pequena e, estranhamente, não era interativa; não havia nenhum significado em obter dados de entrada do terminal. Os programas eram digitados, compilados e executados em uma espécie de modo orientado a lote. O BASIC original tinha somente 14 tipos de instrução diferentes e um único tipo de dados, o número real. O tipo era referenciado como "numbers" uma vez que se acreditava que poucos usuários apreciariam a diferença entre tipos inteiros e reais. Acima de tudo, ela era uma linguagem muito limitada, ainda que muito fácil de aprender.

### 2.7.3 Avaliação

O aspecto mais importante do BASIC original é que ele foi o primeiro método amplamente usado de acesso por terminal remoto a um computador. Os terminais haviam começado a tornar-se disponíveis naquela época. Antes disso, a maioria dos programas era introduzida nos computadores ou por meio de cartões perfurados ou por fita de papel.

Grande parte do projeto do BASIC veio do FORTRAN, com uma pequena influência da sintaxe do ALGOL 60. Posteriormente, ela cresceu de muitas maneiras, com pouco ou nenhum esforço para padronizá-la. O *American National Standards Institute* publicou um padrão Minimal BASIC (ANSI, 1978b), mas esse representava somente o mínimo dos recursos da linguagem. De fato, o BASIC original era muito semelhante à Minimal BASIC.

A Digital Equipment Corporation usou uma versão bastante elaborada do BASIC chamada BASIC-PLUS para escrever partes significativas de seu maior sistema operacional para os computadores PDP-11, RSTS na década de 70, ainda que isso possa parecer surpreendente.

O BASIC tem sido criticado, entre outras coisas, pela estrutura ruim dos programas nela escritos. Segundo nossos critérios de avaliação, a linguagem de fato não se sai bem. Obviamente, as primeiras versões da linguagem não se destinavam e não deviam ser usadas para programas sérios de qualquer tamanho significativo. As versões posteriores são muito melhor adequadas a essas tarefas.

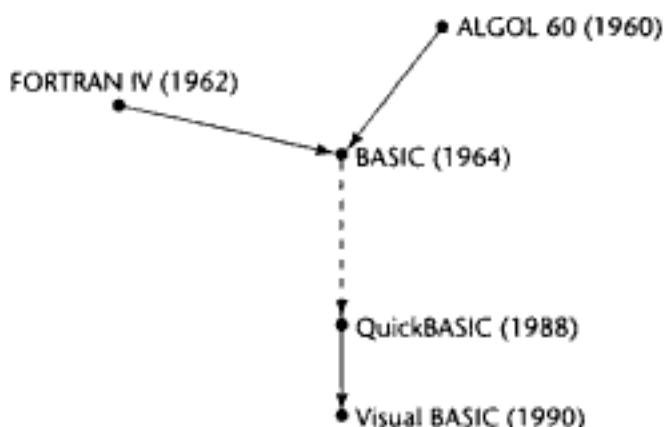
As razões mais prováveis para o sucesso do BASIC são a facilidade com que ele pode ser aprendido e a sua facilidade de implementação, mesmo em computadores muito pequenos.

Duas das versões contemporâneas do BASIC, agora amplamente usadas, são o QuickBASIC (Bradley, 1989) e o Visual BASIC. Ambas rodam em PCs. O Visual BASIC baseia-se no QuickBASIC, mas foi projetado para desenvolver sistemas de software que tenham interfaces com o usuário providas de janelas. O Visual BASIC também tem sido usado como uma linguagem de scripting para programação CGI.

Eis um exemplo de programa em QuickBASIC:

```
REM Exemplo de Programa em QuickBASIC
REM Entrada: Um inteiro, compris, em que compris é
REM           menor do que 100, seguido de compris valores
REM Saída:   O número dos valores de entrada que são maiores
REM           do que a média de todos os valores de entrada
    DIM intlist(99)
    resultado = 0
    soma = 0
    INPUT compris
    IF compris > 0 AND compris < 100 THEN
        REM Leia a entrada em um vetor e compute a soma
        FOR contador = 1 TO compris
            INPUT intlist(contador)
            soma = soma + intlist(contador)
        NEXT contador
        REM Compute a média
        media = soma / compris
        REM Conte o número de valores de entrada que são > do que
        a média
        FOR contador = 1 TO compris
            IF intlist(contador) > media
                THEN resultado = resultado + 1
        NEXT contador
        REM Imprima o resultado
        PRINT "O número de valores que são > do que a média é:";
              resultado
        ELSE
            PRINT "Erro - o tamanho da lista de entrada não é válido"
        END IF
    END
```

A linhagem do BASIC é mostrada na Figura 2.5 (ver p. 76).



**FIGURA 2.5** Genealogia do BASIC.

## 2.8 Tudo para Todos: PL/I

A PL/I representa a primeira tentativa em grande escala de projetar uma linguagem que poderia ser usada para um amplo espectro de áreas de aplicação. Todas as linguagens anteriores e a maioria das subsequentes concentraram-se em uma área de aplicação particular, como, por exemplo, as ciências, a inteligência artificial ou os negócios.

### 2.8.1 Embasamento Histórico

Assim como o FORTRAN, a PL/I foi desenvolvida como um produto IBM. No início da década de 60, os usuários de computadores da indústria estabeleceram-se em dois campos separados e bem diferentes. Do ponto de vista da IBM, os programadores científicos podiam usar os computadores 7090 de grande porte ou o 1620 IBM de pequeno porte. Esse grupo usava os números reais e as matrizes extensivamente. O FORTRAN era a linguagem principal, embora outras linguagens de montagem também fossem usadas. Eles tinham seu próprio grupo de usuários, o SHARE, e tinham pouco contato com qualquer um que trabalhasse em aplicações comerciais.

Para aplicações comerciais, os computadores IBM 7080 grandes ou os 1401 pequenos eram utilizados. Eles precisavam dos tipos de dados decimais e de cadeias de caracteres, bem como de facilidades de entrada e de saída elaboradas e eficientes. Eles usavam o COBOL, ainda que no início de 1963, quando a história da PL/I se iniciou, a conversão da linguagem de montagem para o COBOL estivesse longe de ser completa. Tal categoria de usuários também tinha seu próprio grupo de usuários, o GUIDE, e raramente contatava com usuários científicos.

No início de 1963, os planejadores da IBM perceberam o início de uma mudança nessa situação. Os dois grupos, amplamente separados, movimentavam-se um em direção ao outro de uma maneira que se imaginava certa para criar problemas. Os cientistas começavam a reunir grandes arquivos de dados para serem processados que exigiam facilidades de entrada e de saída mais sofisticadas e eficientes. O pessoal das aplicações comerciais começava a usar análise da regressão para construir sistemas de informação gerencial, os

quais exigiam dados com números reais e matrizes. Começou a parecer que as instalações de computação logo exigiriam dois computadores separados, suportando duas linguagens de programação muito diferentes.

Essas percepções muito naturalmente levaram ao conceito de projetar-se um computador universal único capaz de fazer aplicações tanto com números reais como com aritmética decimal e, por conseguinte, aplicações científicas e comerciais. Assim, nascia o conceito da linha de computadores IBM System/360. Juntamente com isso, veio a idéia de uma linguagem de programação que pudesse ser usada tanto para aplicações comerciais como científicas. Para obterem uma boa medida, a programação de sistemas e o processamento de listas estavam entre suas capacidades. Portanto, a nova linguagem destinava-se a substituir o FORTRAN, o COBOL, a LISP e as aplicações de sistemas da linguagem de montagem.

### 2.8.2 Processo de Projeto

O esforço de projeto iniciou-se quando a IBM e o SHARE formaram a *Advanced Language Development Committee* do SHARE FORTRAN Project em outubro de 1963. Esse novo comitê reuniu-se rapidamente e formou um subcomitê chamado Comitê 3 × 3, assim chamado porque ele tinha três membros da IBM e três do SHARE. O Comitê 3 × 3 reunia-se durante três ou quatro dias a cada duas semanas para projetar a linguagem.

Como aconteceu com o *Short Range Committee* para o COBOL, o projeto inicial foi programado para ser concluído em um tempo notavelmente curto. Aparentemente, independentemente do escopo do esforço de projeto de uma linguagem, no início da década de 60 a convicção prevalecente era de que isso poderia ser feito em três meses. A primeira versão da PL/I, que depois recebeu o nome de FORTRAN VI, pretendia ser concluída em dezembro, menos de três meses depois da formação do comitê. O comitê pleiteou com sucesso em duas ocasiões diferentes por extensões, passando a data limite para janeiro e depois para o final de fevereiro de 1964.

O conceito de projeto inicial era o de que a nova linguagem seria uma extensão do FORTRAN IV, mantendo a compatibilidade, mas essa meta foi deixada de lado juntamente com o nome FORTRAN VI. Até 1965, a linguagem era conhecida como NPL, uma sigla de *New Programming Language*. O primeiro relatório publicado sobre a NPL foi apresentado no encontro do SHARE de março de 1964. Uma descrição mais completa seguiu-se em abril, e a versão que de fato seria implementada foi publicada em dezembro de 1964 (IBM, 1964) pelo grupo de compiladores do IBM Hursley Laboratory, na Inglaterra, escolhido para fazer a implementação. Em 1965, o nome mudou para PL/I para evitar a confusão do nome NPL com o National Physical Laboratory da Inglaterra. Se o compilador tivesse sido desenvolvido fora do Reino Unido, o nome poderia ter sido mantido.

### 2.8.3 Visão Geral da Linguagem

Talvez a melhor definição da PL/I em uma única sentença seja a inclusão das melhores partes, então consideradas, do ALGOL 60 (recursão e estrutura em bloco), do FORTRAN IV (compilação separada com comunicação por meio de dados globais) e do COBOL 60 (estruturas de dados, entrada/saída e facilidades de geração de relatório), juntamente com algumas construções novas, todas mescladas de alguma maneira. Não tentaremos, mesmo de maneira abreviada, discutir todos os recursos da linguagem ou mesmo suas construções mais controversas. Em vez disso, mencionaremos brevemente parte da contribuição da linguagem para o universo de conhecimento das linguagens de programação.

A PL/I foi a primeira linguagem de programação a ter as seguintes facilidades:

- Permitiu-se que os programas criassem tarefas executadas concorrentemente. Ainda que isso tenha sido uma boa idéia, ela foi mal desenvolvida na PL/I.
- Tornou-se possível detectar e manipular 23 diferentes tipos de exceções, ou erros em tempo de execução.
- Permitiu-se que procedimentos fossem usados recursivamente, mas a capacidade podia ser desativada, permitindo um código mais eficiente para procedimentos não-recursivos.
- Ponteiros foram incluídos como um tipo de dados.
- Seções transversais de matrizes podiam ser referenciadas. Por exemplo, a terceira linha da matriz podia ser referenciada como se fosse um vetor.

#### 2.8.4 Avaliação

Qualquer avaliação da PL/I deve iniciar-se reconhecendo a ambição do esforço de projeto. Em retrospecto, parece ingênuo pensarmos que tantas construções poderiam ter sido combinadas de maneira bem-sucedida. Porém, esse julgamento deve ser temperado ao reconhecermos que havia pouca experiência em projetos de linguagem na época. De maneira geral, o projeto da PL/I baseava-se na premissa de que qualquer construção usada e de possível implementação deveria ser incluída, com insuficiente preocupação a respeito de como os muitos recursos iriam se comportar quando colocados juntos. Edsger Dijkstra, em sua *Turing Award Lecture* (Dijkstra, 1972), fez uma das mais fortes críticas à complexidade da PL/I: "Eu absolutamente não consigo ver como podemos manter nossos programas em crescimento firmemente dentro de nossa compreensão intelectual quando, por seu claro estilo barroco, a linguagem de programação — nossa ferramenta básica, imaginem! — já escapa de nosso controle intelectual".

Além do problema da complexidade devido ao seu grande tamanho, a PL/I padecia de um grande número daquilo que, agora, consideramos ser construções mal projetadas. Entre essas, havia os ponteiros, a manipulação de exceções e a concorrência, apesar da construção ser algo novo em cada um desses casos.

Em termos de uso, a PL/I deve ser considerada, pelo menos, um sucesso parcial. Na década de 70, ela desfrutou de um uso significativo tanto em aplicações comerciais como científicas. Ela também foi amplamente usada, naquela época, como um veículo de ensino, principalmente em diversas formas restritas, como a PL/C (Cornell, 1977) e PL/CS (Conway e Constable, 1976).

Eis um exemplo de programa em PL/I:

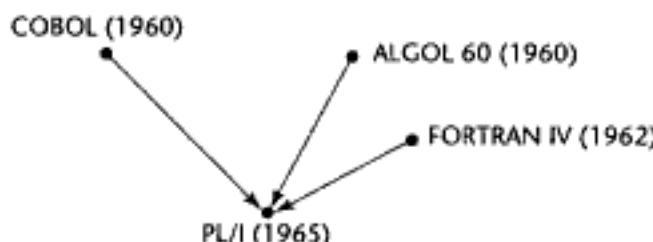
```
/* EXEMPLO DE PROGRAMA EM PL/I
ENTRADA: UM INTEIRO, COMPLISTA, EM QUE COMPLISTA É
MENOR DO QUE 100, SEGUIDO DE COMPLISTA VALORES INTEIROS
SAÍDA: O NÚMERO DE VALORES DE ENTRADA QUE SÃO MAIORES
DO QUE A MÉDIA DE TODOS OS VALORES DE ENTRADA */
PLIEX: PROCEDURE OPTIONS (MAIN);
DECLARE INTLIST (1:99) FIXED;
DECLARE (COMPLISTA, CONTADOR, SOMA, MEDIA, RESULTADO) FIXED;
SOMA = 0;
RESULTADO = 0
GET LIST (COMPLISTA);
```

```

IF (COMPLISTA > 0) & (COMPLISTA < 100) THEN
    DO;
    /* LEIA OS DADOS DE ENTRADA EM UM VETOR E COMPUTE A SOMA */
    DO CONTADOR = 1 TO COMPLISTA;
        GET LIST (INTLIST (CONTADOR));
        SOMA = SOMA + INTLIST (CONTADOR);
    END;
    /* COMPUTE A MÉDIA */
    MEDIA = SOMA / COMPLISTA;
    /* CONTE O NÚMERO DE VALORES QUE SÃO > DO QUE A MÉDIA */
    DO CONTADOR = 1 TO COMPLISTA;
        IF INTLIST (CONTADOR) > MEDIA THEN
            RESULTADO = RESULTADO + 1;
    END;
    /* IMPRIMA O RESULTADO */
    PUT SKIP LIST (' O NÚMERO DE VALORES > QUE A MÉDIA É:');
    PUT LIST (RESULTADO);
    END;
ELSE
    PUT SKIP LIST (' ERRO - TAMANHO DA LISTA DE ENTRADA É
ILEGAL');
END PLIEX;

```

A linhagem da PL/I é mostrada na Figura 2.6.



**FIGURA 2.6** Genealogia da PL/I.

## 2.9 Duas Primeiras Linguagens Dinâmicas: APL e SNOBOL

A estrutura desta seção é diferente daquela usada nas outras seções do capítulo, porque as linguagens aqui discutidas são diferentes. Nem a APL, nem a SNOBOL basearam-se em qualquer linguagem anterior, e nenhuma delas teve muita influência sobre as linguagens principais mais tarde — não obstante a linguagem J basear-se na APL, a ICON (Griswold e Griswold, 1983) basear-se na SNOBOL. Tanto a APL como a SNOBOL são usadas, no livro, para ilustrar e para comparar alguns de seus recursos com os das linguagens mais convencionais. Eis porque elas serão discutidas aqui brevemente.

Em suas aparências e em seus propósitos, a APL e a SNOBOL são muito diferentes. Elas compartilham duas características comuns, entretanto: tipificação dinâmica e alocação

de armazenagem dinâmica. As variáveis em ambas são fundamentalmente sem tipo. Uma variável adquire um tipo quando a ela é atribuído um valor, nesse momento assume o tipo do valor atribuído. A memória é alocada a uma variável somente quando a ela é atribuído um valor, porque antes não havia nenhuma maneira de saber o tamanho necessário para tal armazenamento.

### 2.9.1 Origens e Características da APL

A APL (Polinka e Pakin, 1975) foi projetada por volta de 1960, por Kenneth E. Iverson, na IBM. Originalmente, ela não foi projetada para ser uma linguagem de programação implementada, mas, pelo contrário, destinava-se a ser um veículo para descrever a arquitetura de computador. A APL foi descrita, pela primeira vez, no livro do qual ela recebeu seu nome, *A Programming Language* (Iverson, 1962). Em meados da década de 60, a primeira implementação da APL foi desenvolvida na IBM.

A APL tem um grande número de operadores poderosos, que criaram um problema para os implementadores. O primeiro meio de usar a APL era por intermédio dos terminais de impressão da IBM. Tais terminais tinham esferas de impressão especiais que possuíam o conjunto de caracteres ímpar exigido pela linguagem. Uma razão para que a APL tenha tantos operadores é a permissão de que matrizes sejam manipuladas como se fossem variáveis escalares. Por exemplo, a transposição de qualquer matriz é feita com um operador único. A grande coleção de operadores proporciona uma expressividade muito grande, mas também torna difícil a leitura dos programas APL. Muitas pessoas imaginam a APL como uma linguagem que é melhor usada para programação “jogue fora”. Os programas devem ser descartados depois de serem usados porque sua manutenção é difícil, ainda que possam ser escritos rapidamente.

A APL existe há aproximadamente 35 anos e ainda hoje é usada, ainda que não amplamente. Além disso, ela não foi muito modificada ao longo de sua existência.

### 2.9.2 Origens e Características da SNOBOL

A SNOBOL (pronuncia-se “snoubol”) (Griswold et al., 1971) foi projetada no início da década de 60 por três pessoas do Bell Laboratories: D. J. Farber, R. E. Griswold e F. P. Polensky (Farber et al., 1964). Seu projeto visou ao processamento de texto. O coração da SNOBOL é uma coleção de operações poderosas para casamento de padrões de cadeias. Uma das suas primeiras aplicações foi para escrever editores de texto. Sua natureza dinâmica torna-a mais lenta do que algumas outras linguagens, por isso agora raramente ela é usada para esses programas. Porém, a SNOBOL ainda é uma linguagem viva e com suporte, usada para uma variedade de tarefas de processamento de texto em um grande número de diferentes áreas de aplicação.

## 2.10 A Origem da Abstração de Dados: SIMULA 67

Alguns conceitos que a SIMULA 67 introduziu tornaram-na importante, ainda que ela jamais tenha obtido um uso generalizado e tenha tido pouco impacto nos programadores e na computação.

### 2.10.1 Processo de Projeto

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

Assim que concluíram a implementação da SIMULA I, Nygaard e Dahl iniciaram esforços para estender a linguagem, adicionando recursos inteiramente novos e modificando algumas construções existentes, a fim de torná-la útil para aplicações de propósito mais geral. O resultado foi a SIMULA 67, cujo projeto apresentou-se publicamente pela primeira vez em março de 1967 (Dahl e Nygaard, 1967). Discutiremos somente a SIMULA 67, mesmo que alguns dos seus recursos interessantes também estejam na SIMULA I.

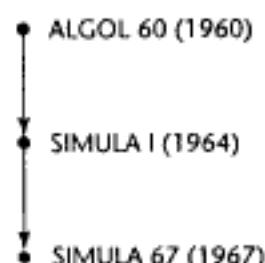
### 2.10.2 Visão Geral da Linguagem

A SIMULA 67 é uma extensão do ALGOL 60, pegando tanto a estrutura em bloco como a estrutura de instrução de controle dessa linguagem. A principal deficiência do ALGOL 60 (e de outras linguagens da época) para aplicações de simulação é o projeto de seus subprogramas. A simulação exige subprogramas com permissão para reiniciar na posição em que pararam anteriormente. Subprogramas com esse tipo de controle são conhecidos como co-rotinas porque os subprogramas chamadores e os chamados têm uma relação um tanto igual entre si, ao contrário da relação hierárquica, que normalmente há nas linguagens imperativas.

Para oferecer suporte para co-rotinas na SIMULA 67, desenvolveu-se a construção de classe. Eis um avanço importante, porque nossas idéias a respeito de abstração de dados iniciaram-se com ele. A idéia básica de uma classe é que uma estrutura de dados e as rotinas que a manipulam são empacotadas juntas. Além disso, uma definição de classe é somente um modelo para uma estrutura de dados e, como tal, é distinta de uma instância de classe; assim, um programa pode criar e usar qualquer número de instâncias de uma classe particular. As instâncias de classe podem conter dados locais. Elas também podem incluir um código a ser executado no momento da criação, que pode inicializar alguma estrutura de dados da instância de classe.

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

A linhagem da SIMULA 67 é mostrada na Figura 2.7.



**FIGURA 2.7** Genealogia da SIMULA 67.

## 2.11 Projeto Ortogonal: ALGOL 68

O ALGOL 68 foi a fonte de diversas idéias novas no projeto de linguagens, algumas das quais subsequentemente adotadas por outras linguagens. Nós a incluímos aqui por essa razão, ainda que seu uso nunca tenha sido generalizado tanto na Europa como nos Estados Unidos. Neste capítulo, discutiremos somente duas de suas contribuições mais importantes, uma vez que algumas construções do ALGOL 68 serão discutidas em capítulos posteriores.

### 2.11.1 Processo de Projeto

O desenvolvimento da família ALGOL não se encerrou quando o relatório revisado apareceu em 1962, mesmo que tenham transcorrido seis anos até que a próxima iteração de projeto fosse publicada. A linguagem resultante, o ALGOL 68 (van Wijngaarden et al., 1969), foi drasticamente diferente de sua predecessora.

Uma das inovações mais interessantes do ALGOL 68 é um de seus principais critérios de projeto: a ortogonalidade. Lembre-se de nossa discussão sobre a ortogonalidade no Capítulo 1. O uso da ortogonalidade resultou em diversos recursos inovadores do ALGOL 68, dois dos quais serão descritos na próxima seção.

### 2.11.2 Visão Geral da Linguagem

Um resultado importante da ortogonalidade no ALGOL 68 foi sua inclusão dos tipos de dados definidos pelo usuário. As linguagens anteriores, como o FORTRAN, incluíam somente algumas estruturas de dados básicas. A PL/I incluía um grande número de estruturas de dados, que tornavam seu aprendizado e sua implementação mais difíceis, mas, evidentemente, ela não podia oferecer uma estrutura de dados apropriada para todas as aplicações.

A abordagem do ALGOL 68 à estrutura de dados deveria oferecer alguns tipos e estruturas primitivas e permitir ao usuário combiná-los em um grande número de diferentes estruturas. Essa provisão para tipos de dados definidos pelo usuário foi transportada até certo ponto para todas as grandes linguagens imperativas projetadas desde então. Os tipos de dados definidos pelo usuário são valiosos porque permitem que o usuário projete abstrações de dados que se encaixam muito estreitamente em problemas particulares. Todos os aspectos da tipificação de dados serão discutidos no Capítulo 6.

Como outra iniciativa pioneira na área dos tipos de dados, o ALGOL 68 introduziu o tipo matriz dinâmica, que será chamada dinâmica no monte (*heap-dynamic*) implícita no Capítulo 5. Uma matriz dinâmica é aquela em que a declaração não especifica limites de subscrito. As atribuições a uma matriz dinâmica provocam a alocação da memória exigida. No ALGOL 68 as matrizes dinâmicas são chamadas matrizes **flex**. Por exemplo, a instrução

```
flex [1:0] int lista
```

<sup>N. de T. Heap: Monte. Em programação, o espaço de memória livre disponível para o programa.</sup>

declara que `lista` é uma matriz dinâmica de números inteiros com um único subscrito cujo limite inferior é 1, mas ela não aloca nenhum espaço para armazenamento. A atribuição agregada

```
lista ::= (3, 5, 6, 2)
```

faz com que seja alocado espaço suficiente para que `lista` contenha quatro inteiros, mudando efetivamente seus limites para `[1:4]`.

### 2.11.3 Avaliação

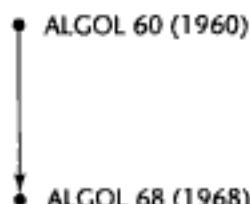
O ALGOL 68 inclui um número significativo de recursos que não eram usados anteriormente. O uso que ele faz da ortogonalidade, que alguns consideram exagerado, foi revolucionário. Muitos dos recursos introduzidos no ALGOL 68 tornaram-se parte de linguagens subsequentes.

O ALGOL 68 repetiu um dos pecados do ALGOL 60, e isso foi um fator que concorreu para sua falta de aceitação generalizada. A linguagem havia sido descrita pelo uso de uma metalinguagem elegante e concisa, mas também desconhecida. Antes que uma pessoa pudesse ler o documento que a descrevia (van Wijngaarden et al., 1969), ela tinha de aprender a nova metalinguagem, chamada gramática van Wijngaarden. Para piorar as coisas, os projetistas inventaram uma coleção de palavras para explicar a gramática e a linguagem. Por exemplo, as palavras-chave são chamadas indicantes (*indicants*), a extração de subcadeias é chamada *trimming*, e o processo de execução de procedimentos é chamado coerção de *deproceduring*, que poderia ser brando, firme ou qualquer outra coisa.

É natural comparar o projeto da PL/I com o do ALGOL 68. Este último obteve capacidade de escrita pelo princípio da ortogonalidade: com alguns conceitos primitivos e com o uso irrestrito de alguns mecanismos de combinação, a PL/I obteve capacidade de escrita ao incluir um grande número de construções fixas. O ALGOL 68 estendeu a simplicidade elegante do ALGOL 60, enquanto que a PL/I simplesmente colocou juntos os recursos de diversas linguagens para atingir suas metas. Obviamente, deve-se ter em mente que a meta da PL/I era fornecer uma ferramenta unificada para uma ampla classe de problemas; por outro lado, o ALGOL 68 se destinava a uma única classe: as aplicações científicas.

A PL/I obteve uma aceitação bem maior do que o ALGOL 68, devido, em grande parte, aos esforços promocionais da IBM e aos problemas para entender e para implementar o ALGOL 68. A implementação foi um problema difícil para ambas, mas a PL/I tinha os recursos da IBM para aplicar na construção de um compilador. O ALGOL 68 não pôde contar com nenhum desses benefícios.

A linhagem do ALGOL 68 é mostrada na Figura 2.8.



**FIGURA 2.8** Genealogia do ALGOL 68.

## 2.12 Algumas Descendentes Importantes dos ALGOLs

Todas as linguagens imperativas, inclusive as orientadas a objeto, como, por exemplo, o C++ e o Java, projetadas a partir de 1960, devem alguma coisa de seus projetos ao ALGOL 60 e/ou ao ALGOL 68. Essa seção discute algumas dessas linguagens. Notavelmente em falta nesta seção estão a Ada, que será discutida na Seção 2.14, o C++, discutida na Seção 2.16, e o Java, discutida na Seção 2.17.

### 2.12.1 Simplicidade de Projeto: Pascal

#### 2.12.1.1 Embasamento Histórico

Niklaus Wirth (Pronuncia-se "Virt") era membro da *International Federation of Information Processing (IFIP) Working Group 2.1*, criado para continuar o desenvolvimento do ALGOL em meados da década de 60. Em agosto de 1965, Wirth e C.A.R. Hoare contribuíram para esse esforço, apresentando ao grupo uma proposta um tanto modesta para adições e para modificações no ALGOL 60 (Wirth e Hoare, 1966). A maioria do grupo rejeitou a proposta por ser um avanço pequeno demais sobre o ALGOL 60. Em vez disso, desenvolveu-se uma revisão proposta muito mais complexa, que se tornou, por fim, o ALGOL 68. Wirth, juntamente com alguns outros membros do grupo, não achava que o relatório ALGOL 68 deveria ter sido lançado, baseando-se na complexidade tanto da linguagem como da metalinguagem usada para descrevê-la. Tal posição demonstrou mais tarde ter alguma validade, porque os documentos daquele e, portanto, a própria linguagem, eram, de fato, consideradas muito difíceis de entender pela comunidade de computação.

A versão de Wirth e Hoare do ALGOL 60 foi chamada de ALGOL-W. Ela foi implementada na Universidade de Stanford e usada principalmente como um veículo de ensino, mas somente em algumas universidades. O método de passar parâmetros por valor e a instrução **case** para seleção múltipla estão entre as principais contribuições do ALGOL-W. Ele é uma alternativa ao método de passagem de parâmetros do ALGOL 60. Ambos serão discutidos no Capítulo 9. A instrução **case** será discutida no Capítulo 8.

O próximo grande esforço de projeto de Wirth, novamente baseado no ALGOL 60, foi o seu mais bem-sucedido: o Pascal. A definição original do Pascal foi publicada em 1971 (Wirth, 1971). Essa versão foi modificada no processo de implementação e ela é descrita em Wirth (1973). Os recursos que freqüentemente são atribuídos ao Pascal vieram de fato de linguagens anteriores. Por exemplo, os tipos de dados definidos pelo usuário foram introduzidos no ALGOL 68, a instrução **case** no ALGOL-W, e os registros do Pascal são semelhantes às variáveis estruturadas do COBOL e da PL/I.

#### 2.12.1.2 Avaliação

O maior impacto do Pascal incidiu sobre o ensino da programação. Em 1970, a maioria dos estudantes de ciência da computação, de engenharia e de ciências eram apresentados à programação com o FORTRAN, ainda que algumas universidades usassem a PL/I, linguagens baseadas nela e no ALGOL-W. Em meados da década de 70, o Pascal tornou-se a linguagem mais amplamente usada para tal finalidade. Isso era muito natural, mesmo que talvez não-previsível, porque o Pascal havia sido, de fato, projetado especificamente para ensinar programação. Só na década de 90 o Pascal deixou de ser a linguagem mais usada para o ensino da programação em colégios e universidades.

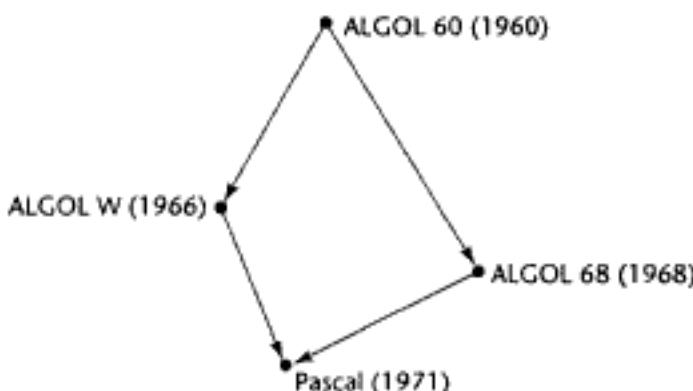
Faltam ao Pascal diversos recursos fundamentais para muitos tipos de aplicações, uma vez que ele foi projetado como uma linguagem de ensino. O melhor exemplo disso é a impossibilidade de escrever um subprograma que tome como parâmetro uma matriz de tamanho variável. Outro exemplo é a falta de qualquer capacidade de compilação separada. Essas deficiências, naturalmente, levaram a muitos dialetos não-padrão, como, por exemplo, o Turbo Pascal.

A popularidade do Pascal, tanto para ensinar programação como para outras aplicações, baseia-se principalmente em sua notável combinação de simplicidade e expressividade. Conforme discutiremos em capítulos posteriores, ele ainda é uma linguagem relativamente segura, especialmente quando comparada com o FORTRAN ou C, não obstante haver algumas inseguranças no Pascal. Em meados da década de 90, a sua popularidade estava em declínio, tanto na indústria como nas universidades.

Eis um exemplo de programa em Pascal:

```
(Exemplo de programa Pascal
Entrada: Um número inteiro, complis, em que
          complis é menor do que 100, seguido de complis
          valores inteiros
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;
  complis, contador, soma, media, resultado : integer;
begin
  resultado := 0;
  soma := 0;
  readln (complis);
  if ((complis > 0) and (complis < 100)) then
    begin
{ Leia a entrada em um vetor e compute a soma }
      for contador := 1 to complis do
        begin
          readln (intlist[contador]);
          soma := soma + intlist[contador]
        end;
{ Compute a média }
      media := soma / complis;
{ Conte o número de valores de entrada que são > do que a
  média }
      for contador := 1 to complis do
        if (intlist[contador] > media) then
          resultado := resultado + 1;
{ Imprima o resultado }
      writeln ('O número de valores > a média é:', resultado)
      end { da cláusula then do if ((complis > 0 ... )
      else
        writeln ('Erro - o tamanho da lista de entrada não é
          válido')
    end.
```

A linhagem do Pascal é mostrada na Figura 2.9.



**FIGURA 2.9** Genealogia do Pascal.

### 2.12.2 Uma Linguagem de Sistemas Portável: C

Como o Pascal, o C contribuiu pouco para o conjunto de recursos de linguagem que se conhecia anteriormente, mas tem sido muito usado. O C é adequado a uma ampla variedade de aplicações, não obstante haver sido originalmente projetado para programação de sistemas.

#### 2.12.2.1 Embasamento Histórico

Os antepassados do C incluem a CPL, a BCPL, o B e o ALGOL 68. A CPL foi desenvolvida na Universidade de Cambridge no início da década de 60. A BCPL é uma linguagem de sistemas simples desenvolvida por Martin Richards em 1967 (Richards, 1969).

O primeiro trabalho sobre o sistema operacional UNIX foi feito no final da década de 60 por Ken Thompson no Bell Laboratories. A primeira versão foi escrita em linguagem de montagem. A primeira linguagem de alto nível implementada sob o UNIX foi o B, que se baseou na BCPL. O B foi projetado e implementado por Thompson em 1970.

Nem a BCPL, nem o B possuem tipo, o que é estranho entre as linguagens de alto nível, ainda que ambas sejam de muito mais baixo nível do que o Java, por exemplo. Ser sem tipo significa que todos os dados são considerados palavras de máquina, o que é extremamente simples, mas leva a muitas complicações e a inseguranças. Por exemplo, há o problema de especificar a aritmética de números reais em vez da de inteiros em uma expressão. Em uma implementação da BCPL, os operandos de uma operação com números reais eram precedidos por pontos finais. Os operandos não-precedidos por pontos finais eram considerados números reais inteiros. Uma alternativa para isso teria sido usar símbolos diferentes para as operações com números reais.

Esse problema, juntamente com vários outros, levaram ao desenvolvimento de uma nova linguagem com tipos baseada no B. Originalmente chamada NB, mas posteriormente chamada C, ela foi projetada e implementada por Dennis Ritchie do Bell Laboratories em 1972 (Kernighan e Ritchie, 1978). Em alguns casos pela BCPL, e em outros casos diretamente, o C foi influenciado pelo ALGOL 68. Isso é visto em suas instruções **for** e **switch**, em seus operadores de atribuição e em seu tratamento dos ponteiros.

O único "padrão" para o C em sua primeira década e meia foi o livro escrito por Kernighan e Ritchie (1978). Durante esse intervalo de tempo, a linguagem lentamente evoluiu, com diferentes implementações, adicionando diferentes recursos. Em 1989, a ANSI produziu uma descrição oficial do C (ANSI, 1989) que incluiu muitos dos recursos que os implementadores já haviam incorporado na linguagem.

Uma nova versão do C, chamada C++, foi desenvolvida na metade e no final da década de 80 (Ellis e Stroustrup, 1997). Sua história e alguns de seus recursos mais significativos serão descritos na Seção 2.16. Os detalhes sobre o suporte e a abstração de dados do C++ são discutidos no Capítulo 11. No Capítulo 12 é discutido o suporte à programação orientada a objetos.

### 2.12.2.2 Avaliação

O C tem instruções de controle adequadas e facilidades de estruturação de dados para permitir seu uso em muitas áreas de aplicação. Também tem um rico conjunto de operadores que permitem um grau elevado de expressividade.

Uma das mais importantes razões pela qual o C é tanto apreciado como detestado encontra-se na sua completa ausência de verificação de tipos. Por exemplo, podem ser escritas funções para as quais os parâmetros não serão verificados quanto ao tipo. Aqueles que gostam do C apreciarão sua flexibilidade; aqueles que não gostam, irão achá-lo muito inseguro. Um importante motivo para o grande crescimento de popularidade obtido na década de 80 é que ele faz parte do amplamente usado sistema operacional UNIX. A inclusão no UNIX oferece um compilador muito barato (muitas vezes gratuito com o UNIX) e muito bom à disposição dos programadores em muitos tipos diferentes de computadores.

Eis um exemplo de programa em C:

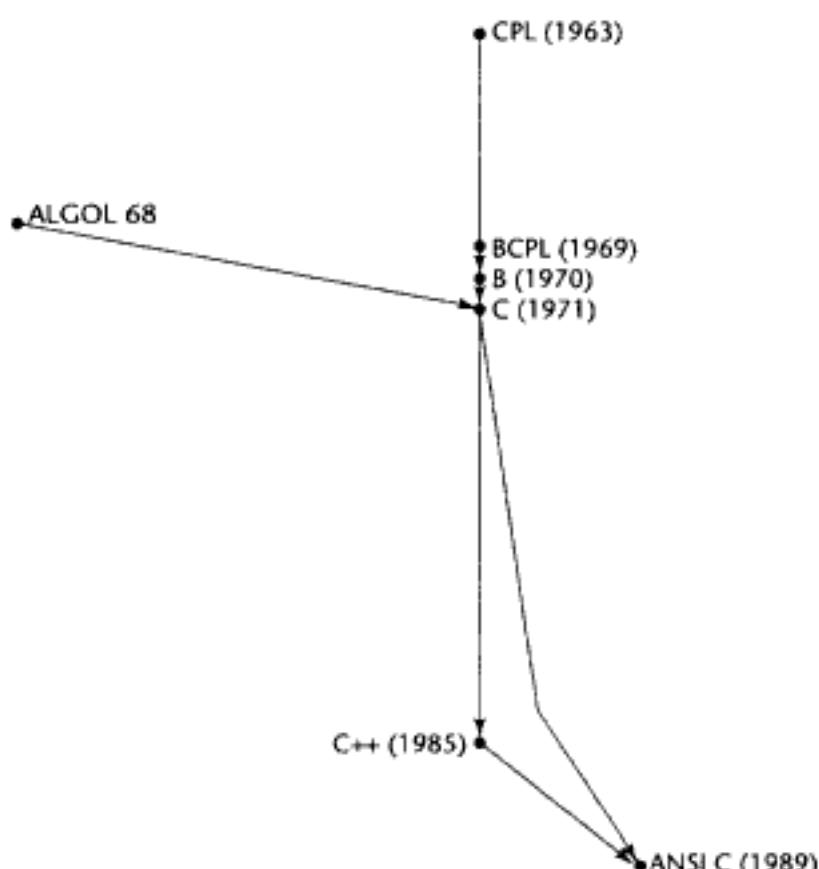
```
/* Exemplo de programa em C
Entrada: Um número inteiro, complis, em que complis
é menor do que 100, seguido de complis valores
inteiros
Saída: O número de valores de entrada que são maiores do
que a média de todos os valores de entrada */
void main () {
    int intlist[98], complis, contador, soma, media, resultado;
    soma = 0;
    resultado = 0;
    scanf("%d", &complis);
    if ((complis > 0) && (complis < 100)) {
        /* Leia a entrada em um vetor e compute a soma */
        for (contador = 0; contador < complis; contador++) {
            scanf("%d", &intlist[contador]);
            soma = soma + intlist[contador];
        }
        /* Calcule a média */
        media = soma / complis;
        /* Conte os valores de entrada que são > do que a média */
        for (contador = 0; contador < complis; contador++)
            if (intlist[contador] > media) resultado++;
        /* Imprima o resultado */
        printf("O número de valores > média é:%d\n", resultado);
    }
}
```

```

else
    printf("Erro - o tamanho da lista de entrada não é
           válido\n");
}

```

A linhagem do C é mostrada na Figura 2.10.



**FIGURA 2.10** Genealogia do C.

### 2.12.3 Outras Descendentes do ALGOL

Esta seção discutirá brevemente a origem e as características de algumas outras descendentes do ALGOL referenciadas mais tarde neste livro.

#### 2.12.3.1 Modula-2

Depois do Pascal, Niklaus Wirth projetou a Modula, que resultou de sua experimentação com a concorrência (Wirth, 1976). Nenhum compilador para a Modula foi lançado e o seu desenvolvimento descontinuou-se logo depois da sua publicação. Wirth, porém, não abandonou seu projeto. Seu foco mudou para a construção de uma linguagem que pretendia ser a única para o novo sistema de computador que, mais tarde, seria chamado Lilith. Embora o próprio computador jamais tenha sido um sucesso comercial, a linguagem Modula-2 o foi (Wirth, 1985).

Os principais recursos distintivos da Modula-2, cujo projeto baseava-se no Pascal e na Modula, eram os módulos, que fornecem suporte para tipos de dados abstratos, procedimentos como tipos, facilidades de baixo nível para programação de sistemas e para corotinas, além de alguns recursos sintáticos, considerados melhorias em relação ao Pascal.

A Modula-2 atingiu um uso generalizado no final da década de 80 e no início da década de 90 como linguagem de ensino em universidades. Ela foi usada, também, pelo menos durante alguns anos, em várias áreas de aplicação industrial.

### 2.12.3.2 Modula-3

A Modula-3 foi projetada conjuntamente pela *Systems Research Center of Digital Equipment Corporation*, de Palo Alto, e pelo *Olivetti Research Center*, de Menlo Park, no final da década de 80 (Cardelli et al., 1989). Ela se baseia na Modula-2, na Mesa (Mitchell et al., 1979), na Cedar (Lampson, 1983) e na Modula-2+ (Rovner, 1986). À Modula-2, ela adiciona classes e objetos para suporte à programação orientada a objeto, para manipulação de exceções, para coleta de lixo (*garbage collection*) e para suporte a concorrência. Os possíveis clientes da Modula-3 para uso generalizado são poucos, mesmo que ela seja uma linguagem bem-projetada e poderosa. Ainda que a Digital lhe dê suporte, a base de usuários da Modula-2 sobre a qual ela poderia ter crescido, desapareceu em sua maioria.

### 2.12.3.3 Oberon

A Oberon, que se baseia livremente na Modula-2, é a linguagem mais recente projetada por Niklaus Wirth. A última versão da Oberon chama-se Oberon-2 (Mössenbock, 1993).

A paixão de Wirth pela simplicidade nas linguagens de programação é evidente no projeto da Oberon. Não obstante alguns recursos terem sido adicionados à Modula-2 para obter a Oberon, um número maior de recursos foi subtraído, colocando esta última em claro contraste com o C++ e com a Ada 95. A Oberon é, em termos de complexidade e de tamanho, o oposto dessas duas linguagens.

O principal recurso adicionado à Modula-2 para obter a Oberon é a extensão de tipos, que suporta programação orientada a objeto. Entre os recursos removidos estão os registros variantes, tipos opacos, tipos enumerados, tipos subfaixa ( *subrange*), o tipo **CARDINAL**, índices de matriz não-inteiros, instruções **with** e **for**. É notável encontrar uma linguagem que seja significativamente menor e menos complexa do que sua antecessora, mas a Oberon é exatamente isso.

### 2.12.3.4 Delphi

O Delphi é híbrido, semelhante ao C++, uma vez que foi criado pela incorporação de suporte à programação orientada a objeto, dentre outras coisas, a uma linguagem imperativa existente. O Delphi é derivado do Pascal. Muitas das diferenças entre o C++ e o Delphi são um resultado das linguagens e das culturas de programação adjacentes a partir das quais eles derivam. Uma vez que o C é poderoso, mas potencialmente inseguro, o C++ também se enquadra nessa descrição, pelo menos nas áreas de verificação de faixa de subsírito de matriz, de aritmética de ponteiro e de suas numerosas coerções de tipo. Similarmente, porque o Pascal é mais elegante e seguro do que o C, o Delphi é mais elegante e seguro do que o C++. O Delphi também é menos complexo do que o C++. Por exemplo, ele não permite sobrecarga de operadores, subprogramas genéricos e classes parametrizadas, características do C++.

O Delphi, à semelhança do Visual C++, oferece uma interface gráfica com o usuário (GUI) que facilita o desenvolvimento das aplicações; proporciona também maneiras simples de incorporar interfaces GUI às aplicações. Ele foi projetado, suportado e vendido pela Borland,\* a mesma empresa que desenvolveu o Turbo Pascal.

## **2.13 Programação Baseada na Lógica: Prolog**

Introduzindo simplificadamente, programação lógica é o uso de uma notação lógica formal para comunicar processos computacionais a um computador. O cálculo de predicado é a notação usada nas atuais linguagens de programação lógicas.

A programação em linguagens de programação lógicas não é baseada em procedimentos. Os programas não declaram exatamente *como* um resultado deve ser computado, mas, em vez disso, descrevem a forma do resultado. O que é necessário para oferecer essa capacidade à linguagem é um meio conciso de abastecer o computador tanto com informações relevantes como com um processo de inferência para computar os resultados desejados. O cálculo de predicado fornece a forma básica de comunicação com o computador, e o método de prova chamado resolução, desenvolvido pela primeira vez por Robinson (1965), oferece a técnica da inferência.

### **2.13.1 Processo de Projeto**

Durante o início da década de 70, Alain Colmerauer e Phillippe Roussel, do *Artificial Intelligence Group*, da Universidade de Aix-Marseille, juntamente com Robert Kowalski, do *Department of Artificial Intelligence*, da Universidade de Edinburgo, desenvolveram o projeto fundamental do Prolog. Os componentes principais do Prolog são um método para especificar proposições de cálculo de predicado e uma implementação de uma forma restrita de resolução. Tanto o cálculo de predicado como a resolução serão descritos no Capítulo 16. O primeiro interpretador Prolog foi desenvolvido em Marseille, em 1972. A versão da linguagem implementada é descrita em Roussell (1975). O nome Prolog vem de *programming logic* (lógica de programação).

### **2.13.2 Visão Geral da Linguagem**

Os programas em Prolog são compostos de conjuntos de instruções. O Prolog tem somente alguns tipos de instruções, mas elas podem ser muito complexas.

Um uso comum do Prolog é como um tipo de banco de dados inteligente. Essa aplicação constitui uma estrutura simples para discussão da linguagem Prolog.

O banco de dados de um programa Prolog consiste em dois tipos de instruções: fatos e regras. São exemplos de instruções de fatos

```
mãe(joanne, jake).
pai(vern, joanne).
```

\*N. de R.T. A Borland é conhecida atualmente como Inprise, nome que utiliza desde a mudança de seu nome comercial em 29/04/1998.

as quais afirmam que joanne é a mãe de jake, e vern é o pai de joanne.

Um exemplo de instrução de regra é:

```
avô(X, Z) :- genitor(X, Y), genitor(Y, Z).
```

que declara poder deduzir-se que X é o avô de Z se for verdadeiro que X é o genitor<sup>\*</sup> de Y; e Y é o genitor de Z, para alguns valores das variáveis X, Y e Z.

O banco de dados Prolog pode ser consultado interativamente com instruções de metas, sendo um exemplo delas

```
pai (bob, darcie).
```

que pergunta se bob é o pai de darcie. Quando essa consulta, ou meta, é apresentada ao sistema Prolog, ele usa o processo de resolução para tentar determinar a verdade da instrução. Se ele puder concluir que a meta é verdadeira, exibirá "true". Se não puder prová-lo, exibirá "false".

### 2.13.3 Avaliação

Há um grupo relativamente pequeno de cientistas da computação acreditando que a programação lógica constitui a melhor esperança de escaparmos das linguagens imperativas, e também do enorme problema de produzirmos a grande quantidade de software confiável que é necessária atualmente. Até agora, porém, há duas importantes razões para que a programação lógica não tenha se tornado mais amplamente usada. Primeiro, como acontece com algumas outras abordagens não-imperativas, a programação lógica até agora tem se demonstrado altamente ineficiente. Em segundo lugar, ela tem-se mostrado um método eficiente somente para algumas áreas relativamente pequenas de aplicação: certos tipos de sistemas de gerenciamento de bancos de dados e algumas áreas de IA.

A programação lógica e o Prolog serão descritos com mais detalhes no Capítulo 16.

## 2.14 O Maior Esforço de Projeto da História: Ada

A linguagem Ada é o resultado do mais extensivo e dispendioso esforço de projeto de uma linguagem já realizado. Ela foi desenvolvida para o Departamento de Defesa (DoD), de modo que a situação de seu ambiente de computação foi fundamental para determinar sua forma.

### 2.14.1 Embasamento Histórico

Em 1974, mais da metade das aplicações de computadores no DoD era composta de sistemas incorporados (que são aqueles cujo hardware é incorporado no dispositivo que ele controla ou para o qual fornece serviços). Os custos de software elevavam-se rapidamente; primeiro, por causa da crescente complexidade dos sistemas. Mais de 450 diferentes lin-

<sup>\*</sup>N. de R.T. Genitor, aqui, seria uma regra que resultaria em verdadeiro quando fosse entrado o pai ou a mãe. Desta forma, a regra avô encontraria tanto os avós paternos como os avós maternos.

guagens de programação estavam em uso para projetos do DoD, e nenhuma delas era padronizada pelo DoD. Todo contratante do Departamento podia definir uma linguagem nova e diferente para cada contrato. Por causa dessa proliferação de linguagens, os softwares de aplicação raramente eram reusados. Além disso, nenhuma ferramenta de desenvolvimento de software era criada (porque, usualmente, elas eram dependentes da linguagem). Um número enorme de linguagens estava em uso, mas nenhuma era, de fato, adequada para aplicações de sistemas incorporados. Por essas razões, o Exército, a Marinha e a Força Aérea propuseram, cada um independentemente, o desenvolvimento de uma linguagem de alto nível para sistemas incorporados.

### 2.14.2 Processo de Projeto

Notando esse interesse generalizado, Malcolm Currie, Diretor de Pesquisas e Engenharia de Defesa, em janeiro de 1975, formou o *High-Order Language Working Group* (HOLWG), inicialmente chefiado pelo Tenente-coronel William Whitaker, da Força Aérea. O HOLWG tinha representantes de todos os serviços militares e contatos com a Inglaterra, com a França e com a Alemanha Ocidental. A incumbência deles era:

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

Em abril de 1975, o HOLWG produziu o documento de requisitos *Strawman* (Homem de Palha) para a nova linguagem (Departamento de Defesa, 1975a), distribuído às unidades militares, às agências federais, aos representantes industriais e universitários selecionados e a partes interessadas na Europa.

Ao documento *Strawman* seguiu-se *Woodenman* (Homem de Madeira) (Departamento de Defesa, 1975b) em agosto de 1975 e pelo *Tinman* (Homem de Lata) (Departamento de Defesa, 1976) em janeiro de 1976. O documento *Tinman* foi considerado um conjunto de requisitos completo para uma linguagem com as características desejadas. O autor principal desses documentos foi David Fisher, do Instituto para Análise de Defesa. O grupo de participantes no esforço foi grande, chegando a mais de 200, com representantes de mais de 40 organizações fora do DoD. Em janeiro de 1977, o documento *Tinman* foi substituído pelo documento de requisitos *Ironman* (Homem de Ferro) (Departamento de Defesa, 1977), que era quase equivalente, em termos de conteúdo, mas tinha um formato bastante diferente.

Em abril de 1977, o documento *Ironman* foi usado como base para um pedido irrestrito de propostas, o qual, depois, tornou-se público, transformando, então, a Ada na primeira linguagem a ser projetada pelo contrato competitivo. Em julho de 1977, quatro das contratantes proponentes — a Softech, a SRI International, a Cii Honeywell/Bull e a Intermetrics — foram escolhidas para produzir, independentemente e em paralelo, a Fase I do projeto da linguagem. Todas as quatro propostas de projeto resultantes basearam-se no Pascal.

Quando a Fase I de seis meses encerrou-se em fevereiro de 1978, houve uma avaliação de dois meses por 400 voluntários em 80 equipes de revisão espalhadas pelo mundo todo. Duas finalistas acabaram escolhidas para continuar a Fase 2 do desenvolvimento — a Intermetrics e a Cii Honeywell/Bull.

Em junho de 1978, a iteração seguinte do documento de requisitos, lançou-se *Steelman* (Homem de Aço) (Departamento de Defesa, 1978).

Ao final da Fase 2, organizou-se outra avaliação de dois meses, e, em maio de 1979, o projeto de linguagem Cii Honeywell foi escolhido como vencedor. Curiosamente, a vencedora era a única concorrente estrangeira entre as quatro finalistas. A equipe de projeto da Cii Honeywell/Bull, da França, era chefiada por Jean Ichbiah.

Na primavera de 1979, Jack Cooper, do Controle de Materiais da Marinha (Navy Materiel Command) recomendou o nome para a nova linguagem, Ada, adotado desde então. Augusta Ada Byron (1815-1851), a Condessa de Lovelace, matemática e filha do poeta Lord Byron, geralmente é reconhecida como sendo a primeira programadora do mundo. Ela trabalhou com Charles Babbage em seus computadores mecânicos, os *Difference and Analytical Engines*, escrevendo programas para diversos processos numéricos.

A Fase 3 do projeto da Ada iniciou-se com a escolha do vencedor. O projeto e o fundamento lógico para ele foram publicados pela ACM em seu *SIGPLAN Notices* (ACM, 1979) e distribuídos para um universo de leitores de mais de 10.000 pessoas. Uma conferência de teste e de avaliação públicos realizou-se em outubro de 1979 em Boston, com representantes de mais de 100 organizações dos Estados Unidos e da Europa. Em novembro, recebeu-se mais de 500 relatórios sobre a linguagem oriundos de 15 diferentes países. A maioria dos relatórios sugeriu pequenas modificações em vez de mudanças drásticas e de rejeições definitivas. Baseando-se nos relatórios sobre a linguagem, a versão seguinte da especificação dos requisitos, o documento *Stoneman* (Homem de Pedra) (Departamento de Defesa, 1980a), foi lançado em fevereiro de 1980.

Uma versão revisada do projeto da linguagem concluiu-se em julho de 1980 e foi aceita como MIL-STD 1815, o *Ada Language Reference Manual* padrão. O número 1815 foi escolhido porque trata-se do ano de nascimento de Augusta Ada Lovelace. Outra versão revisada do *Ada Language Reference Manual* foi lançada em julho de 1982. Em 1983, o American National Standards Institute padronizou a Ada. Essa versão oficial "final" é descrita em Goos e Hartmanis (1983). O projeto da linguagem Ada permaneceu, então, congelado durante pelo menos cinco anos.

### 2.14.3 Visão Geral da Linguagem

Esta seção descreve brevemente quatro dos principais recursos da linguagem Ada. Outros recursos serão descritos ao longo do caminho, uma vez que usamos a linguagem em muitos exemplos no restante deste livro.

Os pacotes na linguagem Ada oferecem os meios para encapsular objetos de dados, especificações para tipos de dados e procedimentos. Isso, por sua vez, fornece o suporte para o uso da abstração de dados no projeto de programas, conforme descrevemos no Capítulo 11.

A linguagem Ada inclui extensas facilidades para a manipulação de exceções que permitem aos programadores ganharem controle depois que qualquer uma de uma ampla variedade de exceções ou de erros em tempo de execução forem detectados. A manipulação de exceções será discutida no Capítulo 14.

As unidades de programa podem ser genéricas em Ada. Por exemplo, é possível escrever um procedimento de ordenação que usa um tipo não-especificado para os dados que serão ordenados. Esse procedimento genérico deve ser instanciado para um tipo específico antes que possa ser usado. Isso é feito com uma instrução que leva o compilador a gerar uma versão do procedimento com o tipo dado. A disponibilidade dessas unidades genéricas aumenta a variedade de unidades de programa que podem ser reusadas, em vez de duplicadas, pelos programadores. As genéricas serão discutidas nos Capítulos 9 e 11.

A linguagem Ada também se presta à execução concorrente de unidades de programa especiais, chamadas tarefas, usando o mecanismo de *rendezvous*. *Rendezvous* é o nome de um método de comunicação entre tarefas e sincronização. A concorrência será discutida no Capítulo 13.

#### 2.14.4 Avaliação

Talvez os mais importantes aspectos de projeto da linguagem Ada a considerarmos sejam os seguintes:

- Não houve nenhum limite à participação, uma vez que o projeto foi competitivo.
- A linguagem Ada incorpora a maioria dos conceitos de engenharia de software e de projeto de linguagens do final da década de 70. Mesmo que seja possível questionar os métodos reais usados para incluir tais recursos, bem como a conveniência de incluir um número tão grande de recursos em uma linguagem, a maioria concorda que os recursos são valiosos.
- Ainda que muitas pessoas não o tenham reconhecido inicialmente, o desenvolvimento de um compilador para a linguagem Ada era uma tarefa difícil. Somente em 1985, quase quatro anos depois que o projeto da linguagem foi concluído, que compiladores Ada verdadeiramente úteis começaram a aparecer.

A crítica mais séria à Ada em seus primeiros anos recaiu sobre seu tamanho grande e sobre sua complexidade. Em particular, Hoare declarou que ela não deve ser usada em qualquer aplicação cuja confiabilidade seja crítica (Hoare, 1981), que é precisamente o tipo de aplicação para a qual ela foi projetada. Por outro lado, outros louvaram-na como o epítome de projeto de uma linguagem.

Eis um exemplo de programa em Ada:

```
-- Exemplo de Programa em Ada
-- Entrada: Um número inteiro, COMPLIS, em que
--           COMPLIS é menor do que 100, seguido de COMPLIS
--           valores inteiros
-- Saída:   O número de valores de entrada que são maiores do
--           que a média de todos os valores de entrada
with TEXT_IO; use TEXT_IO;
procedure ADA_EX is
    package INT_IO is new INTEGER_IO (INTEGER);
    use INT_IO;
    type INT_LIST_TYPE is array (1..99) of INTEGER;
    INT_LIST : INT_LIST_TYPE;
    COMPLIS, SOMA, MEDIA, RESULTADO : INTEGER;
    begin
        RESULTADO := 0;
        SOMA := 0;
        GET (COMPLIS);
        if (COMPLIS > 0) and (COMPLIS < 100) then
            -- Leia os dados de entrada em um vetor e compute a soma
            for CONTADOR := 1 .. COMPLIS loop
                GET (INT_LIST(CONTADOR));
                SOMA := SOMA + INT_LIST(CONTADOR);
```

```

    end loop;
-- Calcule a média
    MEDIA := SOMA / COMPLIS;
-- Conte o número de valores que são > do que a média
    for CONTADOR := 1 .. COMPLIS loop
        if INT_LIST(CONTADOR) > MEDIA then
            RESULTADO := RESULT + 1;
        end if;
    end loop;
-- Imprima o resultado
    PUT ("O número de valores > do que a média é:");
    PUT (RESULTADO);
    NEW_LINE;
else
    PUT_LINE ("Erro - o tamanho da lista de entrada não
              é válido");
end if;
end ADA_EX;

```

## 2.14.5 Ada 95

Um esforço de revisão da Ada iniciou-se em 1988 com o estabelecimento do projeto Ada 9X pelo *Ada Joint Program Office*. O projeto Ada 9X teve três fases: a determinação dos requisitos para a linguagem revista; o desenvolvimento real da definição da linguagem revisada e a transição para o uso da linguagem revisada. Os requisitos foram publicados no Departamento de Defesa (1990).

Os requisitos concentraram-se em quatro áreas da linguagem: capacidades de interface (especialmente para interfaces gráficas), suporte para programação orientada a objeto, bibliotecas mais flexíveis e melhores mecanismos de controle para dados compartilhados.

Dois dos mais importantes novos recursos da nova versão da Ada, chamada Ada 95, serão descritos brevemente nos capítulos seguintes. No restante do livro, usaremos a chamada Ada 83 para a versão original e Ada 95 (seu nome real) para a versão posterior quando for importante distinguir entre as duas versões. Em discussões sobre os recursos de linguagem comuns a ambas as versões, usaremos o nome Ada. A linguagem padrão Ada 95 é definida em AARM (1995).

O tipo de mecanismo de derivação da Ada 83 é estendido para permitir a adição de novos componentes àqueles derivados de um tipo-pai marcado. Isso resulta na herança, um ingrediente-chave nas linguagens de programação orientadas a objeto. A vinculação dinâmica de chamadas de subprogramas a definições de subprogramas é realizada através do despacho de subprograma, que se baseia no valor da marca (*tag*) de tipos derivados por meio de tipos da classe ampla. Esse recurso proporciona o polimorfismo, outro recurso principal da programação orientada a objeto. Tais recursos da Ada serão discutidos no Capítulo 12.

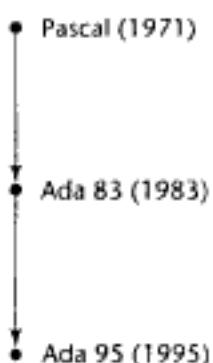
O mecanismo de *rendezvous* da Ada 83 oferecia somente meios desajeitados de compartilhar dados entre processos concorrentes. Era necessário introduzir uma nova tarefa

<sup>1</sup>N. de T. Tag: um conjunto de bits ou de caracteres que identificam várias condições sobre dados em um campo e é freqüentemente usado nos registros de cabeçalho de arquivos que possuam esse mecanismo.

para controlar o acesso a dados compartilhados. Os objetos protegidos da Ada 95 oferecem uma alternativa atraente para isso. Os dados compartilhados são encapsulados em uma estrutura sintática que controla todo o acesso aos dados, por *rendezvous*, ou por chamada de subprograma. Os novos recursos da Ada 95 para concorrência e dados compartilhados serão discutidos no Capítulo 13.

Acredita-se amplamente que a popularidade da Ada 95 caia, porque o Departamento de Defesa não mais exige seu uso em sistemas militares. Obviamente, há outros fatores que dificultam o crescimento de sua popularidade.

A linhagem da Ada é mostrada na Figura 2.11.



**FIGURA 2.11** Genealogia da Ada.

## 2.15 Programação Orientada a Objeto: Smalltalk

Conforme discutimos no Capítulo 1, a programação orientada a objeto inclui a abstração de dados como uma de suas três características fundamentais, sendo as outras duas a herança e a vinculação dinâmica.

A herança apareceu sob uma forma limitada na SIMULA-67, cujas classes podem ser definidas em hierarquias. A herança constitui um método efetivo de reutilização de código.

O conceito de controle unitário da programação orientada a objeto é mais ou menos modelado na idéia de que os programas simulam o mundo real, um legado de suas origens na SIMULA 67. A simulação desse mundo deve incluir objetos simulados uma vez que grande parte do mundo real é repleta de objetos. De fato, uma linguagem baseada nos conceitos de simulação do mundo real somente precisa incluir um modelo de objetos que possa enviar e receber mensagens e reagir às mensagens que recebe.

A essência da programação orientada a objeto é resolver problemas, identificando os objetos do mundo real do problema e o processamento exigido desses objetos, e depois criar simulações desses objetos, de seus processos e da comunicação requerida entre eles. Tipos de dados abstratos, vinculação dinâmica e herança são os conceitos que tornam a solução de problemas orientados a objeto não somente possível, mas também conveniente e efetiva.

### 2.15.1 Processo de Projeto

Os conceitos que levaram ao desenvolvimento do Smalltalk originaram-se no trabalho de dissertação de Ph.D. de Alan Kay, no final da década de 60, na Universidade de Utah (Kay,

1969). Kay teve uma notável previsão ao antever a disponibilidade futura de computadores de mesa poderosos. Lembre-se que os primeiros microcomputadores somente foram comercializados em meados da década de 70, e eles estavam apenas longinquoamente relacionados com as máquinas pressentidas por Kay, que eram vistas para executar um milhão ou mais de instruções por segundo e conter vários megabytes de memória. Essas máquinas, na forma de estações de trabalho, tornaram-se amplamente disponíveis somente no início da década de 80.

Kay acreditava que os computadores de mesa seriam usados por não-programadores e, dessa forma, precisariam de capacidades de interface humana muito poderosas. Os computadores do final da década de 60 eram amplamente orientados a instruções em lote e eram usados exclusivamente por programadores profissionais e por cientistas. Para ser usado por não-programadores, determinou Kay, um computador teria de ser altamente interativo e usar gráficos sofisticados em sua interface com os usuários. Alguns dos conceitos gráficos surgiram da experiência LOGO de Seymour Papert, em que gráficos eram usados para ajudar crianças a usar computadores (Papert, 1980).

Kay originalmente imaginou um sistema que chamou de Dynabook, e que pretendia ser um processador de informações gerais. Baseava-se, em parte, na linguagem *Flex*, que ele havia ajudado a projetar e a *Flex* baseava-se principalmente na *SIMULA 67*. O Dynabook baseava-se no paradigma da escrivaninha típica, na qual havia um grande número de papéis, alguns parcialmente cobertos. A folha de cima, muitas vezes, é o foco de atenção, com as outras fora de foco temporariamente. O monitor do Dynabook teria como modelo esse cenário, usando janelas de tela. O usuário interagiria com esse monitor tanto por um teclado como tocando a tela com seus dedos. Depois que o projeto preliminar do Dynabook fez com que ele ganhasse um Ph.D., a meta de Kay voltou-se para a construção dessa máquina.

Kay encontrou seu caminho para o *Xerox Palo Alto Research Center* (Xerox PARC) e apresentou suas idéias sobre o Dynabook. Isso o conduziu ao seu emprego lá e ao subsequente nascimento do *Learning Research Group* na Xerox. O primeiro encargo do grupo foi projetar uma linguagem para suportar o paradigma de programação de Kay e implementá-la no melhor computador pessoal então disponível. Esses esforços resultaram em um Dynabook "Interim" (intermediário), que consistia no hardware Xerox Alto e no software Smalltalk-72. Juntos, eles formaram uma ferramenta de pesquisa para desenvolvimento adicional. Levou-se a efeito uma série de projetos de pesquisa com esse sistema, incluindo diversos experimentos para ensinar programação a crianças. Juntamente com os sistemas vieram desenvolvimentos adicionais, levando a uma seqüência de linguagens que culminaram no Smalltalk-80, a versão discutida neste livro. À medida que a linguagem cresceu, o mesmo aconteceu com a potência do hardware na qual ela residia. Em 1980, tanto a linguagem como o hardware da Xerox quase coincidiam com a primeira visão de Alan Kay.

## 2.15.2 Visão Geral da Linguagem

As unidades de programa do Smalltalk são objetos, estruturas que encapsulam dados locais e um conjunto de operações chamadas de métodos e que ficam disponíveis a outros objetos. Um método especifica a reação do objeto quando ele recebe uma mensagem particular que corresponde àquele método. O mundo do Smalltalk não é preenchido por outra coisa a não ser por objetos, de constantes inteiras a grandes sistemas de software complexos.

Toda computação no Smalltalk é feita por meio da mesma técnica uniforme: envia-se uma mensagem a um objeto para invocar um de seus métodos. Uma resposta a uma mensagem é um objeto, que retorna a informação solicitada ou simplesmente notifica o remetente

que o processamento solicitado está concluído. A diferença fundamental entre uma mensagem e uma chamada de subprograma é esta: uma mensagem é enviada a um objeto de dados, o qual, então, é processado pelo código associado ao objeto: uma chamada de subprograma normalmente envia os dados a serem processados a uma unidade de código de subprograma.

Do ponto de vista de simulação, que nunca está distante, o Smalltalk é uma simulação de uma coleção de computadores (objetos) que se comunicam entre si (por intermédio de mensagens). Cada objeto é uma abstração de um computador em termos de que ele armazena dados e oferece capacidades de processamento para manipular estes últimos. Além disso, objetos podem enviar e receber mensagens. Essencialmente, essas são as capacidades fundamentais dos computadores: armazenar e manipular dados e comunicar-se.

No Smalltalk, abstrações de objetos são classes, muito semelhantes às da SIMULA 67. Instâncias das classes podem ser criadas e serão, então, os objetos do programa. Cada objeto tem seus próprios dados locais e representa uma instância diferente de sua classe. A única diferença entre dois objetos da mesma classe é o estado de suas variáveis locais.

Como acontece na SIMULA 67, hierarquias de classes podem ser formadas no Smalltalk. Subclasses de determinada classe são refinamentos dela, herdando a funcionalidade e as variáveis locais da classe principal ou superclasse. Subclasses podem adicionar nova memória local e funcionalidade e modificar ou ocultar a funcionalidade herdada.

Como foi brevemente discutido no Capítulo 1, o Smalltalk não é apenas uma linguagem; é também um ambiente completo de desenvolvimento de software. A interface com o ambiente é altamente gráfica, fazendo muito uso de janelas sobrepostas e menus suspensos (pop-up) e de um dispositivo de entrada (mouse).

### 2.15.3 Avaliação

O Smalltalk colaborou muito para promover dois aspectos distintos da computação. Os sistemas de janelas que, agora, são o método predominante de interface com o usuário para sistemas de software se desenvolveram a partir do Smalltalk. Hoje, as metodologias de projeto de software e de linguagens de programação mais significativas são orientadas a objeto. Elas atingiram a maturidade somente no Smalltalk, embora algumas das idéias das linguagens orientadas a objeto tenham vindo da SIMULA 67. É claro que o impacto do Smalltalk sobre o mundo da computação é extenso e terá longa duração.

Eis um exemplo de definição de classe Smalltalk:

```

"Exemplo de Programa Smalltalk"
"A seguinte é uma definição de classe, cujas instanciações
podem desenhar polígonos equiláteros com qualquer número de
lados"
class name           Poligono
superclass          Object
instance variable names
nossaCaneta
numLados
comprimentoLado

"Métodos de classe"
"Crie uma instância"
new
    ^ super new getPen"

```

```

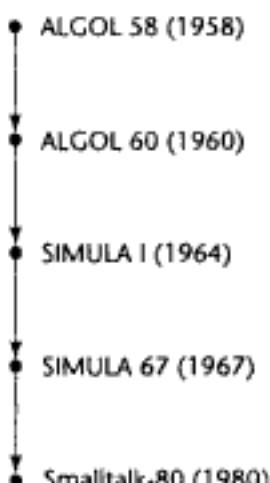
    "Pegue uma caneta para desenhar polígonos"
    getPen
        nossaCaneta <- Pen new defaultNib: 2
    "Métodos de instâncias"
    "Desenhe um polígono"
    draw
        numLados timesRepeat: [nossaCaneta go: comprimentoLado;
                                turn: 360 // numLados]

    "Defina o tamanho dos lados"
    comprimento: len
        comprimentoLado <- len

    "Defina o número de lados"
    lados: num
        numLados <- num

```

A linhagem do Smalltalk é mostrada na Figura 2.12.



**FIGURA 2.12** Genealogia do Smalltalk.

## 2.16 Combinando Recursos Imperativos e Orientados a Objeto: C++

As origens do C foram discutidas na Seção 2.12; as origens do Smalltalk foram discutidas na Seção 2.15. O C++ constrói facilidades de linguagem sobre o C para suportar grande parte daquilo em que o Smalltalk foi o pioneiro. O C++ evoluiu a partir do C e passou por uma seqüência de modificações para melhorar seus recursos imperativos e de adições para suportar a programação orientada a objeto.

### 2.16.1 Processo de Projeto

O primeiro passo do C rumo ao C++ foi dado por Bjarne Stroustrup, do Bell Laboratories, em 1980. As modificações incluíram a adição da verificação de tipos de parâmetro de fun-

ção e conversão e, o que é mais significativo, as classes, as quais se relacionam com as da SIMULA 67 e com as do Smalltalk. Também foram incluídas as classes derivadas, o controle de acesso público/privado de componentes herdados, as funções construtoras e destruidoras e as classes amigas (*friend*). Durante 1981, funções *inline*, parâmetros assumidos e sobrecarga do operador de atribuição foram adicionados. A linguagem resultante foi chamada de C with Classes e é descrita em Stroustrup (1983).

É útil considerarmos algumas metas do C with Classes, cuja principal era oferecer uma linguagem em que os programas pudessem ser organizados como eram na SIMULA 67; ou seja, com classes e herança. Outra meta importante era que não deveria haver nenhuma penalidade ao desempenho, em relação ao C. Assim, o C with Classes poderia ser usado por todas as aplicações para as quais o C poderia ser usado, mas virtualmente nenhum dos recursos inseguros do C seriam removidos. Por exemplo, a verificação da faixa de índice de matriz não era nem mesmo considerada, porque resultaria em uma significativa desvantagem de desempenho em relação ao C.

Em 1984, ampliou-se essa linguagem com a inclusão das funções virtuais, que oferecem vinculação dinâmica de chamadas de função para definições específicas de função, sobrecarga de nomes de função e de operadores, além de tipos de referência. Essa versão da linguagem foi chamada de C++. Ela é descrita em Stroustrup (1984).

Em 1985, surgiu a primeira implementação disponível, um sistema chamado Cfront, que traduz programas em C++ para programas em C. Essa versão da Cfront e a versão do C++ que ela implementou foram chamadas Release 1.0. Ela é descrita em Stroustrup (1986).

Entre 1985 e 1989, o C++ continuou a evoluir, baseando-se, em grande parte, na reação dos usuários à primeira implementação distribuída. Essa versão seguinte foi chamada de 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, a herança múltipla (classes com mais de uma classe principal) e classes abstratas, juntamente com algumas outras inclusões. As classes abstratas são descritas no Capítulo 12.

A Release 3.0 do C++ evoluiu entre 1989 e 1990. Ela adicionou modelos, que oferecem tipos parametrizados e manipulação de exceções. A versão corrente do C++ é descrita em Stroustrup (1997).

## 2.16.2 Visão Geral da Linguagem

A parte mais fundamental do suporte para programação orientada a objeto é o mecanismo de classe/objeto, que se originou na SIMULA 67 e é o recurso central do Smalltalk. O C++ oferece uma coleção de classes predefinidas, juntamente com a possibilidade de classes definidas pelo usuário. As classes do C++ são tipos de dados que, à semelhança das classes Smalltalk, podem ser instanciados qualquer número de vezes. Essas instanciações no C++ são simplesmente declarações de objetos, ou de dados. As definições de classes especificam objetos de dados (chamados de membros de dados) e de funções (chamadas de funções-membro). As classes podem especificar uma ou mais classes primárias, fornecendo herança e herança múltipla, respectivamente. As classes herdam os dados e as funções que fazem parte da classe pai especificada.

Os operadores em C++ podem ser sobrecarregados; isso quer dizer que o usuário pode criar operadores para outros já existentes em tipos definidos por ele mesmo. As funções em C++ também podem ser sobrecarregadas, significando que o usuário pode definir mais de uma função com o mesmo nome, contanto que ou o número de parâmetros ou seus tipos sejam diferentes.

A vinculação dinâmica no C++ é fornecida pelas funções de classe virtuais que definem operações dependentes do tipo, usando funções sobrecarregadas, dentro de uma coleção de classes relacionadas pela herança. Um ponteiro para um objeto da classe A também pode apontar para objetos que herdam a classe A. Quando esse ponteiro aponta para uma função virtual sobrecarregada, a função do tipo atual é escolhida dinamicamente.

Tanto as funções como as classes podem ser modeladas, o que significa que elas podem ser parametrizadas. Por exemplo, uma função pode ser escrita de maneira modelada para permitir que ela tenha versões para uma variedade de tipos de parâmetro. As classes desfrutam da mesma flexibilidade.

O C++ inclui uma manipulação de exceções de maneira diferente da Ada. Uma diferença é que as exceções detectadas por hardware não podem ser manipuladas. A manipulação de exceções do C++ será discutida no Capítulo 14.

### 2.16.3 Avaliação

O C++ tornou-se rapidamente uma linguagem muito popular e continua a sê-lo. Um fator dessa popularidade é a disponibilidade de compiladores bons e baratos. Outro fator em favor da popularidade do C++ é que ele é quase completamente compatível em ordem descendente com o C (isso quer dizer que a maioria dos programas em C pode, pelo menos em sua maior parte, ser compilada como programas C++), e é possível vincular código C++ com código C na maioria das implementações. Finalmente, os programadores agora estão interessados na programação orientada a objeto. O C++ é o veículo pelo qual muitos programadores usam a metodologia orientada a objeto.

Quanto ao lado negativo, uma vez que o C++ é uma linguagem muito grande e complexa, evidentemente padece de insuficiências similares às da PL/I. Herdou a maioria das inseguranças do C, o que o torna menos confiável do que a Ada e o Java. Finalmente, em parte por causa de sua base no C, o C++ é mais assemelhado com a PL/I do que com o ALGOL 68 em termos de ser mais uma coleção de idéias reunidas do que o resultado de um plano de projeto global de uma linguagem.

Os recursos orientados a objeto do C++ serão descritos mais detalhadamente no Capítulo 12.

A linhagem do C++ é mostrada na Figura 2.10 (ver p. 88).

### 2.16.4 Uma Linguagem Relacionada: Eiffel

A Eiffel é outra linguagem híbrida que contém tanto recursos imperativos como orientados a objeto (Meyer, 1992). Ela foi projetada por uma única pessoa, Bertrand Meyer, um francês que vive na Califórnia. A linguagem inclui recursos que suportam tipos de dados abstratos, herança e vinculação dinâmica, de modo que suporta amplamente a programação orientada a objeto. Talvez o recurso mais distintivo da Eiffel seja o uso integrado de assinaturas para reforçar o “contrato” entre subprogramas e seus chamadores. É uma idéia que nasceu na Plankalkül, mas foi ignorada pela maioria das outras linguagens projetadas desde então. É natural comparar a Eiffel com o C++. A primeira é menor e mais simples do que a outra, mas tem uma expressividade e uma capacidade de escrita quase igual. As razões para a crescente popularidade do C++, enquanto a Eiffel ainda tem um uso limitado, não são difíceis de determinar. O C++ é, evidentemente, a maneira mais fácil para as organizações de desenvolvimento de software mudarem para a programação orientada a

objeto, porque, em muitos casos, seus desenvolvedores já conhecem o C. A Eiffel não desfruta desse caminho fácil para a adoção. Além disso, nos primeiros anos em que o C++ se difundiu, o sistema Cfront estava à disposição e era barato, enquanto que os compiladores Eiffel não estavam tão disponíveis e eram mais caros. O C++ tinha o apoio do prestigioso Bell Laboratories, ao passo que a Eiffel era financiada por Bertrand Meyer e pela sua relativamente pequena empresa de software, a Interactive Software Engineering.

## 2.17 Programando a World Wide Web: Java

Os projetistas do Java iniciaram com o C++, removendo numerosas construções, mudaram algumas e adicionaram outras. A linguagem resultante oferece grande parte do poder e da flexibilidade do C++, mas em uma linguagem menor, mais simples e mais segura.

### 2.17.1 Processo de Projeto

O Java, como muitas linguagens de programação, foi projetado para uma aplicação para a qual não parecia haver uma linguagem satisfatória. No caso do Java, entretanto, era de fato uma sequência de aplicações, sendo a primeira a programação de dispositivos eletrônicos incorporados, como, por exemplo, torradeiras, fornos de microondas e sistemas de TV interativos. Pode não parecer que a confiabilidade seja um fator importante em um software para forno de microondas. Se um equipamento desses tivesse um software com mau funcionamento, provavelmente isso não constituiria um perigo grave a qualquer pessoa e provavelmente não acarretaria grandes questões legais. Porém, se o software de um modelo particular fosse considerado errôneo depois que milhares de unidades tivessem sido fabricadas e vendidas, a devolução acarretaria um significativo custo. Portanto, a confiabilidade é uma característica importante do software nos produtos eletrônicos de consumo.

Em 1990, a Sun Microsystems concluiu que nenhuma das duas linguagens de programação que estavam considerando, o C e o C++, eram satisfatórias para desenvolver software de dispositivos eletrônicos de consumo. Mesmo o C sendo relativamente pequeno, não oferecia suporte para programação orientada a objeto, o que eles consideravam uma necessidade. O C++ suportava tal programação, mas seu tamanho e sua complexidade eram vistos como perigos significativos. Acreditava-se também que nem o C, nem o C++ ofereciam o necessário nível de confiabilidade. O projeto do Java foi orientado pelo conceito fundamental de oferecer maior simplicidade e confiabilidade do que acreditavam ser oferecido pelo C++.

Ainda que o impulso inicial do Java tenha sido para os produtos eletrônicos de consumo, nenhum dos produtos com os quais ele foi usado em seus primeiros anos chegou a ser comercializado. Quando a World Wide Web tornou-se amplamente usada, a partir de 1993, em grande parte por causa dos novos navegadores gráficos, o Java passou a ser considerado uma ferramenta útil para a programação da Web. Em seus primeiros anos de uso público, a Web tem sido a aplicação mais comum do Java.

A equipe de projeto do Java foi chefiada por James Gosling, que já havia projetado o editor UNIX *emacs* e o sistema de janelas NeWS.

## 2.17.2 Visão Geral da Linguagem

Conforme afirmamos antes, o Java baseou-se no C++, mas foi especificamente projetado para ser menor, mais simples e mais confiável. O Java tem tanto tipos como classes. Tipos primitivos não são objetos baseados em classes. Esses incluem todos os tipos escalares, inclusive aqueles para dados inteiros, reais, booleanos e de caracteres. Os objetos são acessados por variáveis de referência, mas os valores de tipo primitivos são acessados exatamente como os valores escalares em linguagens puramente imperativas como o C e a Ada 83. As matrizes Java são instâncias de uma classe predefinida, enquanto que no C++ elas não são, apesar de muitos usuários desta última construírem classes-invólucros para matrizes com a finalidade de adicionar recursos como a verificação da faixa de índice, que é implícita no Java.

O Java não tem ponteiros, mas seus tipos de referência oferecem algumas das capacidades dos ponteiros. Essas referências são usadas para apontar a instâncias de classes — de fato, essa é a única maneira pela qual as instâncias podem ser referenciadas. Todos os objetos são alocados no monte. Embora os ponteiros e as referências possam parecer bastante semelhantes, existem algumas importantes diferenças semânticas. Aqueles indicam localizações da memória, mas essas apontam para objetos. Isso torna sem sentido qualquer tipo de operação aritmética sobre as referências, eliminando essa prática propensa a erros. A distinção entre o valor de um ponteiro e o valor para o qual ele aponta cabe ao programador em muitas linguagens, em que os ponteiros, às vezes, devem ser explicitamente "desreferenciados". As referências são sempre implicitamente "desreferenciadas", quando necessário. Assim, elas se comportam mais como variáveis escalares comuns.

O Java tem um tipo booleano primitivo, usado principalmente para as expressões de controle de suas instruções de controle (como `if` e `while`). Diferentemente do C e do C++, expressões aritméticas não podem ser usadas como expressões de controle. O Java não tem nenhum tipo de registro, de união ou de enumeração.

Uma diferença significativa entre o Java e muitas de suas contemporâneas que suportam programação orientada a objeto, inclusive a Ada 95 e o C++, é que não é possível escrever subprogramas independentes em Java. Todos os subprogramas Java são métodos e definidos em classes. Não há nenhuma construção em Java que seja chamada de função ou de subprograma. Além disso, métodos somente podem ser chamados por intermédio de uma classe ou de um objeto.

Outra importante diferença entre o C++ e o Java é que o C++ suporta herança múltipla diretamente em suas definições de classe. Alguns acham que a herança múltipla acarreta mais complexidade e confusão do que deve. O Java suporta somente herança única, embora alguns benefícios daquela possam ser ganhos, usando-se sua construção de interface.

O Java inclui uma forma relativamente simples de controle de concorrência por meio de seu modificador `synchronize`, que pode aparecer em métodos e em blocos. Em qualquer um dos casos, ele faz com que um bloqueio (*lock*) seja anexado. O bloqueio garante o acesso ou a execução mutuamente exclusivos. Em Java, é relativamente fácil criar processos concorrentes chamados linhas de execução.

O Java usa a desalocação de armazenagem implícita para seus objetos, chamada de "Coleta de lixo"<sup>1</sup>. Isso libera o programador de preocupar-se em devolver o espaço de armazenagem ao monte quando ele não for mais necessário. Programas escritos em linguagens que exigem desalocação explícita freqüentemente padecem daquilo que, às vezes, é chamado

<sup>1</sup>N. de T. Coleta de lixo: *garbage collection*. Uma rotina que procura por segmentos de programa ou de dados na memória que não estão mais ativos para recuperar aquele espaço.

de vazamento de memória, o que significa que um espaço é alocado, mas nunca é desalocado. Isso, evidentemente, pode levar ao eventual esgotamento de toda a memória disponível.

Diferentemente do C e do C++, o Java inclui coerções de tipo (conversões de tipo implícitas) somente se eles estiverem ampliando-se (de um tipo "menor" para um tipo "maior"). Assim, coerções de `int` para `float` são feitas, mas coerções de `float` para `int` não.

### 2.17.3 Avaliação

Os projetistas do Java fizeram bem em aparar os recursos excessivos e/ou inseguros do C++. Por exemplo, a eliminação de metade das coerções que são feitas no C++ foi claramente um passo na direção de uma maior confiabilidade. A verificação da faixa de índice nos acessos a matrizes também torna a linguagem mais segura. A adição da concorrência aumenta o escopo das aplicações que podem ser escritas na linguagem, o mesmo acontecendo com as bibliotecas de classes para *applets*, interfaces gráficas, acesso a banco de dados e a redes.

Por outro lado, o Java ainda é uma linguagem complexa. Sua falta de herança múltipla leva a alguns casos peculiares. Por exemplo, para rodar um *applet* como uma tarefa que pode ser executada concorrentemente com outras tarefas, é preciso reunir tanto os atributos dos *applets* como das linhas de execução. Esse casamento não é necessariamente bonito.

A portabilidade do Java, pelo menos na forma intermediária, freqüentemente tem sido atribuída ao projeto da linguagem, mas não é. Qualquer linguagem poderia ser traduzida para uma forma intermediária e "rodada" em uma plataforma que dispusesse de uma máquina virtual para essa forma intermediária. O preço dessa espécie de portabilidade é o custo da interpretação, que usualmente é mais ou menos uma ordem de magnitude maior do que a execução do código de máquina. A versão inicial do interpretador Java, chamada *Java Virtual Machine (JVM)*, era de fato pelo menos dez vezes mais lenta que um programa C equivalente. Contudo, melhoramentos significativos têm sido feitos nessa tecnologia e a eficiência de programas Java é agora competitiva com a de programas escritos em linguagens compiladas como o C++.

Atualmente, o Java é amplamente usado para programar páginas da World Wide Web. Antes dele aparecer, qualquer computação significativa exigida por uma página da Web era feita pela *Common Gateway Interface (CGI)* usando alguma aplicação que roda no servidor da Web. Os *applets* Java são pequenos programas que rodam no cliente da Web quando é encontrada uma chamada a eles na HTML da página que está sendo exibida. Quando chamada, a forma de código intermediário do *applet* é descarregada do servidor para o cliente e imediatamente interpretada. A saída é exibida na página da Web.

O uso do Java cresceu mais rapidamente do que o de qualquer outra linguagem de programação. Uma das razões para isso é seu valor para programar páginas da Web elaboradas. Outra é que o sistema compilador/interpretador para o Java tem sido gratuito e fácil de obter na Web. É claro que uma das razões para a rápida elevação do Java à proeminência é simplesmente que os programadores gostam de seu projeto. Finalmente, durante algum tempo, houve um grande número de programadores C++ que apresentava objeções quanto àquilo que percebia como problemas com esta linguagem. O Java lhes oferece uma alternativa com grande parte da potência do C++, mas um número menor de problemas.

O Java atualmente é usado em diferentes áreas de aplicação. Eis um exemplo de programa em Java:

```
// Exemplo de Programa em Java
// Entrada: Um número inteiro, complis, em que
//           complis é menor do que 100, seguido de complis
//           valores inteiros
// 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 complis,
            contador,
            soma = 0,
            media,
            resultado = 0;
        int[] intlist = int[99];
        complis = Integer.parseInt(in.readLine());
        if ((complis > 0) && (complis < 100)) {
/* Leia a entrada em um vetor e compute a soma */
            for (contador = 0; contador < complis; contador++) {
                intlist[contador] =
                    Integer.valueOf(in.readLine()).intValue();
                soma += intlist[contador];
            }
/* Calcule a média */
            media = soma / complis;
/* Conte os valores de entrada que são > do que a média */
            for (contador = 0; contador < complis; contador++)
                if (intlist[contador] > media) resultado++;
/* Imprima o resultado */
            System.out.println(
                "\nNúmero de valores > do que a média é:" +
                resultado);
        } //** fim da cláusula then do if ((complis > 0) ...
        else System.out.println(
            "Erro - o tamanho da lista de entrada não é
            válido\n");
    } //** fim do método principal
} //** fim da classe IntSort
```

**RESUMO**

Investigamos o desenvolvimento e os ambientes de desenvolvimento de um grande número das linguagens de programação mais importantes. Esse capítulo deve ter dado ao leitor uma boa perspectiva sobre as questões atuais no projeto de linguagens. Esperamos ter preparado o cenário para uma discussão em profundidade dos recursos importantes das linguagens contemporâneas.

**NOTAS BIBLIOGRÁFICAS**

Talvez a fonte mais importante de informação histórica sobre o desenvolvimento das linguagens de programação seja *History of Programming Languages*, editada por Richard Wexelblat (Wexelblat, 1981). Ela contém a base do desenvolvimento e o ambiente de 13 importantes linguagens de programação, narrados pelos próprios projetistas. Um trabalho similar resultou de uma segunda conferência sobre a "história", dessa vez publicada como uma edição especial da *ACM SIGPLAN Notices* (ACM, 1993a). Nesse trabalho, a história e a evolução de mais 13 linguagens de programação são discutidas.

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

Outro livro de grande interesse é *Programming Languages: History and Fundamentals*, de Jean Sammet (Sammet, 1969). É um trabalho de 785 páginas repleto de detalhes de 80 linguagens de programação das décadas de 50 e 60. Sammet também publicou diversas atualizações para seu livro, como, por exemplo, Sammet (1976).

**QUESTÕES DE REVISÃO**

1. Em que ano a Plankalkül foi projetada? Em que ano esse projeto foi publicado?
2. Quais duas estruturas de dados comuns foram incluídas na Plankalkül?
3. Como eram implementados os pseudocódigos do início da década de 50?
4. O sistema Speedcoding foi inventado para superar duas significativas deficiências do hardware de computador no início da década de 50. Quais eram essas duas deficiências?
5. Por que a lentidão de interpretação de programas era aceitável no início da década de 50?
6. Quais dois importantes recursos de hardware apareceram primeiro no computador IBM 704?
7. Em que ano o projeto do FORTRAN teve início?
8. Qual era a principal área de aplicação dos computadores na época em que o FORTRAN foi projetado?
9. Qual era a fonte de todas as instruções de fluxo de controle do FORTRAN I?
10. Qual foi o mais significativo recurso adicionado ao FORTRAN I para obter o FORTRAN II?
11. Quais instruções de fluxo de controle foram adicionadas ao FORTRAN IV para obter o FORTRAN 77?
12. Qual versão do FORTRAN foi a primeira a ter qualquer tipo de variável dinâmica?
13. Qual versão do FORTRAN foi a primeira a ter manipulação de cadeias de caracteres?
14. Por que os lingüistas estavam interessados na inteligência artificial no final da década de 50?
15. Onde o LISP foi desenvolvido? Por quem?
16. De que maneira as linguagens Scheme e COMMON LISP são opostas uma à outra?
17. Qual dialeto do LISP é usado para cursos introdutórios de programação em algumas universidades?
18. Quais duas organizações profissionais juntas projetaram o ALGOL 60?
19. Em qual versão do ALGOL apareceu a estrutura em blocos?

20. Qual elemento de linguagem em falta no ALGOL 60 prejudicou suas chances de obter um uso generalizado?
21. Qual linguagem foi projetada para descrever a sintaxe do ALGOL 60?
22. Em qual linguagem o COBOL baseou-se?
23. Em que ano o processo do projeto do COBOL teve início?
24. Qual estrutura de dados que apareceu no COBOL que se originou na Plankalkül?
25. Qual organização foi a maior responsável pelo sucesso inicial do COBOL (em termos de extensão de uso)?
26. Qual grupo de usuários era o alvo na primeira versão do BASIC?
27. Por que o BASIC foi uma linguagem importante no início da década de 80?
28. A PL/I foi projetada para substituir quais duas linguagens?
29. Para qual nova linha de computadores a PL/I foi projetada?
30. Quais recursos da SIMULA 67 agora são partes importantes de algumas linguagens orientadas a objeto?
31. Qual inovação de estruturação de dados foi introduzida no ALGOL 68, mas freqüentemente é creditada ao Pascal?
32. Qual critério de projeto foi usado extensivamente no ALGOL 68?
33. Qual linguagem introduziu a instrução **case**?
34. Quais operadores do C foram modelados em operadores similares no ALGOL 68?
35. Quais as duas características do C que o tornam menos seguro do que o Pascal?
36. O que é uma linguagem não-baseada em procedimentos (*nonprocedural*)?
37. Quais são os dois tipos de instruções que compõem um banco de dados Prolog?
38. Qual é a principal área de aplicação para a qual a Ada foi projetada?
39. Como são chamadas as unidades de programa concorrentes da Ada?
40. Qual construção da Ada oferece suporte para tipos de dados abstratos?
41. O que povoa o mundo da Smalltalk?
42. Quais três conceitos são a base da programação orientada a objeto?
43. Por que o C++ inclui os recursos do C que são conhecidos como inseguros?
44. O que as linguagens Ada e COBOL têm em comum?
45. Qual foi a primeira aplicação para o Java?
46. Dê dois motivos pelos quais o Java é mais seguro do que o C++.

## PROBLEMAS

1. Quais recursos da Plankalkül você acha que teriam tido a maior influência sobre o FORTRAN 0 se os projetistas do FORTRAN estivessem familiarizados com a Plankalkül?
2. Descreva as capacidades do sistema 701 Speedcoding de Backus e compare-as com as de uma calculadora manual programável contemporânea.
3. Escreva uma breve história dos sistemas A-0, A-1 e A-2 projetados por Grace Hopper e seus colegas.
4. Como um projeto de pesquisa, compare as facilidades do FORTRAN 0 com as do sistema de Laning e Zierler.
5. Quais das três metas originais do comitê de projeto ALGOL, em sua opinião, foi a mais difícil de atingir naquela época?
6. Faça uma suposição fundamentada de qual é o erro de sintaxe mais comum em programas LISP.
7. O LISP iniciou-se como uma linguagem funcional pura, mas, pouco a pouco, adquiriu cada vez mais recursos imperativos. Por quê?
8. Descreva, em detalhes, as três razões mais importantes, em sua opinião, pelas quais o ALGOL 60 não se tornou uma linguagem amplamente usada.
9. Por que, em sua opinião, o COBOL permite identificadores longos quando o FORTRAN e o ALGOL não?
10. Qual é a principal razão que você já ouviu, segundo a qual os cientistas da computação raramente usam o BASIC?

11. Esboce a principal motivação da IBM para desenvolver a PL/I.
12. A motivação da IBM para desenvolver a PL/I foi correta, dada a história dos computadores e dos desenvolvimentos de linguagens desde 1964?
13. Descreva, com suas próprias palavras, o conceito de ortogonalidade no projeto de linguagens de programação.
14. Qual é a razão principal, em sua opinião, pela qual a PL/I tornou-se mais amplamente usada do que o ALGOL 68?
15. Quais são os argumentos tanto a favor como contra a idéia de uma linguagem sem tipos?
16. Existe alguma linguagem de programação lógica além do Prolog?
17. Qual é a sua opinião a respeito do argumento segundo o qual linguagens demasiadamente complexas são muito perigosas de usar e que, portanto, devemos manter todas as linguagens pequenas e simples?
18. Você acha uma boa idéia que um comitê projete uma linguagem? Sustente sua opinião.
19. As linguagens evoluem continuamente. Quais tipos de restrições você acha que são apropriadas para alterações nas linguagens de programação? Compare suas respostas com a evolução do FORTRAN.
20. Construa uma tabela identificando todos os grandes desenvolvimentos de linguagem, além de quando eles ocorreram, em qual linguagem apareceram primeiro e as identidades dos desenvolvedores.

## Capítulo 3

# Descrevendo a Sintaxe e a Semântica



**Grace M. Hopper**

Grace M. Hopper, uma oficial da Marinha e ex-funcionária da UNIVAC, projetou uma série de sistemas “compiladores” do início a meados da década de 50, que eram usados para programação de aplicações comerciais. Em 1958, esses sistemas evoluíram para a primeira linguagem de programação de alto nível para aplicações comerciais, a FLOW-MATIC, na qual o COBOL, em grande parte, baseou-se. Ela também esteve envolvida no esforço de projeto do COBOL, servindo como consultora para o comitê executivo da CODASYL.

- 3.1** Introdução
- 3.2** O Problema Geral de Descrever a Sintaxe
- 3.3** Métodos Formais de Descrever a Sintaxe
- 3.4** Gramática de Atributos
- 3.5** Descrevendo o Significado dos Programas: Semântica Dinâmica

Neste capítulo serão definidos, primeiramente, os termos *sintaxe* e *semântica*. Depois, será apresentada uma discussão detalhada do método mais comum para descrever a sintaxe: a gramática livre de contexto (também conhecida como Forma de Backus-Naur). Essa será seguida por uma descrição dos diagramas de sintaxe. A gramática de atributos, que pode ser usada para descrever tanto a sintaxe como a semântica estática das linguagens de programação, será discutida em seguida. Por fim, três métodos formais para descrever a semântica — operacional, axiomática e denotacional — serão introduzidos. Por causa da inerente complexidade dos métodos de descrição da semântica, nossa discussão sobre eles será breve. Pode-se escrever facilmente um livro inteiro apenas sobre um dos três (e diversos autores o fizeram).

### **3.1 Introdução**

---

A tarefa de apresentar uma descrição concisa e inteligível de uma linguagem de programação é difícil, mas fundamental para o sucesso dela. O ALGOL 60 e o ALGOL 68 foram apresentados com descrições formais concisas; em ambos os casos, porém, as descrições não eram facilmente compreensíveis, parcialmente porque cada uma usava uma notação nova. Os níveis de aceitação de ambas as linguagens sofreram em consequência disso. Por outro lado, algumas linguagens padeceram do problema de ter muitos dialetos ligeiramente diferentes, uma consequência de uma definição comprehensível, mas informal e imprecisa.

Um dos problemas para descrever uma linguagem é a diversidade das pessoas que devem entender a descrição. A maioria das novas linguagens de programação estão sujeitas a um período de escrutínio pelos usuários potenciais antes que seus projetos sejam concluídos. O sucesso desse ciclo de reavaliação depende muito da clareza da descrição.

Os implementadores de linguagens de programação, evidentemente, devem ser capazes de determinar como as expressões, as instruções e as unidades de programa são formadas, e, também, o efeito pretendido pelas mesmas quando executadas. A dificuldade do trabalho dos implementadores é, em parte, determinada pela clareza e pela precisão da descrição da linguagem.

Finalmente, os usuários devem ser capazes de determinar como codificar sistemas consultando um manual de referência da linguagem. Os livros didáticos e os cursos entram nesse processo, mas os manuais normalmente são a única fonte de informação impressa autorizada sobre uma linguagem.

O estudo das linguagens de programação, à semelhança do estudo das naturais, pode ser dividido em exames da sintaxe e da semântica. A **sintaxe** de uma linguagem de programação é a forma de suas expressões, de suas instruções e de suas unidades de programa. Sua **semântica** é o significado destes três. Por exemplo, a sintaxe da instrução **if** da linguagem C é

```
if (<expr>) <instrução>
```

A semântica dessa forma de instrução obedece a seguinte condição: se o valor atual da expressão for verdadeiro, a instrução incorporada será selecionada para execução.

A sintaxe e a semântica estão estreitamente relacionadas, embora freqüentemente sejam separadas para propósitos de discussão. Em uma linguagem de programação bem projetada, a semântica deve seguir-se diretamente da sintaxe; ou seja, a forma de uma instrução deve sugerir fortemente o que esta pretende realizar.

Descrever a sintaxe é mais fácil do que a semântica, em parte porque uma notação concisa e universalmente aceita está disponível para descrição da sintaxe, mas nenhuma foi desenvolvida ainda para a semântica.

## 3.2 O Problema Geral de Descrever a Sintaxe

As linguagens, sejam naturais (como o Inglês) ou artificiais (como o Java), são conjuntos de seqüências de caracteres de algum alfabeto. As seqüências são chamadas **sentenças** ou instruções. As regras de sintaxe especificam quais seqüências de caracteres do alfabeto da linguagem estão nela. O Inglês, por exemplo, tem uma coleção grande e complexa de regras para especificar a sintaxe de suas sentenças. Em comparação, até mesmo as maiores e mais complexas linguagens de programação são sintaticamente muito simples.

As descrições formais da sintaxe das linguagens de programação, em nome da simplicidade, freqüentemente não incluem descrições das unidades sintáticas de nível mais baixo. Essas pequenas unidades são chamadas **lexemas** (*lexemes*), cuja descrição pode ser dada por uma especificação léxica, a qual pode estar separada da descrição sintática. Os lexemas de uma linguagem de programação incluem seus identificadores, literais, seus operadores e suas palavras especiais. Pode-se imaginar os programas como seqüências de lexemas em vez de caracteres.

Um **símbolo** (*token*) de uma linguagem é uma categoria de seus lexemas. Por exemplo, um identificador é um símbolo que pode conter lexemas ou instâncias, como por exemplo, `soma` e `total`. Em alguns casos, um símbolo tem somente um único lexema possível. Por exemplo, o símbolo para o operador aritmético `+`, que pode ter o nome `op_soma`, tem apenas um lexema possível. Considere o seguinte exemplo de instrução C:

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

Os lexemas e os símbolos dessa instrução são:

Lexemas	Símbolos
index	identificador
=	sinal_igual
2	int_literal
*	op_mult
cont	identificador
+	op_soma
17	int_literal
;	ponto_e_virgula

Os exemplos de descrições de linguagens deste capítulo são simples e a maioria inclui descrições dos lexemas.

### 3.2.1 Reconhecedores da Linguagem

Em geral, as linguagens podem ser formalmente definidas de duas maneiras diferentes: por **reconhecimento** e por **geração** (ainda que nenhuma delas forneça uma definição prática).

ca em si para as pessoas tentarem aprender ou mesmo usar uma linguagem de programação). Suponhamos que uma linguagem  $L$  use o alfabeto  $\Sigma$  de caracteres. Para definirmos  $L$  formalmente, usando o método do reconhecimento, precisaríamos construir um mecanismo  $R$ , denominado dispositivo de reconhecimento, capaz de ler seqüências de caracteres do alfabeto  $\Sigma$ .  $R$  precisaria ser projetado de tal forma que indique se determinada seqüência de entrada estava ou não em  $L$ . Com efeito,  $R$  aceitaria ou rejeitaria a seqüência dada. Tais dispositivos são como filtros, separando as sentenças corretas das incorretamente formadas. Se  $R$ , quando alimentado com qualquer seqüência de caracteres de  $\Sigma$ , aceitá-la somente se ela estiver em  $L$ , então  $R$  será uma descrição de  $L$ . Uma vez que as linguagens mais úteis são, para todos os efeitos práticos, infinitas, esse poderia parecer um processo extenso e ineficiente. Os dispositivos de reconhecimento, entretanto, não são usados para enumerar todas as sentenças de uma linguagem.

A parte de análise sintática de um compilador é um reconhecedor da linguagem que o mesmo traduz. Nesse papel, o reconhecedor não precisa testar todas as seqüências possíveis de caracteres de um conjunto para determinar se cada uma delas está na linguagem. Ao contrário, precisa somente definir se determinados programas estão na linguagem. Efetivamente, então, o analisador da sintaxe determina se os programas dados estão sintaticamente corretos. Os analisadores sintáticos serão discutidos no Capítulo 4.

### 3.2.2 Geradores de Linguagem

Um gerador de linguagem é um dispositivo que pode ser usado para gerar as sentenças. Podemos imaginar o gerador como tendo um botão que, quando pressionado, produz uma sentença da linguagem. Uma vez que a sentença particular é imprevisível quando produzida por um gerador após pressionar o seu botão, esse parece ser um dispositivo de utilidade limitada como um descritor da linguagem. Porém, as pessoas preferem certas formas de geradores aos reconhecedores porque podem lê-los e entendê-los mais facilmente. Em comparação, a parte de verificação sintática de um compilador (um reconhecedor de linguagem) não é uma descrição tão útil para o programador porque ela somente pode ser usada no modo de tentativa e de erro. Por exemplo, para determinar a sintaxe correta de uma instrução particular, usando um compilador, o programador somente pode submeter uma versão suposta e ver se o compilador aceita. Por outro lado, freqüentemente é possível determinar se a sintaxe de uma instrução particular está correta, comparando-a com a estrutura do gerador.

Há uma estreita conexão entre geração formal e dispositivos de reconhecimento para a mesma linguagem. Essa foi uma das descobertas originais da ciência da computação, e levou a grande parte daquilo que se sabe, hoje, sobre as linguagens formais e sobre a teoria de projeto de compiladores. Retornaremos à relação entre os geradores e os reconhecedores na próxima seção.

## 3.3 Métodos Formais para Descrever a Sintaxe

Esta seção discutirá os mecanismos de geração formal de linguagem comumente usados para descrever a sintaxe. Tais mecanismos freqüentemente são chamados de gramática.

### 3.3.1 Forma de Backus-Naur e Gramática Livre de Contexto

De meados até o final da década de 50, dois homens, John Backus e Noam Chomsky, em esforços de pesquisa independentes, inventaram a mesma notação, que se tornou, desde então, o método mais usado para descrever formalmente a sintaxe das linguagens de programação.

#### 3.3.1.1 Gramática Livre de Contexto

Em meados da década de 50, Chomsky, um famoso lingüista, descreveu quatro classes de dispositivos generativos ou gramáticas que definem quatro classes de linguagens (Chomsky, 1956, 1959). Duas dessas classes gramaticais, denominadas livres de contexto e regulares, passaram a ser úteis para descrever a sintaxe das linguagens de programação. Os símbolos destas podem ser descritos por meio da gramática regular. Linguagens inteiras de programação, com pequenas exceções, podem ser descritas pela gramática livre de contexto. Uma vez que Chomsky era um lingüista, seu principal interesse era a natureza teórica das linguagens naturais. Ele não tinha interesse, naquela época, nas linguagens artificiais usadas para a comunicação com computadores. Assim, foi somente mais tarde que seu trabalho se aplicou à informática.

#### 3.3.1.2 Origens da Forma de Backus-Naur

Pouco depois do trabalho de Chomsky sobre classes de linguagens, o grupo da ACM-GAMM começou a projetar o ALGOL 58. Um documento decisivo descrevendo o ALGOL 58 foi apresentado por John Backus, um proeminente membro do grupo ACM-GAMM, em uma conferência internacional em 1959 (Backus, 1959). Esse documento introduziu uma nova notação formal para especificar a sintaxe das linguagens de programação. A notação foi ligeiramente modificada mais tarde por Peter Naur para a descrição do ALGOL 60 (Naur, 1960). Esse método revisado da descrição da sintaxe tornou-se conhecido como **forma de Backus-Naur** ou simplesmente **BNF**.

A BNF é uma notação muito natural para descrever a sintaxe. De fato, algo similar à BNF foi usado por Panini para descrever a sintaxe do Sânsrito várias centenas de anos antes de Cristo (Ingerman, 1967).

A BNF tornou-se, e ainda continua a ser, o método mais popular para descrever concisamente a sintaxe de uma linguagem de programação, ainda que seu uso no relatório do ALGOL 60 não tenha sido prontamente aceito pelos usuários de computador.

É notável que a BNF seja quase idêntica aos dispositivos generativos de Chomsky para linguagens independentes do contexto, chamadas **gramáticas livres de contexto**. No restante deste capítulo, iremos referir-nos às gramáticas livres de contexto simplesmente por gramáticas. Além disso, os termos BNF e gramáticas são usados de maneira intercambiável.

#### 3.3.1.3 Fundamentos

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

A BNF usa abstrações para estruturas sintáticas. Uma simples instrução de atribuição em C, por exemplo, poderia ser representada pela abstração <atribuição> (colchetes angulares).

lados freqüentemente são usados para delimitar nomes de abstrações). A definição real de <atribuição> poderia ser dada por

$$<\text{atribuição}> \rightarrow <\text{var}> = <\text{expressão}>$$

O símbolo à esquerda da seta, chamado de lado esquerdo (LE), é a abstração que está sendo definida. O texto à direita da seta é a definição do LE. Ele é chamado de lado direito (LD) e consiste em uma mistura de símbolos, de lexemas e de referências a outras abstrações (de fato, símbolos também são abstrações). Ao todo, a definição é chamada **regra ou produção**. No exemplo de regra que acabamos de apresentar, as abstrações <var> e <expressão> evidentemente devem ser definidas antes que a definição <atribuição> torne-se útil.

Essa regra particular especifica que a abstração <atribuição> é definida como uma instância da abstração <var> seguida do lexema =, seguido de uma instância da abstração <expressão>. Um exemplo de sentença cuja estrutura sintática é descrita pela regra é

```
total = sub1 + sub2
```

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

Os símbolos não-terminais podem ter duas ou mais definições distintas, representando duas ou mais formas sintáticas possíveis na linguagem. Múltiplas definições podem ser escritas como uma única regra, com duas definições diferentes separadas pelo símbolo |, que significa o OU lógico. Por exemplo, a instrução Pascal **if** pode ser descrita com as regras

$$\begin{aligned} <\text{inst\_if}> &\rightarrow \text{if } <\text{expr\_lógica}> \text{ then } <\text{inst}> \\ <\text{inst\_if}> &\rightarrow \text{if } <\text{expr\_lógica}> \text{ then } <\text{inst}> \text{ else } <\text{inst}> \end{aligned}$$

ou com a regra

$$\begin{aligned} <\text{inst\_if}> &\rightarrow \text{if } <\text{expr\_lógica}> \text{ then } <\text{inst}> \\ &\quad | \text{ if } <\text{expr\_lógica}> \text{ then } <\text{inst}> \text{ else } <\text{inst}> \end{aligned}$$

A BNF é suficientemente poderosa para descrever a grande maioria das sintaxes das linguagens de programação mesmo sendo simples. Em particular, ela pode descrever listas de construções similares, a ordem em que diferentes construções devem aparecer, as estruturas aninhadas em qualquer profundidade, a precedência e a associatividade de operadores.

### 3.3.1.4 Descrevendo Listas

Em matemática, listas de tamanho variável freqüentemente são escritas usando-se três pontos (...); 1, 2, ... é um exemplo. A BNF não inclui os três pontos, de modo que 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 instrução de declaração de dados). A alternativa mais comum é a recursão. Uma regra é **recursiva** se o LE aparecer em seu LD. As regras seguintes ilustram como a recursão é usada para descrever listas:

$$\begin{aligned} <\text{lista\_ident}> &\rightarrow \text{identificador} \\ &\quad | \text{ identificador , } <\text{lista\_ident}> \end{aligned}$$

Isso define `<lista_ident>` ou como um único símbolo, ou como um identificador seguido de uma vírgula seguida de outra instância de `<lista_ident>`. A recursão é usada para descrever listas em muitos dos exemplos de gramáticas do restante deste capítulo.

### 3.3.1.5 Gramáticas e Derivações

A BNF é um dispositivo gerativo para definir linguagens. As sentenças da linguagem são geradas por uma seqüência de aplicações das regras, iniciando-se com um símbolo não-terminal da gramática chamado **símbolo de início**. Uma geração de sentença é chamada **derivação**. Em uma gramática para uma linguagem completa, o símbolo de início representa um programa completo e usualmente é chamado `<programa>`. A gramática simples mostrada no Exemplo 3.1 é usada para ilustrar derivações:

---

#### *EXEMPLO 3.1 Uma Gramática para uma Pequena Linguagem*

---

```

<programa> → begin <lista_inst> end
<lista_inst> → <inst>
| <inst> ; <lista_inst>
<inst> → <var> = <expressão>
<var> → A | B | C
<expressão> → <var> + <var>
| <var> - <var>
| <var>

```

---

A linguagem do Exemplo 3.1 tem somente uma forma de instrução: atribuição. Um programa consiste na palavra inicial `begin` seguida de uma lista de instruções separadas por ponto-e-vírgulas, seguidas da palavra especial `end`. Uma expressão é ou uma variável única, ou duas variáveis separadas pelo operador `+` ou pelo operador `-`. Os únicos nomes de variáveis nessa linguagem são `A`, `B` e `C`.

Uma derivação de um programa nessa linguagem é a seguinte:

```

<programa> = > begin <lista_inst> end
= > begin <inst> ; <lista_inst> end
= > begin <var> = <expressão> ; <lista_inst> end
= > begin A = <expressão> ; <lista_inst> end
= > begin A = <var> + <var> ; <lista_inst> end
= > begin A = B + <var> ; <lista_inst> end
= > begin A = B + C ; <lista_inst> end
= > begin A = B + C ; <inst> end
= > begin A = B + C ; <var> := <expressão> end
= > begin A = B + C ; B = <expressão> end
= > begin A = B + C ; B = <var> end
= > begin A = B + C ; B = C end

```

Essa derivação, igual a todas, começa com o símbolo de iniciar, neste caso, `<programa>`. Lê-se o símbolo `= >` como "deriva". Cada cadeia sucessiva na seqüência é derivada da cadeia anterior substituindo um dos não-terminais por uma das definições do mesmo. Cada uma

das cadeias da derivação, inclusive <programa>, é chamada de **forma sentencial**. Nessa derivação, o não-terminal substituído sempre é o da extrema esquerda na forma sentencial anterior. Derivações que usam tal ordem de substituição são chamadas **derivações à extrema esquerda** (*leftmost derivations*). A derivação prossegue até que a forma sentencial não contenha nenhum não-terminal. A forma sentencial, composta somente de terminais ou de lexemas, é a sentença gerada.

Além da derivação à extrema esquerda, uma derivação pode ser à extrema direita ou em uma ordem que não seja nem à extrema esquerda, nem à extrema direita. A ordem de derivação não tem nenhum efeito sobre a linguagem gerada por uma gramática.

Ao escolher LDs alternativos de regras para substituir não-terminais na derivação, é possível gerar diferentes sentenças na linguagem. Ao escolher exaustivamente todas as combinações de opções, a linguagem inteira pode ser gerada. Essa linguagem, como a maioria das outras, é infinita, de modo que não se pode gerar todas as sentenças da mesma em tempo finito.

O Exemplo 3.2 é outra gramática de parte de uma típica linguagem de programação:

### *EXEMPLO 3.2 Uma Gramática para Instruções de Atribuição Simples*

```

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

A gramática do Exemplo 3.2 descreve instruções de atribuição cujos lados direitos são expressões aritméticas com operadores de multiplicação e adição e com parênteses. Por exemplo, a instrução

$A = B * ( A + C )$

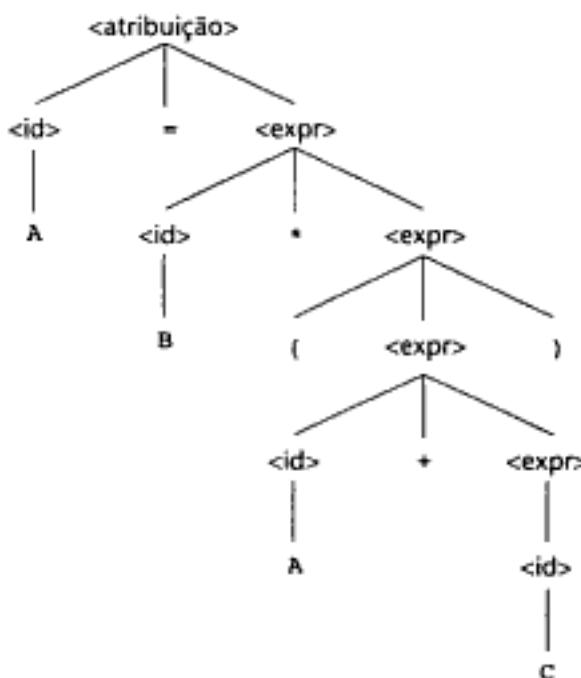
é gerada pela derivação à extrema esquerda:

```

<atribuição> = > <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 (Parse Trees)

Uma das características mais atraentes das gramáticas é que elas descrevem naturalmente a estrutura sintática hierárquica das linguagens que definem; tais estruturas são chamadas **árvores de análise** (parse trees). Por exemplo, a árvore de análise da Figura 3.1 mostra a estrutura da instrução de atribuição derivada acima.



**FIGURA 3.1** Uma árvore de análise para a instrução simples  $A = B * (A + C)$

Todo vértice interno de uma árvore de análise é rotulado com um símbolo não-terminal; toda folha é rotulada com um símbolo terminal. Toda subárvore de uma árvore de análise descreve uma instância de uma abstração na instrução.

### 3.3.1.7 Ambigüidade

Diz-se que uma gramática é **ambígua** quando gera uma sentença para a qual há duas ou mais árvores de análise distintas. Considere a gramática mostrada no Exemplo 3.3, uma variação da gramática do Exemplo 3.2.

---

#### EXEMPLO 3.3 Uma Gramática Ambígua para Instruções de Atribuição Simples

---

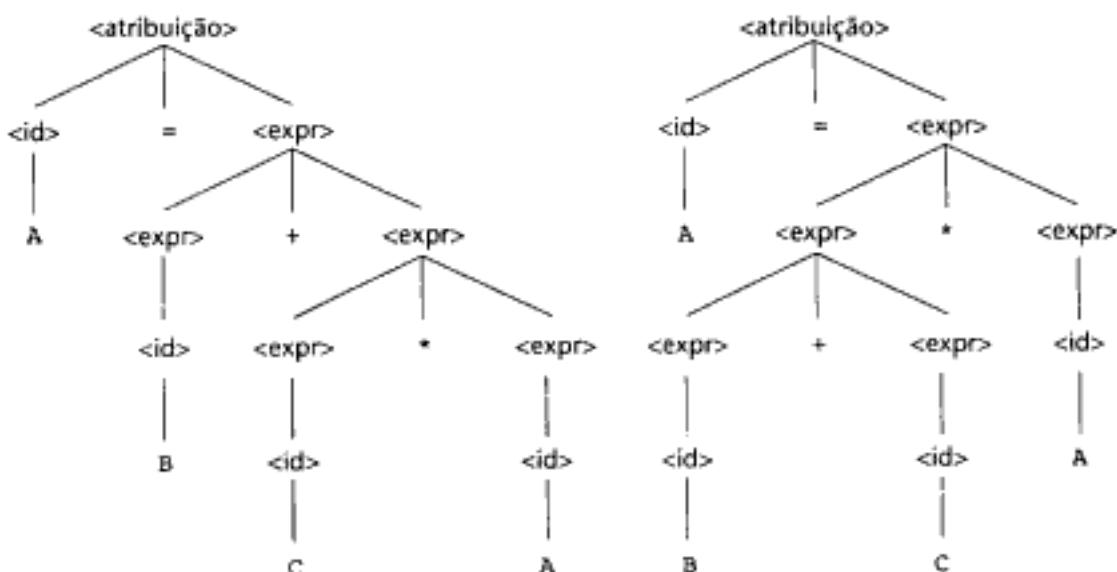
```

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

A gramática do Exemplo 3.3 é ambígua porque a sentença

$$A = B + C * A$$

tem duas árvores de análise distintas, como mostra a Figura 3.2.



**FIGURA 3.2** Duas árvores de análise distintas para a mesma sentença  $A = B + C * A$

A ambigüidade ocorre porque a gramática especifica ligeiramente menos a estrutura sintática do que a gramática do Exemplo 3.2. Em vez de permitir que a árvore de análise de uma expressão cresça somente à direita, essa gramática permite o crescimento tanto à esquerda como à direita.

A ambigüidade sintática das estruturas de linguagem é um problema porque os compiladores freqüentemente baseiam a semântica dessas estruturas em suas formas sintáticas. Em especial, o compilador decide qual código gerar para uma instrução, examinando sua árvore de análise. Se uma estrutura de linguagem tiver mais de uma árvore de análise, o significado daquela não poderá ser determinado de maneira única. Esse problema será discutido em dois exemplos específicos nas três seções seguintes.

### 3.3.1.8 Precedência de Operadores

Conforme afirmou-se anteriormente, uma gramática pode descrever certa estrutura sintática de tal forma que parte do significado da estrutura possa ser deduzido de sua árvore de análise. Em particular, o fato de um operador em uma expressão aritmética ser gerado mais baixo na árvore de análise (e, portanto, dever ser analisado primeiro) pode ser usado para indicar que ele tem precedência sobre um operador produzido mais acima na árvore. Na primeira árvore de análise da Figura 3.2, por exemplo, o operador de multiplicação é gerado mais baixo na árvore, o que poderia indicar que ele tem precedência sobre o operador de adição na expressão. A segunda árvore, entretanto, indica exatamente o oposto. Parece, portanto, que as duas indicam informações de precedência conflitantes.

Note que, embora a gramática do Exemplo 3.2 não seja ambígua, a ordem de precedência de seus operadores não é usual. Nessa gramática, uma árvore de análise de uma

sentença com múltiplos operadores, independentemente dos operadores envolvidos, tem o operador da extrema direita na expressão no ponto mais baixo daquela, com os outros operadores na árvore movendo-se progressivamente mais para cima à medida que se move para a esquerda na expressão. Por exemplo, na árvore da Figura 3.1, o operador mais (+) é o da extrema direita na expressão e o mais baixo naquela, dando a ele precedência sobre o operador de multiplicação à sua esquerda.

Uma gramática pode ser escrita para separar os operadores de adição e de multiplicação, a fim de que eles fiquem coerentemente em uma disposição que vai do mais alto para o mais baixo, respectivamente, na árvore de análise. Essa disposição pode ser mantida independentemente da ordem em que os operadores aparecem em uma expressão. A disposição correta é especificada usando-se abstrações separadas para os operandos dos operadores com precedência diferente. Isso requer não-terminalis adicionais e algumas regras novas. A gramática do Exemplo 3.4 é uma dessas.

#### *EXEMPLO 3.4 Uma Gramática Não-Ambígua para Expressões*

```

<atribuição> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
         | <termo>
<termo> → <termo> * <fator>
         | <fator>
<fator> → ( <expr> )
         | <id>
  
```

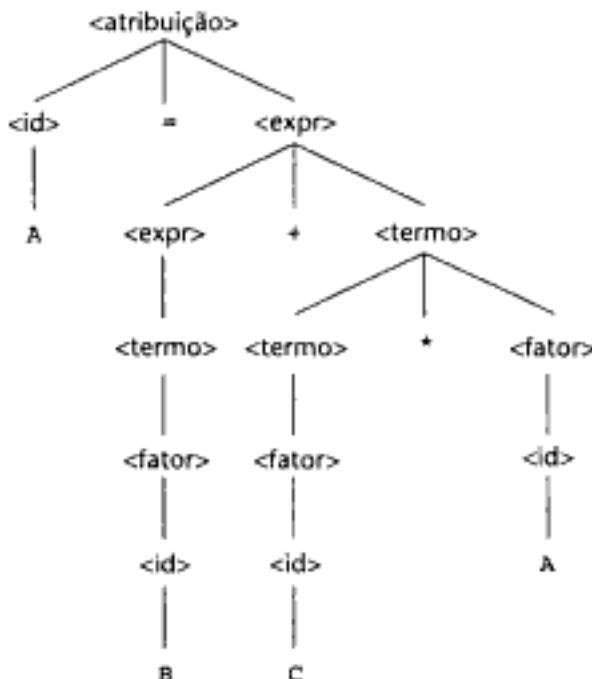
A gramática do Exemplo 3.4 gera a mesma linguagem que as dos Exemplos 3.2 e 3.3, mas ela indica a ordem de precedência usual dos operadores de multiplicação e de adição. A derivação seguinte da sentença  $A = B + C * A$  usa a gramática do Exemplo 3.4:

```

<atribuição> = > <id> = <expr>
                = > A = <expr>
                = > A = <expr> + <termo>
                = > A = <termo> + <termo>
                = > A = <fator> + <termo>
                = > A = <id> + <termo>
                = > A = B + <termo>
                = > A = B + <termo> * <fator>
                = > A = B + <fator> * <fator>
                = > A = B + <id> * <fator>
                = > A = B + C * <fator>
                = > A = B + C * <id>
                = > A = B + C * A
  
```

A árvore de análise única para essa sentença, usando a gramática do Exemplo 3.4, é mostrada na Figura 3.3 (ver p. 120).

A conexão entre árvores de análise e derivações é muito estreita: cada uma pode ser facilmente construída a partir da outra. Toda derivação com uma gramática não-ambígua



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

tem uma árvore de análise única, não obstante esta árvore possa ser representada por diferentes derivações. Por exemplo, a seguinte derivação da sentença  $A = B + C * A$  é diferente da derivação daquela dada anteriormente. Essa é uma derivação à extrema direita, ao passo que a anterior é à extrema esquerda. Ambas, porém, são representadas pela mesma árvore de análise.

```

<atribuição> = > <id> = <expr>
= > <id> = <expr> + <termo>
= > <id> = <termo> + <termo> + <fator>
= > <id> = <expr> + <termo> * <id>
= > <id> = <expr> + <termo> * A
= > <id> = <expr> + <fator> * A
= > <id> = <expr> + <id> * A
= > <id> = <expr> + C * A
= > <id> = <termo> + C * A
= > <id> = <fator> + C * A
= > <id> = <id> + C * A
= > <id> = B + C * A
= > A = B + C * A
  
```

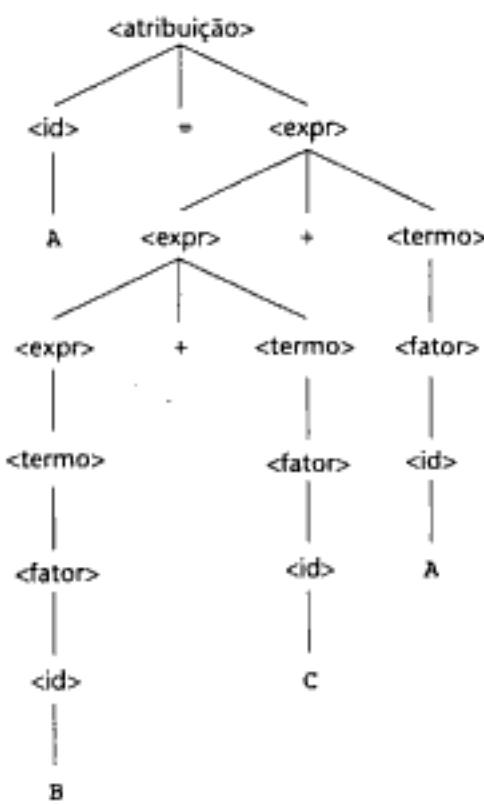
### 3.3.1.9 Associatividade de Operadores

Outra questão interessante a respeito das gramáticas para expressões é se a associatividade de operadores também é corretamente descrita; ou seja, as árvores de análise para expressões com duas ou mais ocorrências de operadores adjacentes com igual precedência têm essas ocorrências na ordem hierárquica apropriada? Um exemplo de instrução de atribuição com uma dessas expressões é:

$$A = B + C + A.$$

A árvore de análise para essa sentença, conforme a definição da gramática do Exemplo 3.4, é mostrada na Figura 3.4.

Esta árvore de análise mostra o operador de adição esquerdo mais baixo do que o operador de adição direito. Essa é a ordem correta se a adição pretende ser associativa à esquerda, o que é típico. Na maioria dos casos, a associatividade da adição em um computador é irrelevante. Em Matemática, a adição é associativa, o que significa que as ordens de avaliação associativas à esquerda e à direita significam a mesma coisa; ou seja,  $(A + B) + C = A + (B + C)$ . A aritmética computadorizada de números inteiros também é associativa. Em algumas situações, entretanto, a adição com números não é. Por exemplo, suponhamos que os valores reais armazenem sete dígitos de precisão. Considere os problemas para adicionar 11 números, reais, em que um deles seja  $10^7$  e os outros 10 sejam 1. Se os números pequenos (os 1s) forem adicionados cada um ao número grande, um de cada vez, não haverá nenhum efeito sobre esse número, porque os pequenos ocorrerão no oitavo dígito do grande. Porém, se os pequenos forem adicionados primeiro, e o resultado adicionado ao número grande, o produto na precisão de sete dígitos será  $1.000001 \times 10^7$ . A subtração e a divisão não são associativas, seja na Matemática ou em um computador. Portanto, a associatividade correta pode ser fundamental para uma expressão que contenha qualquer uma delas.



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

Quando a regra BNF tem seu LE também aparecendo no início de seu LD, diz-se que a regra é **recursiva à esquerda**, o que especifica a associatividade à esquerda. Por exemplo, a recursão à esquerda das regras da gramática do Exemplo 3.4 faz com que ela torne tanto a adição como a multiplicação associativas à esquerda.

Na maioria das linguagens que o oferece, o operador de exponenciação é associativo à direita. Para indicar a associatividade à direita, pode-se usar a recursão para o mesmo lado. Uma regra de gramática é **recursiva à direita** se o LE aparecer na extremidade direita do LD. Regras como

```
<fator> → <exp> ** <fator>
          | <exp>
<exp>   → ( <expr> )
          | <id>
```

poderiam ser usadas para descrever a exponenciação como um operador associativo à direita.

### 3.3.1.10 Uma Gramática Não-Ambígua para if-then-else

As regras BNF dadas na Seção 3.3.1.3 para uma forma particular de instrução **if-then-else** são repetidas aqui:

```
<inst_if> → if <expr_lógica> then <inst>
           | if <expr_lógica> then <inst> else <inst>
```

Se também tivermos  $\langle \text{inst} \rangle \rightarrow \langle \text{inst\_if} \rangle$ , esta gramática será ambígua. A forma sentencial mais simples que ilustra a ambigüidade é

```
if <expr_lógica> then if <expr_lógica> then <inst> else <inst>
```

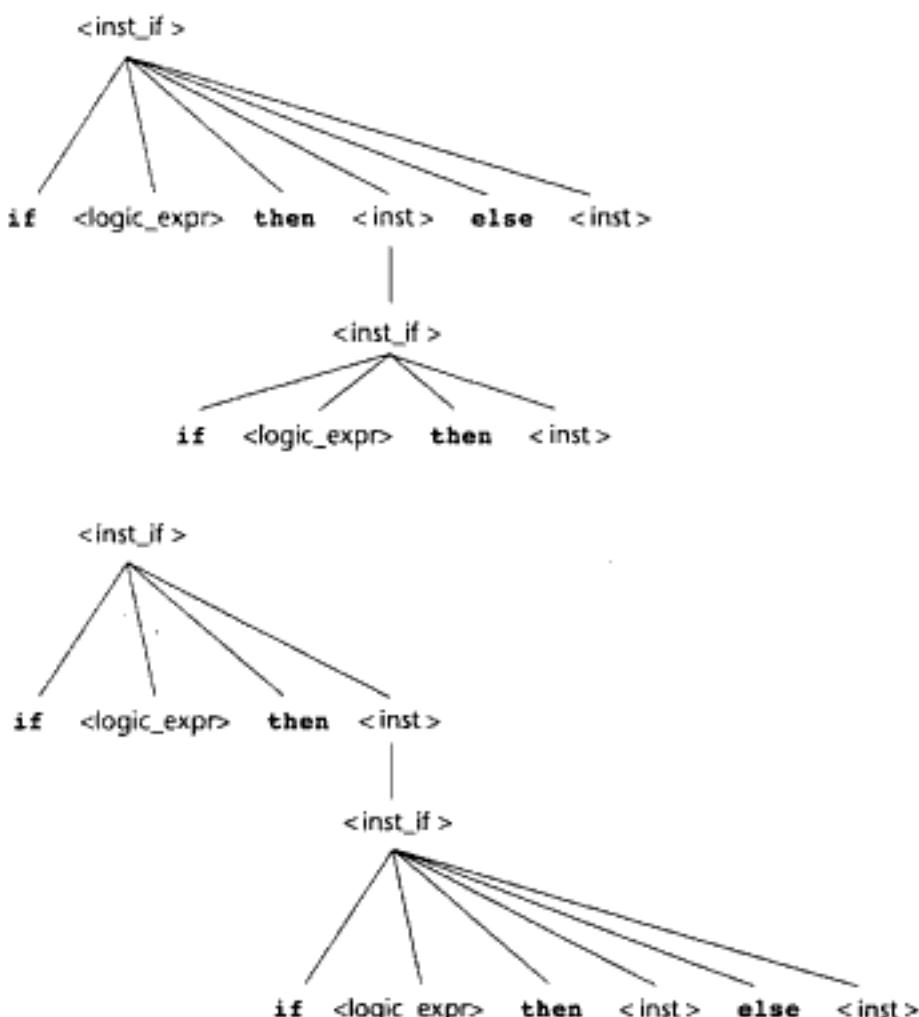
As duas árvores de análise da Figura 3.5 mostram a ambigüidade dessa forma sentencial. Examinaremos as questões ligadas a esse problema de associação do **else** no Capítulo 8.

Agora, desenvolveremos uma gramática não-ambígua que descreve a instrução **if**. A regra para construções **if**, na maioria das linguagens, é que uma cláusula **else**, quando presente, é casada com a **then** anterior mais próxima ainda não-utilizada. Portanto, entre uma **then** e sua **else** coincidente, não pode haver uma instrução **if** sem uma cláusula **else**. Sendo assim, para essa situação, as instruções devem ser distinguidas entre as casadas e as livres; cujas livres são **ifs sem else** e todas as outras instruções são casadas. O problema com a gramática acima é que ela trata todas as instruções como se tivessem uma significação sintática igual, ou seja, como se todas fossem casadas.

Para refletir as diferentes categorias de instruções, diferentes abstrações, ou não-terminais, devem ser usadas. A gramática não-ambígua baseada nessas idéias é a seguinte:

```
<inst> → <casada> | <livre>
<casada> → if <expr_lógica> then <casada> else <casada>
           | qualquer instrução não-if
<livre>  →  if <expr_lógica> then <inst>
           | if <expr_lógica> then <casada> else <livre>
```

Há apenas uma árvore de análise possível, usando esta gramática, para a sentença mostrada na Figura 3.5.



**FIGURA 3.5** Duas árvores de análise distintas para a mesma forma sentencial.

### 3.3.2 BNF Estendida

Por causa de algumas pequenas inconveniências na BNF, ela foi estendida de diversas maneiras. A maioria das versões assim construídas é chamada de *Extended BNF*, ou simplesmente EBNF, ainda que nem todas sejam exatamente a mesma coisa. As extensões não aumentam o poder descritivo da LD, aumentam, simplesmente, sua legibilidade e sua capacidade de escrita.

Três extensões são comumente incluídas nas várias versões da EBNF. A primeira delas denota uma parte opcional de um RHS, que é delimitada por colchetes. Por exemplo, uma instrução de seleção C pode ser descrita como

<seleção> → if ( <expressão> ) <instrução> [else <instrução>];

Sem o uso dos colchetes, a descrição sintática dessa instrução exigiria duas regras.

A segunda extensão é o uso de chaves em um LD para indicar que a parte nelas contida pode ser repetida indefinidamente ou omitida completamente. Esta permite que listas sejam construídas com uma única regra, em vez de usar recursão e duas regras. Por exemplo, listas de identificadores separadas por vírgulas podem ser descritas da maneira mostrada a seguir.

$\langle \text{lista\_ident} \rangle \rightarrow \langle \text{identificador} \rangle \ , \ \langle \text{identificador} \rangle$

Essa é uma substituição da recursão por uma forma de iteração implícita; a parte contida nas chaves pode ser iterada qualquer número de vezes.

A terceira extensão comum lida com opções de múltipla escolha. Quando um único elemento deve ser escolhido de um grupo, as opções são colocadas entre parênteses e separadas pelo operador OU, |. Por exemplo, a regra seguinte descreve uma instrução Pascal **for**:

$\langle \text{inst\_for} \rangle \rightarrow \text{for } \langle \text{var} \rangle \ := \langle \text{expr} \rangle \ (\text{to} \mid \text{downto}) \ \langle \text{expr} \rangle \ \text{do } \langle \text{inst} \rangle$

Mais uma vez, seriam necessários duas regras BNF para descrever essa estrutura. Os colchetes, as chaves e os parênteses nas extensões BNF são metassímbolos, o que significa que eles são ferramentas notacionais e não símbolos terminais nas entidades sintáticas que ajudam a descrever. Nos casos em que esses metassímbolos também são símbolos terminais na linguagem que está sendo descrita, as instâncias que são símbolos terminais podem ser sublinhadas.

### *EXEMPLO 3.5 Versões BNF e EBNF de uma Gramática de Expressão*

BNF:	$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle \ + \ \langle \text{termo} \rangle$   $\langle \text{expr} \rangle \ - \ \langle \text{termo} \rangle$   $\langle \text{termo} \rangle$ $\langle \text{termo} \rangle \rightarrow \langle \text{termo} \rangle \ * \ \langle \text{fator} \rangle$   $\langle \text{termo} \rangle \ / \ \langle \text{fator} \rangle$   $\langle \text{fator} \rangle$
EBNF:	$\langle \text{expr} \rangle \rightarrow \langle \text{termo} \rangle \ ((+ \mid -) \ \langle \text{termo} \rangle)$ $\langle \text{termo} \rangle \rightarrow \langle \text{fator} \rangle \ ((\ast \mid /) \ \langle \text{fator} \rangle)$

Algumas versões da EBNF permitem que um sobreescrito numérico seja anexado a chave direita para indicar um limite máximo para o número de vezes que a parte contida nas chaves pode ser repetida. Além disso, algumas versões usam um sinal de mais (+) como sobreescrito para indicar uma ou mais repetições. Por exemplo,

$\langle \text{composto} \rangle \rightarrow \text{begin } \langle \text{inst} \rangle \ \{\langle \text{inst} \rangle\} \ \text{end}$

e

$\langle \text{composto} \rangle \rightarrow \text{begin } \{ \langle \text{inst} \rangle \}^+ \ \text{end}$

são equivalentes.

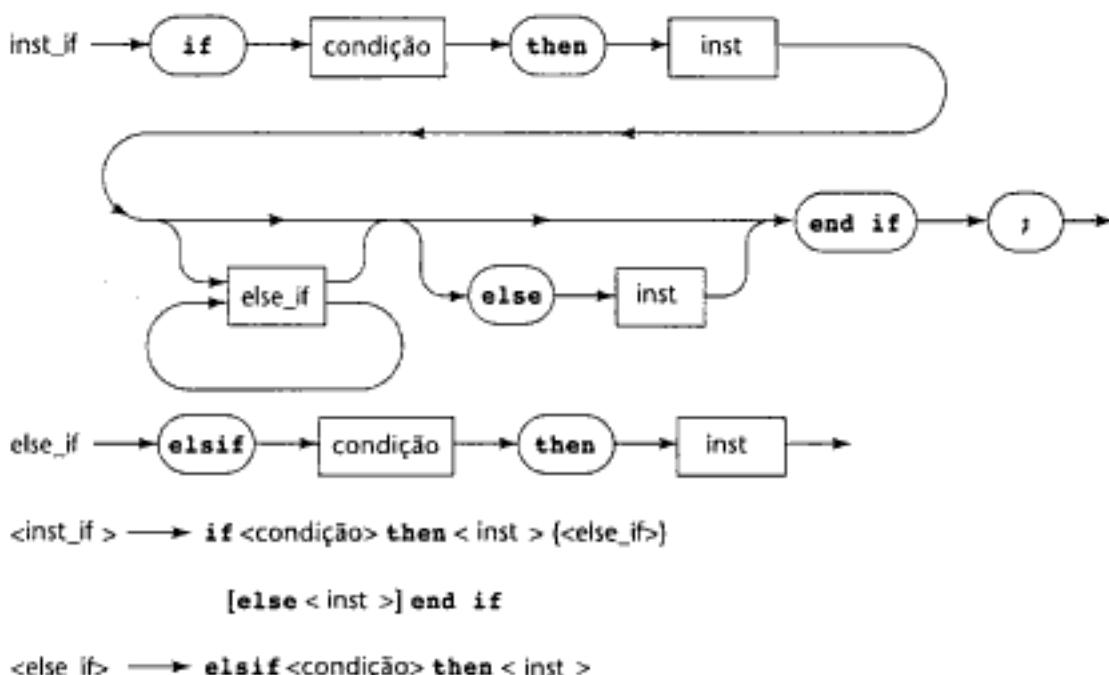
### 3.3.3 Grafos de Sintaxe

Um **grafo** é um conjunto de vértices, alguns dos quais ligados por meio de linhas chamadas arestas. Um **grafo dirigido** é aquele em que as arestas são direcionais; ou seja, elas têm pontas de flecha em uma extremidade para indicar a direção. Uma árvore de análise é uma forma restrita de grafo dirigido.

As informações nas regras BNF e EBNF podem ser representadas em grafos dirigidos, chamados **grafos de sintaxe**, diagramas de sintaxe ou gráficos de sintaxe. Um grafo distinto é usado para cada unidade sintática, da mesma maneira que um símbolo não-terminal em uma gramática representa essa unidade.

Os grafos de sintaxe usam diferentes tipos de vértices para representar os símbolos terminais e os não-terminais dos lados direitos das regras de uma gramática. Os vértices retangulares contêm os nomes das unidades sintáticas (não-terminais). Os círculos ou elipses contêm símbolos terminais.

A sintaxe da instrução Ada **if** é descrita tanto em EBNF como com um grafo de sintaxe na Figura 3.6. Note que tanto os colchetes como as chaves na descrição EBNF são metassímbolos ao invés dos símbolos terminais usados na Ada.



**FIGURA 3.6** O grafo de sintaxe e descrições EBNF da instrução Ada **if**.

Usar gráficos para descrever a sintaxe oferece a mesma vantagem que usá-los para descrever qualquer coisa: aumenta a legibilidade ao permitir-nos visualizá-la em duas dimensões.

### 3.3.4 Gramáticas e Reconhecedores

Anteriormente neste capítulo, sugerimos que havia uma estreita relação entre dispositivos de geração e de reconhecimento para determinada linguagem. De fato, dada uma gramática livre de contexto, um reconhecedor para a linguagem gerada por ela pode ser algorítmicamente construído. Desenvolveu-se um grande número de sistemas que executam essa construção, permitindo uma rápida criação da parte da análise sintática de um compilador para uma nova linguagem e isso é muito valioso. Um dos mais usados entre esses geradores de analisadores sintáticos é o chamado *yacc* (*yet another compiler-compiler*) (Johnson, 1975).

### 3.4 Gramáticas de Atributos

Uma *gramática de atributos* é um dispositivo usado para descrever mais detalhes da estrutura de uma linguagem de programação do que é possível com uma gramática livre de contexto. A extensão permite que certas regras de linguagem sejam descritas, como, por exemplo, a compatibilidade de tipos. Antes de definirmos formalmente a forma das gramáticas de atributos, devemos esclarecer o conceito da semântica estática.

#### 3.4.1 Semântica Estática

Existem algumas características da estrutura das linguagens de programação difíceis de descrever com a BNF, e algumas impossíveis. Como exemplo de uma regra de linguagem com essas características, considere as regras de compatibilidade de tipos. Em Java, por exemplo, um valor real não pode ser atribuído a uma variável de tipo inteiro, ainda que o oposto seja legal. Mesmo que essa restrição possa ser especificada em BNF, ela exige símbolos não-terminais e regras adicionais. Se todas as regras de tipificação do Java forem especificadas em BNF, a gramática iria tornar-se grande demais para ser útil porque o seu tamanho determina a extensão do analisador.

Como um exemplo de uma regra de linguagem que não pode ser especificada em BNF, considere a regra comum segundo a qual todas as variáveis devem ser declaradas antes de serem referenciadas. Pode-se provar que essa regra não pode ser especificada em BNF. Outro exemplo é a regra segundo a qual o `end` de um subprograma Ada seguido de um nome deve coincidir com o nome do subprograma.

Esses dois problemas exemplificam a categoria das regras de linguagem chamada **semântica estática** a qual se relaciona apenas indiretamente com o significado dos programas durante a execução; em vez disso, ela tem a ver com as formas legais dos programas (sintaxe no lugar de semântica). Em muitos casos, as regras de semântica estática declaram suas restrições de tipo. Ela é assim chamada porque a análise necessária para verificar essas especificações pode ser feita na compilação.

Por causa dos problemas de descrever a semântica estática com a BNF, uma variedade de mecanismos mais poderosos foi idealizada para cumprir essa tarefa. Entre eles está a gramática de atributos, projetada por Knuth (1968a) para descrever tanto a sintaxe como a semântica estática dos programas.

A semântica dinâmica será discutida na Seção 3.5.

#### 3.4.2 Conceitos Básicos

Gramáticas de atributos são gramáticas com a adição de atributos, funções de computação de atributos e funções predicadas. Os **atributos**, associados a símbolos gramaticais, são semelhantes a variáveis na medida em que podem ter valores atribuídos a eles. As **funções de computação de atributos**, às vezes chamadas de funções semânticas, são associadas a regras gramaticais para especificar como os valores de atributos são computados. As **funções predicadas**, que declaram a parte da sintaxe e as regras semânticas estáticas da linguagem, estão associadas a regras gramaticais.

Esses conceitos ficarão mais claros depois que definirmos formalmente as gramáticas de atributos e apresentarmos um exemplo.

### 3.4.3 Definição de Gramáticas de Atributos

Uma gramática de atributos é uma gramática com os seguintes recursos adicionais:

- Um conjunto de atributos  $A(X)$  está associado a cada símbolo gramatical  $X$ . O conjunto  $A(X)$  consiste de dois conjuntos disjuntos  $S(X)$  e  $I(X)$ , denominados atributos sintetizados e herdados, respectivamente. Os **atributos sintetizados** são usados para passar informações semânticas acima em uma árvore de análise, enquanto os **atributos herdados** passam informações semânticas abaixo na árvore de análise.
- Um conjunto de funções semânticas e um conjunto possivelmente vazio de funções predicadas sobre os atributos dos símbolos da regra gramatical estão associados a cada uma destas. Para a regra  $X_0 \rightarrow X_1 \dots X_n$ , os atributos sintetizados de  $X_0$  são computados com uma função semântica da forma  $S(X_0) = f(A(X_1), \dots, A(X_n))$ . Assim, o valor de um atributo sintetizado em um vértice da árvore de análise depende somente dos valores dos atributos dos vértices-filhos dele. Atributos herdados de símbolos  $X_j$ ,  $1 \leq j \leq n$  (na regra acima) são computados com uma função semântica da forma  $I(X_j) = f(A(X_0), \dots, A(X_n))$ . Assim, o valor de um atributo herdado em um vértice da árvore de análise depende dos valores de atributo do vértice-pai dele mesmo e dos valores de seus irmãos. Note que, para evitar a circularidade, os atributos herdados freqüentemente se restringem a funções da forma  $I(X_j) = f(A(X_0), \dots, A(X_{j-1}))$ , o que impede a um atributo herdado depender de si mesmo ou dos atributos à direita na árvore de análise.

Uma função predicada tem a forma da expressão booleana no conjunto de atributos  $\{A(X_0), \dots, A(X_n)\}$ . As únicas derivações permitidas com uma gramática de atributos são aquelas em que todo predicado associado a cada não-terminal é verdadeiro. Uma função predicada falsa indica uma violação das regras de sintaxe ou de semântica estática da linguagem.

Uma árvore de análise de uma gramática de atributos é aquela baseada em sua gramática BNF subjacente, com um conjunto possivelmente vazio de valores de atributo anexado a cada vértice. Se todos os valores de atributo em uma árvore de análise tiverem sido computados, diz-se que ela é **totalmente atribuída**. Ainda que, na prática, nem sempre aconteça dessa maneira, é conveniente imaginar os valores de atributo como sendo computados depois que a árvore de análise não-atribuída completa tiver sido construída.

### 3.4.4 Atributos Intrínsecos

**Atributos intrínsecos** são atributos sintetizados de vértices-folha cujos valores determinaram-se fora da árvore de análise. 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 determinado a partir das instruções de declaração anteriores. Inicialmente, presumindo-se que uma árvore de análise não-atribuída tenha sido construída e que valores de atributo são desejados, os únicos destes com valores são os intrínsecos dos vértices-folha. Dados os valores de atributo intrínsecos em uma árvore de análise, as funções semânticas podem ser usadas para computar aqueles restantes.

### 3.4.5 Exemplos de Gramáticas de Atributos

O seguinte fragmento de uma gramática de atributos que descreve a regra segundo a qual o nome no **end** de um procedimento Ada deve coincidir com o nome do procedimento pode ser considerado um exemplo muito simples de como se pode usar esta gramática para descrever a semântica estática. O atributo de cadeia de **<nome\_do\_proc>**, denotado por **<nome\_do\_proc>.string**, é a cadeia de caracteres real que foi encontrada pelo analisador léxico. Note que quando há mais de uma ocorrência de um não-terminal em uma regra de sintaxe de uma gramática de atributos, eles são subscritos com colchetes para distingui-los. Nenhum destes últimos faz parte da linguagem descrita.

Regra de sintaxe: **<def\_proc> → procedure <nome\_do\_proc>[1]**  
**<corpo\_do\_proc> end <nome\_do\_proc>[2];**  
 Regra semântica: **<nome\_do\_proc>[1].string = <nome\_do\_proc>[2].string**

Em seguida, considere um exemplo mais amplo de uma gramática de atributos. Nesse caso, ele é usado para mostrar como esta pode ser usada para verificar as regras de tipo de uma instrução de atribuição simples. A sintaxe e a semântica desta são as seguintes: os únicos nomes de variáveis são **A**, **B** e **C**. O lado direito das atribuições pode ou ser uma variável, ou uma expressão na forma de uma variável adicionada a outra. As variáveis podem ser de um dos dois tipos: **int** ou **real**. Quando elas estão no lado direito de uma atribuição, não precisam ser do mesmo tipo. O tipo da expressão, quando os tipos dos operandos não são os mesmos, sempre é **real**. Quando elas são os mesmos, o tipo da expressão é o dos operandos. O tipo do lado esquerdo da atribuição deve coincidir com o do lado direito. Assim, os tipos de operandos no lado direito podem ser mistos, mas a atribuição é válida somente se o LE e o valor resultante da avaliação do LD tiverem o mesmo tipo. A gramática de atributos especifica essas regras semânticas.

A parte sintática de nosso exemplo de gramática de atributos é

$$\begin{aligned} \text{atribuição} &\rightarrow \text{var} = \text{expr} \\ \text{expr} &\rightarrow \text{var} + \text{var} \\ &\quad | \text{var} \\ \text{var} &\rightarrow \text{A} \mid \text{B} \mid \text{C} \end{aligned}$$

Os atributos para os não-terminais no exemplo de gramática serão descritos nos próximos parágrafos.

**tipo\_efetivo** Um atributo sintetizado associado aos não-terminais **var** e **expr**. Ele é usado para armazenar o tipo efetivo, **int** ou **real** no exemplo, de uma variável ou de expressão. No caso de uma variável, o tipo efetivo é intrínseco. No caso de uma expressão, ele é determinado a partir dos tipos reais do vértice-filho ou dos vértices-filhos do não-terminal **expr**.

**tipo Esperado** Um atributo herdado associado ao não-terminal **expr**. Ele é usado para armazenar o tipo, **int** ou **real**, esperado para a expressão, conforme determinado pelo tipo da variável no lado esquerdo da instrução de atribuição.

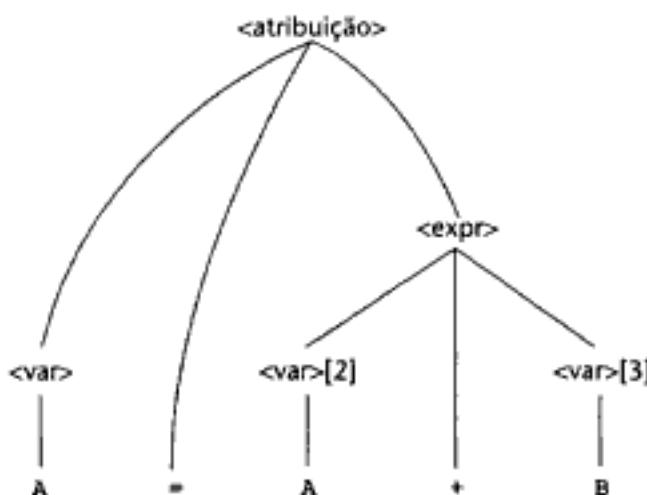
Segue-se a gramática de atributos completa no Exemplo 3.6.

**EXEMPLO 3.6 Uma Gramática de Atributos para Instruções de Atribuição Simples**

1. Regra de sintaxe:  $\langle \text{atribuição} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$   
 Regra semântica:  $\langle \text{expr} \rangle.\text{tipo\_esperado} \leftarrow \langle \text{var} \rangle.\text{tipo\_efetivo}$
2. Regra de sintaxe:  $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle[2] + \langle \text{var} \rangle[3]$   
 Regra semântica:  $\langle \text{expr} \rangle.\text{tipo\_efetivo} \leftarrow$   
 $\quad \text{if } (\langle \text{var} \rangle[2].\text{tipo\_efetivo} = \text{int}) \text{ e}$   
 $\quad (\langle \text{var} \rangle[3].\text{tipo\_efetivo} = \text{int})$   
 $\quad \text{then int}$   
 $\quad \text{else real}$   
 $\quad \text{end if}$   
 Predicado:  $\langle \text{expr} \rangle.\text{tipo\_efetivo} = \langle \text{expr} \rangle.\text{tipo\_esperado}$
3. Regra de sintaxe:  $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle$   
 Regra semântica:  $\langle \text{expr} \rangle.\text{tipo\_efetivo} \leftarrow \langle \text{var} \rangle.\text{tipo\_efetivo}$   
 Predicado:  $\langle \text{expr} \rangle.\text{tipo\_efetivo} = \langle \text{expr} \rangle.\text{tipo\_esperado}$
4. Regra de sintaxe:  $\langle \text{var} \rangle \rightarrow \text{A} \mid \text{B} \mid \text{C}$   
 Regra semântica:  $\langle \text{var} \rangle.\text{tipo\_efetivo} \leftarrow \text{look-up}(\langle \text{var} \rangle.\text{string})$

A função *look-up* pesquisa determinado nome de variável na tabela de símbolos e retorna o tipo desta.

Um exemplo de uma árvore de análise da sentença  $\text{A} = \text{A} + \text{B}$  gerada pela gramática do Exemplo 3.6 é mostrada na Figura 3.7. Como na gramática, os números entre colchetes são adicionados depois dos rótulos de vértice repetidos na árvore de modo que possam ser referenciados de maneira não-ambígua.



**FIGURA 3.7** Uma árvore de análise para  $\text{A} = \text{A} + \text{B}$ .

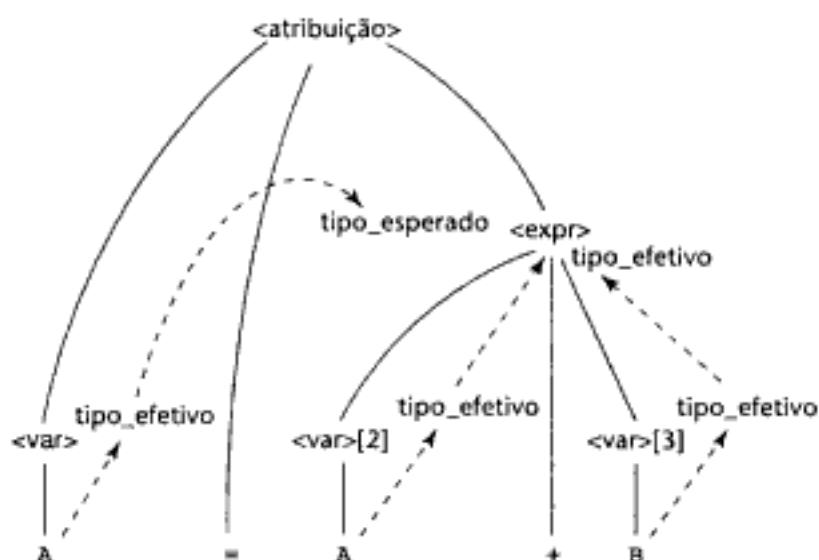
### 3.4.6 Computando Valores de Atributo

Considere, agora, o processo de representar os atributos na árvore de análise. Isso poderia ocorrer em uma ordem inteiramente cima-baixo, da raiz para as folhas, se todos os atributos fossem herdados. Alternativamente, poderia ocorrer em uma ordem inteiramente baixo-cima, das folhas para a raiz, se todos os atributos fossem sintetizados. Uma vez que nossa gramática tem tantos atributos sintetizados como herdados, o processo de avaliação não pode estar em qualquer direção particular. A seguir, apresentamos uma avaliação dos atributos, em uma ordem em que eles podem ser processados. Determinar esta ordem de avaliação para o caso geral de uma gramática é um problema complexo que exige a construção de um grafo de dependência para mostrar todas as dependências dos atributos.

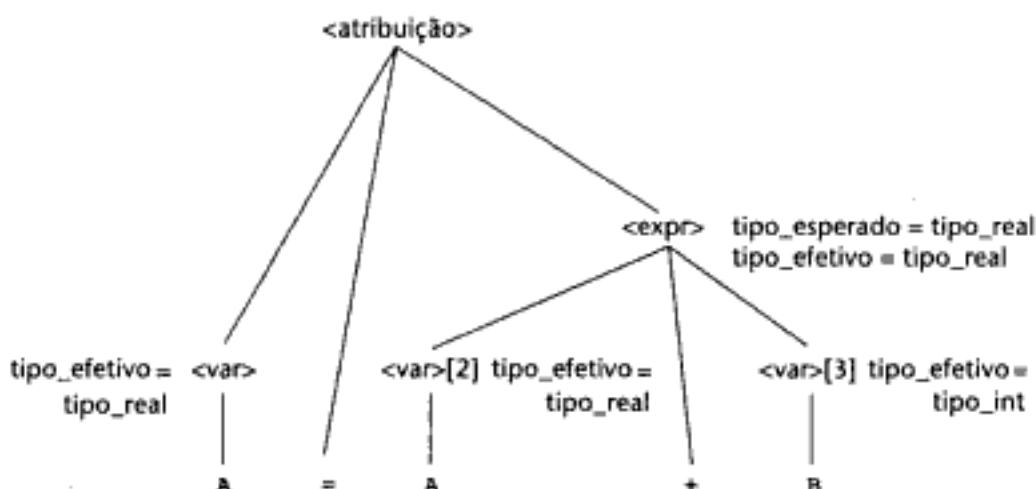
1.  $\langle \text{var} \rangle.\text{tipo\_efetivo} \leftarrow \text{look-up}(A)$  (Regra 4)
2.  $\langle \text{expr} \rangle.\text{tipo\_esperado} \leftarrow \langle \text{var} \rangle.\text{tipo\_efetivo}$  (Regra 1)
3.  $\langle \text{var} \rangle[2].\text{tipo\_efetivo} \leftarrow \text{look-up}(A)$  (Regra 4)
- $\langle \text{var} \rangle[3].\text{tipo\_efetivo} \leftarrow \text{look-up}(B)$  (Regra 4)
4.  $\langle \text{expr} \rangle.\text{tipo\_efetivo} \leftarrow \text{ou int ou real}$  (Regra 2)
5.  $\langle \text{expr} \rangle.\text{tipo\_esperado} = \langle \text{expr} \rangle.\text{tipo\_efetivo}$  é ou TRUE ou FALSE (Regra 2)

A árvore da Figura 3.8 mostra o fluxo dos valores de atributo do exemplo da Figura 3.7. As linhas sólidas são usadas para a árvore de análise e as linhas tracejadas mostram o fluxo dos atributos na árvore.

A árvore da Figura 3.9 (ver p. 131) mostra os valores de atributo finais nos vértices. Neste exemplo, A é definido como real e B como int.



**FIGURA 3.8** O fluxo dos atributos na árvore.



**FIGURA 3.9** Uma árvore de análise completamente atribuída.

### 3.4.7 Avaliação

As gramáticas de atributos têm sido usadas em uma ampla variedade de aplicações, entre elas: no fornecimento de descrições completas da sintaxe e da semântica estática de linguagens de programação (Watt, 1979); como a definição formal de uma linguagem que pode ser introduzida em um sistema de geração de compiladores (Farrow, 1982); como base de diversos sistemas de edição voltados para a sintaxe (Teitelbaum e Reps, 1981; Fischer *et al.*, 1984). Elas também têm sido usadas em sistemas de processamento de linguagens naturais (Correa, 1992).

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 sua complexidade. O grande número de atributos e de regras semânticas necessários para uma linguagem completa dificulta a leitura e a escrita dessas gramáticas. Além disso, os valores de atributo de uma árvore de análise grande são de avaliação custosa. Por outro lado, gramáticas de atributos menos formais constituem uma poderosa ferramenta comumente usada por escritores de compiladores mais interessados no processo de produção destes do que no formalismo.

## 3.5 Descrevendo o Significado dos Programas: Semântica Dinâmica

Vamos voltar-nos, agora, à difícil tarefa de descrever a **semântica dinâmica** ou o significado das expressões, das instruções e das unidades de programa. Por causa do poder e da naturalidade da notação disponível, descrever a sintaxe é uma questão relativamente simples. Por outro lado, nenhuma notação universalmente aceita foi idealizada para a semântica dinâmica. Nesta seção, descreveremos brevemente diversos métodos desenvolvidos. No restante, quando usarmos o termo *semântica*, estaremos referindo-nos à semântica dinâmica e a estática será referida como semântica estática.

Há diversas razões diferentes pelas quais alguém pode preocupar-se em descrever a semântica. Primeiro, os programadores evidentemente precisam saber precisamente o que as instruções de uma linguagem fazem. Mas, normalmente, eles descobrem isso lendo explicações em inglês em manuais. Tais explicações freqüentemente são imprecisas e incompletas. Os escritores de compiladores determinam a semântica da linguagem para a qual estão escrevendo compiladores a partir das descrições em inglês, usadas por causa da complexidade das descrições semânticas formais. Uma meta evidente de pesquisa é encontrar um formalismo semântico que possa ser usado por programadores e por escritores de compiladores. Trabalhos experimentais têm sido feitos na geração automática de compiladores para linguagens de programação diretamente de suas descrições semânticas. Finalmente, as provas de exatidão do programa recorrem a certa descrição formal da semântica da linguagem.

### 3.5.1 Semântica Operacional

A idéia existente por trás da **semântica operacional** é descrever o significado de um programa ao executar suas instruções em uma máquina, seja ela real ou simulada. As alterações que ocorrem no estado de uma máquina, quando ela executa determinada instrução, definem o significado desta. Para entender o conceito, considere uma instrução em linguagem de máquina. Digamos que o estado de um computador sejam os valores de todos os seus registradores e de suas localizações de memória, inclusive códigos de condição e registros de status. Ao se registrar o estado do computador, executar uma determinada instrução e depois examinar o novo estado da máquina, a semântica será clara: ela é representada pela mudança no estado do computador, causada pela execução da instrução.

#### 3.5.1.1 O Processo Básico

A descrição da semântica operacional de instruções das linguagens de alto nível exige a construção de um computador real ou de um virtual. Lembre-se do Capítulo 1, em que o hardware de um computador era definido como um interpretador puro de sua linguagem de máquina. Um interpretador puro para qualquer linguagem de programação pode ser construído em software, o qual se torna um computador virtual para a linguagem. A semântica de uma linguagem de alto nível pode ser descrita usando-se um interpretador puro para a linguagem. Entretanto, há dois problemas com tal abordagem. Primeiro, as complexidades e as idiossincrasias do hardware e do sistema operacional usados para rodar o interpretador puro poderiam dificultar o entendimento das ações. Em segundo lugar, uma definição semântica feita dessa maneira somente estaria disponível àquelas pessoas com um computador identicamente configurado.

Esses problemas podem ser evitados substituindo-se o computador real por um computador virtual de baixo nível, implementado como uma simulação de software. Os registradores, a memória, as informações de status e o processo de execução seriam todos simulados. O conjunto de instruções seria projetado de tal forma que fossem facilitados o entendimento e a descrição da semântica de cada instrução. Assim, a máquina seria idealizada e, portanto, altamente simplificada, facilitando o entendimento de suas mudanças de estado.

O uso do método operacional para descrever completamente a semântica de uma linguagem de programação L requer a construção de dois componentes. Primeiro, é necessário um tradutor para converter as instruções existentes em L para a linguagem de baixo nível escolhida. O outro componente é a máquina virtual. As mudanças de estado nesta,

levadas a efeito pela execução do código que resulta da tradução de determinada instrução na linguagem de alto nível, definem o seu significado.

Esse processo básico das semânticas operacionais não é incomum. De fato, o conceito é freqüentemente usado nos livros didáticos de programação e nos manuais de consulta de linguagens. Por exemplo, a semântica da construção C **for** pode ser descrita em termos de instruções muito simples, como em:

Instrução C	Semântica Operacional
<b>for</b> (expr1; expr2; expr3) {	expr1;
...	<b>loop</b> : if expr2 = 0 <b>goto</b> out
}	...
	expr3;
	<b>goto</b> loop
	out:...

O leitor humano dessa descrição é o computador virtual e presume-se que ele seja capaz de "executar" corretamente as instruções na definição e de reconhecer os efeitos da "execução".

Como exemplo de uma linguagem de baixo nível que poderia ser usada para semântica operacional, considere a seguinte lista de instruções, adequadas para as 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
```

Nessa lista, **relop** é um dos operadores relacionais do conjunto { =, <, >, <, >=, <= }, **ident** é um identificador; e **var** é ou um identificador, ou uma constante. Tais instruções são todas simples e, dessa forma, fáceis de entender e de implementar.

Uma ligeira generalização das três instruções de atribuição acima permite que expressões aritméticas e instruções de atribuição mais gerais sejam descritas. As novas instruções são mostradas a seguir.

```
ident = var op_bin var
ident = op_un var
```

em que **op\_bin** é um operador aritmético binário e **op\_un** é um operador unário. Tipos de dados aritméticos múltiplos e conversões de tipo automáticas, obviamente, complicam bastante essa generalização. Acrescentar apenas mais algumas instruções relativamente simples permitiria que a semântica de matrizes, de registros, de ponteiros e de subprogramas fosse descrita.

No Capítulo 8, a semântica de várias instruções de controle será descrita usando-se a semântica operacional.

### 3.5.1.2 Avaliação

O primeiro e mais significativo uso da semântica operacional formal descreve a semântica da PL/I (Wegner, 1972). Essa máquina abstrata particular e as regras de tradução para a PL/I receberam juntas o nome de Vienna Definition Language (VDL), em homenagem à cidade onde ela foi idealizada pela IBM.

A semântica operacional constitui um meio efetivo de descrever a semântica para usuários e para implementadores da linguagem, contanto que as descrições mantenham-se simples e informais. A descrição VDL da PL/I, infelizmente, é tão complexa que não serve virtualmente a qualquer propósito prático.

A semântica operacional depende de algoritmos, não da matemática. As instruções de uma linguagem de programação são descritas em termos das instruções de uma de nível mais baixo. Essa abordagem pode levar a circularidades, em que os conceitos são indiretamente definidos em termos de si mesmos. Os métodos descritos nas duas seções seguintes são muito mais formais, por basearem-se na lógica e na matemática, não em máquinas.

### 3.5.2 Semântica Axiomática

A **semântica axiomática** foi definida em conjunto com o desenvolvimento de um método para provar a exatidão dos programas que mostra a computação descrita por sua especificação, quando pode ser construída. Em uma prova, cada instrução de um programa tanto é precedida como seguida de uma expressão lógica que especifica restrições a variáveis. Estas, em vez do estado inteiro de uma máquina abstrata (como acontece com a semântica operacional), são usadas para especificar o significado da instrução. A notação usada para descrever restrições, na verdade a linguagem da semântica axiomática, é o cálculo de predicado. Ainda que expressões booleanas simples sejam freqüentemente adequadas para expressar restrições, em alguns casos não o são.

#### 3.5.2.1 Asserções

A semântica axiomática baseia-se na lógica matemática. As expressões lógicas são chamadas **predicados** ou **asserções**. Uma asserção que precede imediatamente uma instrução de programa descreve as restrições às variáveis dele nesse ponto. Uma asserção que se segue imediatamente a uma instrução descreve as novas restrições a essas variáveis (e possivelmente a outras) depois da execução da instrução. Essas asserções são denominadas **pré-condição** e **pós-condição**, respectivamente, da instrução. O desenvolvimento de uma descrição axiomática ou prova de determinado programa exige que toda instrução tenha uma pré-condição e uma pós-condição.

Nas seções seguintes, examinaremos as asserções partindo do ponto de vista de que as pré-condições para instruções são computadas a partir de determinadas pós-condições, embora seja possível considerarmos o sentido oposto. Presumimos que todas as variáveis são do tipo inteiro:

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

As asserções de pré-condição e de pós-condição são apresentadas entre chaves para distinguí-las das instruções de programa. Uma pré-condição possível para essa instrução é  $\{x > 10\}$ .

#### 3.5.2.2 As Pré-Condições Mais Fracas

A **pré-condição mais fraca** é a menos restritiva que garantirá a validade da pós-condição associada. Por exemplo, na instrução e na pós-condição acima,  $\{x > 10\}$ ,  $\{x > 50\}$  e  $\{x > 1000\}$  são todas pré-condições válidas. A mais fraca de todas as pré-condições, neste caso, é  $\{x > 0\}$ .

Se a pré-condição mais fraca puder ser computada a partir da pós-condição dada para cada instrução de uma linguagem, provas de exatidão podem ser construídas para os programas. A prova é iniciada usando-se como pós-condição da última instrução os resultados desejados da execução do programa e trabalhando-se para trás, computando-se as pré-condições mais fracas para cada instrução até que o início do programa seja alcançado. Nesse ponto, a primeira pré-condição declara as condições sob as quais se computará os resultados desejados.

Para algumas instruções de programa, a computação de uma pré-condição mais fraca a partir da instrução e de uma pós-condição é simples e pode ser especificada por meio de um axioma. Na maioria dos casos, entretanto, a pré-condição mais fraca pode ser computada somente por meio de uma regra de inferência. Um **axioma** é uma afirmação lógica que se presume verdadeira; uma **regra de inferência** é um método de suposição da verdade de uma asserção, baseando-se nos valores de outras.

Para usar a semântica axiomática com determinada linguagem de programação, quer seja para provas de exatidão como para especificações de semântica formal, um axioma ou uma regra de inferência deve ser definido para cada tipo de instrução na linguagem. Nas subseções seguintes, apresentaremos um axioma para instruções de atribuição e regras de inferência para sequências de instruções, instruções de seleção e laços de pré-teste lógicos.

### 3.5.2.3 Instruções de Atribuição

Digamos que  $x = E$  seja uma instrução de atribuição geral, e  $Q$  sua pós-condição. Então, sua pré-condição,  $P$ , é definida pelo axioma

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

significando que  $P$  é computado como  $Q$  com todas as instâncias de  $x$  substituídas por  $E$ . Por exemplo, se tivermos a instrução de atribuição e a pós-condição

$$a = b / 2 - 1 \{a < 10\}$$

a pré-condição mais fraca será computada substituindo-se  $b / 2 - 1$  na asserção  $\{a < 10\}$ , da seguinte maneira:

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

Assim, a pré-condição mais fraca para a atribuição e para a pós-condição dadas é  $\{b < 22\}$ . Note que o axioma de atribuição tem a garantia de ser verdadeiro somente na ausência de efeitos colaterais. Uma instrução de atribuição tem um efeito colateral se mudar alguma variável que não de seu lado esquerdo.

A notação usual para especificar a semântica axiomática de determinada forma de instrução é

$$\{P\} S \{Q\}$$

em que  $P$  é a pré-condição,  $Q$  é a pós-condição e  $S$  é a forma da instrução. No caso da instrução de atribuição, a notação é

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

Como outro exemplo de computação de uma pré-condição para uma instrução de atribuição, considere o seguinte:

$$x = 2 * y - 3 \{x > 25\}$$

A pré-condição é computada da seguinte maneira:

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

Então,  $\{y > 14\}$  é a pré-condição mais fraca para essa instrução de atribuição e de pós-condição.

Note que a ocorrência do lado esquerdo da instrução de atribuição em seu lado direito não afeta o processo de computação da pré-condição mais fraca. Por exemplo, para

$$x = x + y - 3 \quad \{x > 10\}$$

a pré-condição mais fraca é

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

No início de nossa discussão, estabelecemos que a semântica axiomática foi desenvolvida para provar a exatidão de programas. Levando isso em conta, é natural imaginarmos, a esta altura, como o axioma para instruções de atribuição pode ser usado para provar alguma coisa. Eis como: determinada instrução de atribuição que possui tanto uma pré-condição como uma pós-condição pode ser considerada um teorema. Se o axioma de atribuição, quando aplicado à pós-condição e à instrução de atribuição, produzir a pré-condição dada, o teorema estará demonstrado. Por exemplo, considere a instrução 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. Portanto, provamos a instrução lógica acima.

Em seguida, considere a instrução lógica

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

Nesse caso, a pré-condição dada,  $\{x > 5\}$ , não é a mesma coisa que a asserção produzida pelo axioma. Porém, é evidente que  $\{x > 5\} \Rightarrow \{x > 3\}$ . Para usarmos isso em uma prova, precisaremos de uma regra de inferência, chamada **regra de consequência**. A forma geral de uma regra de inferência é

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

a qual estabelece que se  $S_1, S_2, \dots$  e  $S_n$  forem verdadeiros, a verdade  $S$  poderá ser inferida.

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 instrução de programa. A regra pode ser enunciada da seguinte maneira: se a instrução lógica  $\{P\} S \{Q\}$  for verdadeira e a asserção  $P'$  implica a asserção  $P$  e a asserção  $Q$  implica a asserção  $Q'$ , pode-se inferir que  $\{P'\} S \{Q'\}$ . Em linguagem mais simples, a regra de consequência diz que a pós-condição sempre pode ser enfraquecida e uma pré-condição sempre pode ser fortalecida.

Isso é bastante útil em provas de programa. Por exemplo, permite a conclusão da prova do último exemplo de instrução lógica acima. Se admitirmos que P é  $\{x > 3\}$ , que Q e Q' são  $\{x > 0\}$  e P' é  $\{x > 5\}$ , teremos

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

Isso conclui a prova.

### 3.5.2.4 Seqüências

A pré-condição mais fraca de uma seqüência de instruções não pode ser descrita por meio de um axioma porque ela depende dos tipos particulares de instruções da seqüência. Nesse caso, a pré-condição somente pode ser descrita com uma regra de inferência. Admitamos que S1 e S2 sejam instruções de programa adjacentes. Se S1 e S2 tiverem as seguintes pré e pós-condições

$$\begin{aligned} \{P_1\} & S1 \{P_2\} \\ \{P_2\} & S2 \{P_3\} \end{aligned}$$

a regra de inferência para esta seqüência de duas instruções será

$$\frac{\{P_1\} \quad S1 \{P_2\}, \quad \{P_2\} \quad S2 \{P_3\}}{\{P_1\} \quad S1; S2 \{P_3\}}$$

Assim, para o exemplo acima,  $\{P_1\} \quad S1; S2 \{P_3\}$  descreve a semântica axiomática da seqüência S1; S2. Se S1 e S2 forem as instruções de atribuição

$$x1 = E1$$

e

$$x2 = E2$$

teremos:

$$\begin{aligned} \{P_{x2 \rightarrow E2}\} \quad x2 &= E2 \{P_3\} \\ \{((P_{x2 \rightarrow E2})_{x1 \rightarrow E1}) \quad x1 &= E1 \{P_{x2 \rightarrow E2}\} \end{aligned}$$

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

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

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

A pré-condição para a última instrução de atribuição é

$$y < 7$$

Essa será, então, a pós-condição para a primeira. A pré-condição para a primeira instrução de atribuição agora poderá ser computada:

$$\begin{aligned} 3 * x + 1 &< 7 \\ x &< 2 \end{aligned}$$

### 3.5.2.5 Seleção

Consideraremos em seguida a regra de inferência para instruções de seleção. Consideraremos somente as seleções que incluem cláusulas **else**. A regra de inferência será

$$\{B \text{ and } P\} S1 \{Q\}, \{(not B) \text{ and } P\} S2 \{Q\}$$

$$\frac{\{P\} \text{ if } B \text{ then } S1 \text{ else } S2 \{Q\}}{}.$$

Essa regra indica que as instruções de seleção devem ser provadas para ambos os casos. A primeira instrução lógica acima da linha é a cláusula **then**; a segunda é a cláusula **else**.

Considere o seguinte exemplo de computação usando a regra de inferência de seleção. O exemplo de instrução de seleção é

```
if (x > 0)
    y = y - 1
else y = y + 1
```

Suponhamos que a pós-condição para essa instrução de seleção seja  $\{y > 0\}$ . Podemos usar o axioma para atribuição na cláusula **then**

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

Isso produzirá  $\{y - 1 > 0\}$  ou  $\{y > 1\}$ . Aplicaremos, agora, o mesmo axioma à cláusula **else**

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

Isso produzirá a pré-condição  $\{y + 1 > 0\}$  ou  $\{y > -1\}$ . Uma vez que  $\{y > 1\} \Rightarrow \{y > -1\}$ , a regra de consequência nos permite usar  $\{y > 1\}$  para a pré-condição da instrução de seleção.

### 3.5.2.6 Laços de Pré-Teste Lógico

Outra construção fundamental de uma linguagem de programação imperativa é o laço pré-teste lógico, ou laço **while**. Computar a pré-condição mais fraca para um laço **while** é inherentemente mais difícil do que uma sequência, porque o número de iterações não pode ser previamente determinado em todos os casos. No caso em que o número de iterações é conhecido, o laço pode ser tratado como uma sequência.

O problema de computar a pré-condição mais fraca para laços é similar ao problema de provar um teorema sobre todos os números inteiros positivos. No último caso, normalmente usa-se a indução, e o mesmo método induutivo pode ser usado para os 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 **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

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

em que  $I$  é a invariante de laço.

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

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

A invariante de laço deve satisfazer uma série de exigências 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 após a finalização do laço. Essas restrições levam-nos 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 que controla o laço e pelas instruções do corpo do laço. Daí, o nome invariante.

Outro fator complicador para os laços **while** é a questão da finalização de laços. Se Q for uma pós-condição satisfeita imediatamente após a saída do laço, a pré-condição P para o laço será aquela que garante Q na saída do laço e também garante que o laço irá se encerrar.

A descrição axiomática completa de uma construção **while** exige que tudo que se segue seja verdadeiro, sendo I a invariante de laço:

$$\begin{aligned} P \Rightarrow I \\ \{I\} B \{I\} \\ \{I \text{ and } B\} S \{I\} \\ \{I \text{ and } (\text{not } B)\} \Rightarrow Q \\ \text{o laço encerra-se} \end{aligned}$$

Para encontrar uma invariante de laço, podemos usar um método semelhante ao usado na determinação da hipótese indutiva na indução matemática, que é o seguinte: a relação para alguns casos é computada, com a esperança de que surja um padrão que se aplique ao caso geral. É útil tratarmos o processo de produção de uma pré-condição mais fraca como uma função, wp. Em geral

$$wp(\text{instrução}, \text{pós-condição}) = \text{pré-condição}$$

Para encontrar I, usamos a pós-condição de laço Q para computar pré-condições para diversos números de iterações do corpo do laço, iniciando com nenhum. Se o corpo do laço contiver uma única instrução de atribuição, o axioma para instruções de atribuição pode ser usado para computar esses casos. Considere o exemplo de laço

```
while y <> x do y = y + 1 end {y = x}
```

Tenha o cuidado de lembrar que o sinal de igualdade está sendo usado para dois diferentes propósitos aqui. Nas asserções, ele significa igualdade matemática; fora das asserções, significa o operador de atribuição.

Para zero iterações, a pré-condição mais fraca evidentemente é:

$$\{y = x\}$$

Para uma iteração é:

$$wp(y = y + 1, \{y = x\}) = \{y + 1 = x\}, \text{ ou } \{y = x - 1\}$$

Para duas iterações é:

$$wp(y = y + 1, \{y = x - 1\}) = \{y + 1 = x - 1\}, \text{ ou } \{y = x - 2\}$$

Para três iterações é:

$$wp(y = y + 1, \{y = x - 2\}) = \{y + 1 = x - 2\}, \text{ ou } \{y = x - 3\}$$

Agora está claro que  $\{y < x\}$  será suficiente para casos de uma ou de mais iterações. Combinando isso com  $\{y = x\}$  para o caso das zero iterações, obteremos  $\{y \leq x\}$ , o qual pode ser usado para a invariante de laço. Uma pré-condição para a instrução **while** pode ser determinada a partir da invariante de laço. Nesse exemplo,  $P = I$  pode ser usado.

Devemos assegurar que nossa escolha satisfaça os cinco critérios para I em nosso laço de exemplo. Primeiro, porque  $P = I$ ,  $P \Rightarrow I$ . A segunda exigência é que I não seja afetado pela avaliação da expressão booleana do laço, a qual é  $y <> x$ . A avaliação dessa expressão não muda nada, de modo que não pode afetar I. Em seguida, deve ser verdadeiro que

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

Em nosso exemplo, temos

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

Aplicando o axioma de atribuição a

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

obtemos  $\{y + 1 \leq x\}$ , que é equivalente a  $\{y < x\}$ , o que é consequência de  $\{y \leq x\}$  and  $y <> x$ . Assim, a instrução acima está provada.

A seguir, temos

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

Em nosso exemplo, temos

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

Então, isso é evidentemente verdadeiro. Em seguida, a finalização do laço deve ser considerada. Nesse exemplo, a questão é se o laço

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

encerra-se. Lembrando que se presume que x e y são variáveis inteiros, podemos ver facilmente que esse laço se encerra. A pré-condição garante que y inicialmente não seja maior do que x. O corpo do laço aumenta y a cada iteração até que y seja igual a x. Não importa o quanto y é menor do que x inicialmente; por fim, ele se tornará igual a x. Assim, o laço irá encerrar-se. Uma vez que nossa escolha de I satisfaz todos os cinco critérios, ele é adequado para a invariante de laço e para a pré-condição do laço.

O processo usado acima para computar a invariante para um laço nem sempre produz uma asserção que seja a pré-condição mais fraca (embora isso aconteça no exemplo anterior).

Como outro exemplo de descoberta de uma invariante de laço, considere a seguinte instrução de laço:

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

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

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

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

{ $s$  é uma potência não-negativa de 2}

Novamente, o I computado pode servir como P, e I é aprovado quanto às cinco exigências. Diferentemente de nosso exemplo anterior de como encontrar uma pré-condição de laço, esta evidentemente não é a pré-condição mais fraca. Considere usar a pré-condição  $\{s > 1\}$ . A instrução lógica

```
{s > 1} while s > 1 do s = s / 2 end {s = 1}
```

pode ser facilmente provada, e essa pré-condição é significativamente mais ampla do que a computada acima. O laço e a pré-condição são satisfeitos por qualquer valor positivo de  $s$ , não apenas por potências de 2, como o processo indica. Por causa da regra de consequência, usar uma pré-condição que seja mais forte do que a pré-condição mais fraca não invalida uma prova.

Encontrar invariantes de laços nem sempre é fácil. É útil entendermos a natureza dessas invariantes. Primeiro, esta é uma versão enfraquecida da pós-condição de laço e também uma pré-condição para ele. Assim, I deve ser fraco o bastante para ser satisfeito antes do início da execução do laço, mas quando combinado com a condição de saída deste, ele deverá ser forte o bastante para forçar a verdade da pós-condição.

Por causa da dificuldade de provar a finalização de laços, essa exigência freqüentemente é ignorada. Se a finalização do laço puder ser demonstrada, a descrição axiomática do laço será chamada de **exatidão total**. Se as outras condições puderem ser cumpridas, mas a finalização não estiver garantida, ela será chamada de **exatidão parcial**.

Em laços mais complexos, encontrar uma invariante de laço adequada, mesmo para exatidão parcial, exige bastante habilidade. Uma vez que a computação da pré-condição para um laço **while** depende de encontrar uma invariante de laço, provar a exatidão de programas com laços **while**, usando-se a semântica axiomática, pode ser difícil.

Apresentamos a seguir um exemplo de uma prova de exatidão de um programa em pseudocódigo que computa a função factorial.

```
{n >= 0}
cont = n;
fatorial = 1;
while cont <> 0 do
    fatorial = fatorial * cont;
    cont = cont - 1;
end
{fatorial = n!}
```

O método descrito anteriormente para encontrar a invariante de laço não funciona para o laço desse exemplo. É necessário um pouco de habilidade aqui, que pode ser ajudada com um breve estudo do código. O laço computa a função factorial na ordem da última multiplicação primeiro; ou seja,  $(n - 1) * n$  é feito primeiro, presumindo que  $n$  seja maior do que 1. Assim, parte da invariante pode ser

```
fatorial = (cont + 1) * (cont + 2) * ... * (n - 1) * n
```

Mas também devemos garantir que  $cont$  seja sempre não-negativo, o que podemos fazer adicionando isso à parte acima, para obtermos

```
I = (fatorial = (cont + 1) * ... * n) AND (cont >= 0)
```

Em seguida, devemos verificar se esse I satisfaz as exigências para invariantes. Novamente, admitimos que I também seja usado como P, de forma que P implique claramente I. A avaliação da expressão booleana da instrução **while**, `cont <> 0`, evidentemente não afeta I. A questão seguinte é

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

I and B é

```
((fatorial = (cont + 1) * ... * n) AND (cont >= 0)) AND
(cont <> 0)
```

que se reduz a

```
(fatorial = (cont + 1) * ... * n) AND (cont > 0)
```

Em nosso caso, devemos computar a pré-condição do corpo do laço, usando a invariante para a pós-condição. Para

$\{P\} cont = cont - 1 \{I\}$

computamos P como sendo

```
((fatorial = cont * (cont + 1) * ... * n) AND (cont >= 1))
```

Usando isso como a pós-condição para a primeira atribuição no corpo do laço,

```
{P} fatorial = fatorial * cont {{fatorial = cont * (cont + 1)
* ... * n) AND (cont >= 1)}
```

Neste caso, P é

```
((fatorial = (cont + 1) * ... * n) AND (cont >= 1))
```

É claro que I e B implicam este P; então, pela Regra de Consequência,

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

é verdadeiro. Finalmente, o último teste de I é

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

Para nosso exemplo, isto é

```
((fatorial = (cont + 1) * ... * n) AND (cont >= 0)) AND
(cont = 0)) => fatorial = n!
```

Isso, evidentemente, é verdadeiro, porque quando `cont = 0`, a primeira parte é precisamente a definição de fatorial. Assim, nossa escolha de I cumpre as exigências para uma invariante de laço. Agora podemos usar nosso P (que é o mesmo que I) do **while** como a pós-condição na segunda atribuição do programa.

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

que resulta para P

```
(1 = (cont + 1) * ... * n) AND (cont >= 0))
```

Usando isso como a pós-condição para a primeira atribuição no código

```
{P} cont = n {{(1 = (cont + 1) * ... * n) AND
(cont >= 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 inteiro,  $\{n \geq 0\}$ . Portanto, foi demonstrado que o programa está correto.

### 3.5.2.7 Avaliação

Para definir a semântica de uma linguagem de programação completa usando o método axiomático, um axioma ou uma regra de inferência deve ser usado para cada tipo de instrução. A definição de axiomas e de regras de inferência para algumas das instruções de linguagens de programação, como vimos, é uma tarefa difícil. Uma solução evidente para esse problema é projetar a linguagem tendo em mente o método axiomático, de modo que sejam incluídas somente instruções para as quais se pode escrever axiomas e regras de inferência. Infelizmente, essa linguagem seria bastante pequena e simples, dado o estado da ciência da semântica axiomática.

A semântica axiomática é uma ferramenta poderosa para pesquisar provas de exatidão de programas e constitui uma excelente estrutura sobre a qual argumentar a respeito dos programas, tanto durante como após a sua construção. Sua utilidade para descrever o significado das linguagens de programação para os seus usuários ou para escritores de compiladores é, entretanto, altamente limitada.

## 3.5.3 Semântica Denotacional

A **semântica denotacional** é o método mais rigoroso bastante conhecido para descrever o significado de programas. Ela baseia-se solidamente na teoria da função recursiva. Uma cuidadosa discussão do uso da semântica denotacional para descrever a semântica das linguagens de programação é longa e complexa. Nossa intenção é apresentar apenas o suficiente para conscientizar sobre como a semântica denotacional funciona.

O conceito fundamental de semântica denotacional é definir para cada entidade da linguagem tanto um objeto matemático como uma função que relate as instâncias daquela entidade com as deste. Uma vez que os objetos são rigorosamente definidos, eles representam o significado exato de suas entidades correspondentes. A idéia baseia-se no fato de que há maneiras rigorosas de manipular objetos matemáticos, mas não construções de linguagens de programação. A dificuldade com esse método está em criar os objetos e as funções de correspondência. O método é denominado denotacional porque os objetos matemáticos denotam o significado de suas entidades sintáticas correspondentes.

### 3.5.3.1 Dois Exemplos Simples

Usamos uma construção de linguagem muito simples, números binários, para apresentar o método denotacional. A sintaxe dos números binários pode ser descrita por meio das seguintes regras gramaticais:

```
<num_bin> → 0  
| 1  
| <num_bin> 0  
| <num_bin> 1
```

Uma árvore de análise para o exemplo de número binário, 110, é mostrada na Figura 3.10.

Para descrever o significado dos números binários usando a semântica denotacional e as regras gramaticais acima, o significado real é associado a cada regra com um único símbolo terminal como seu LD. Nesse caso, os objetos são simples números decimais. Então, neste exemplo, o significado de um número binário será o número decimal equivalente.

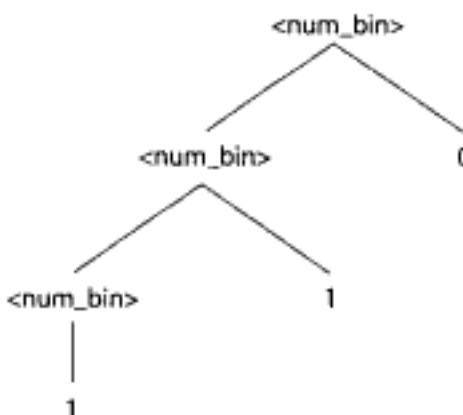
Em nosso exemplo, objetos significativos devem estar associados às duas primeiras regras gramaticais. As duas outras são, em certo sentido, regras computacionais, porque combinam um símbolo terminal, que pode ser associado a algum objeto, com um não-terminal, que possa representar alguma construção. Presumindo uma avaliação que progride para cima na árvore de análise, o não-terminal no lado direito já teria seu significado anexado. Assim, essa regra de sintaxe exigiria uma função que computasse o significado do LE, que deveria, então, representar o significado do LD completo.

Admitamos que o domínio de valores semânticos dos objetos seja  $N$ , o conjunto dos valores inteiros decimais não-negativos. Esses objetos é que desejamos associar aos números binários. A função semântica, chamada  $M_{bin}$ , relaciona os objetos sintáticos, descritas pelas regras gramaticais acima, com os objetos de  $N$ . A função  $M_{bin}$  é definida da seguinte maneira:

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

Note que colocamos apóstrofos nos dígitos sintáticos para mostrar que eles não são dígitos matemáticos. Isso é similar à relação entre dígitos codificados em ASCII e dígitos matemáticos. Quando um programa lê um número como uma cadeia, ele precisa ser convertido para um símbolo matemático antes que possa ser usado como um número no programa.

Os significados, ou os objetos denotados (que, neste caso, são números decimais), podem ser anexados aos vértices da árvore de análise acima, produzindo a árvore da Figura 3.11. Essa é a semântica dirigida para a sintaxe. As entidades sintáticas são associadas a objetos matemáticos com significado concreto.



**FIGURA 3.10** Uma árvore de análise do número binário 110.

Em parte porque precisaremos dele mais tarde, apresentaremos um exemplo semelhante para descrever o significado sintático dos literais decimais.

$$\begin{aligned} <\text{num\_dec}> \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ & \mid <\text{num\_dec}> (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9) \end{aligned}$$

As relações denotacionais para estas regras de sintaxe são:

$$\begin{aligned} M_{\text{dec}}('0') &= 0, M_{\text{dec}}('1') = 1, M_{\text{dec}}('2') = 2, \dots, M_{\text{dec}}('9') = 9 \\ M_{\text{dec}}(<\text{num\_dec}> '0') &= 10 * M_{\text{dec}}(<\text{num\_dec}>) \\ M_{\text{dec}}(<\text{num\_dec}> '1') &= 10 * M_{\text{dec}}(<\text{num\_dec}>) + 1 \\ \cdots \\ M_{\text{dec}}(<\text{num\_dec}> '9') &= 10 * M_{\text{dec}}(<\text{num\_dec}>) + 9 \end{aligned}$$

Nas seções seguintes, apresentaremos a semântica denotacional de algumas construções simples. A pressuposição simplificadora mais importante feita aqui é que tanto a sintaxe quanto a semântica estática das construções estão corretas. Além disso, presumimos que somente dois tipos escalares estão incluídos, o inteiro e o booleano.

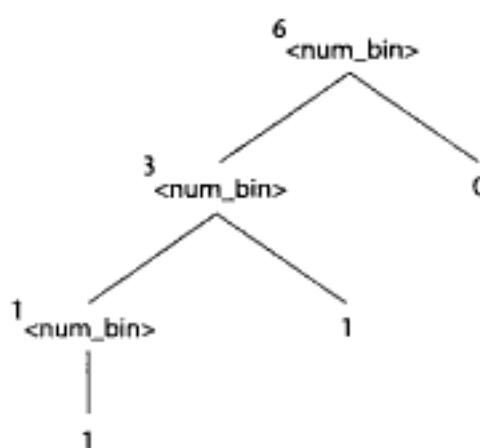
### 3.5.3.2 O Estado de um Programa

A semântica denotacional de um programa poderia ser definida, em termos das mudanças de estado, em um computador ideal. A semântica operacional é definida dessa maneira, e a semântica denotacional quase o é também. Em uma simplificação adicional, entretanto, elas são definidas somente em termos dos valores de todas as variáveis do programa. A diferença fundamental entre semântica operacional e semântica denotacional está no fato de que as mudanças de estado na primeira são definidas por algoritmos codificados, ao passo que na outra as mudanças de estado são definidas por funções matemáticas rigorosas.

Admitamos que o estado  $s$  de um programa seja representado como um conjunto de pares ordenados, da seguinte maneira:

$$\{<i_1, v_1>, <i_2, v_2>, \dots, <i_n, v_n>\}$$

Cada  $i$  é o nome de uma variável, e os  $v$ 's associados são os valores atuais dessas variáveis. Qualquer um dos  $v$ 's pode ter o valor especial **undef**, indicando que sua variável associada



**FIGURA 3.11** Uma árvore de análise com objetos denotados para 110.

está indefinida no momento. Admitamos que VARMAP seja uma função de dois parâmetros, um nome de variável e o estado do programa. O valor de  $VARMAP(i, s)$  é  $v_i$  (o valor emparelhado com  $i$  no estado  $s$ ). A maioria das funções de associação semântica para programas e construções de programa relaciona estados com estados. Tais mudanças de estado são usadas para definir o significado de programas e de construções de programas. Algumas construções de linguagem, como, por exemplo, expressões, são associadas a valores, não a estados.

### 3.5.3.3 Expressões

As expressões são fundamentais para a maioria das linguagens de programação. Presumimos, aqui, que as expressões não têm nenhum efeito colateral. Além disso, lidamos somente com expressões muito simples. Os únicos operadores são  $+$  e  $*$ , e a expressão pode ter um operador no máximo; os únicos operandos são variáveis escalares e literais inteiros; não há parêntese; e o valor de uma expressão é um número inteiro. A seguir, apresentamos uma descrição BNF dessas expressões:

```

<expr> → <num_dec> | <var> | <expr_binária>
<expr_binária> → <expr_esquerda> <operador> <expr_direita>
<operador> → + | *

```

O único erro que consideramos nas expressões é que uma variável tenha um valor indefinido. Evidentemente, outros erros podem ocorrer, mas a maioria deles depende da máquina. Admitamos que  $Z$  seja o conjunto dos números inteiros, e admitamos que **error** seja o valor de erro. Então  $Z \cup \{\text{error}\}$  é o conjunto de valores para os quais uma expressão pode ser avaliada.

Segue-se a função de correspondência necessária para uma expressão  $E$  e o estado  $s$  dados. Para fazermos a distinção entre as definições de funções matemáticas e as instruções de atribuição de linguagens de programação, usamos o símbolo  $\Delta =$  para definir as funções matemáticas.

```

M_e(<expr>, s) Δ=
  case <expr> of
    <num_dec> => M_{dec}(<num_dec>, s)
    <var> => if VARMAP(<var>, s) = undef
      then error
      else VARMAP(<var>, s)
    <expr_binária> =>
      if (M_e(<expr_binária>.<expr_esquerda>, s) = undef OR
          M_e(<expr_binária>.<expr_direita>, s) = undef)
      then error
      else if (<expr_binária>.<operador> = '+') then
        M_e(<expr_binária>.<expr_esquerda>, s) +
        M_e(<expr_binária>.<expr_direita>, s)
      else M_e(<expr_binária>.<expr_esquerda>, s) *
        M_e(<expr_binária>.<expr_direita>, s)

```

### 3.5.3.4 Instruções de Atribuição

Uma instrução de atribuição é a avaliação de uma expressão mais o ajuste da variável do lado esquerdo com o valor da expressão. Isso pode ser descrito com a seguinte função:

```

 $M_a(x = E, s) \Delta = \begin{cases} \text{if } M_e(E, s) = \text{error} \\ \quad \text{then error} \\ \quad \text{else } s' = \{\langle i_1', v_1' \rangle, \langle i_2', v_2' \rangle, \dots, \langle i_n', v_n' \rangle\}, \text{ onde} \\ \quad \quad \text{para } j = 1, 2, \dots, n, v_j' = \text{VARMAP}(i_j, s) \text{ se } i_j \neq x; \\ \quad \quad M_e(E, s) \text{ se } i_j = x \end{cases}$ 

```

Note que as duas comparações nas duas últimas linhas acima,  $i_j \neq x$  e  $i_j = x$ , são de nomes, não de valores.

### 3.5.3.5 Laços de Pré-Teste Lógico

A semântica denotacional de um laço lógico simples é enganosamente simples. Para agilizarmos a discussão, supomos que há duas outras funções de correspondência existentes,  $M_{s1}$  e  $M_b$ , que relacionam listas de instruções com estados, e expressões booleanas com valores booleanos (ou **error**), respectivamente. A função é

```

 $M_1(\text{while } B \text{ do } L, s) \Delta = \begin{cases} \text{if } M_b(B, s) = \text{undef} \\ \quad \text{then error} \\ \quad \text{else if } M_b(B, s) = \text{false} \\ \quad \quad \text{then } s \\ \quad \quad \text{else if } M_{s1}(L, s) = \text{error} \\ \quad \quad \quad \text{then error} \\ \quad \quad \quad \text{else } M_1(\text{while } B \text{ do } L, M_{s1}(L, s)) \end{cases}$ 

```

O significado do laço é simplesmente o valor das variáveis de programa depois que as instruções do laço tiverem sido executadas o número prescrito de vezes, presumindo que não tenha havido erros. Fundamentalmente, o laço foi convertido de iteração para recursão, cujo controle é matematicamente definido por outras funções de correspondência de estado recursivas. A recursão é mais fácil de ser descrita com rigor matemático do que a iteração.

Outra observação significativa nesse ponto é a que tal definição, à semelhança dos laços de programa reais, pode computar nada devido à não-finalização.

### 3.5.3.6 Avaliação

Objetos e funções, como os usados nas instruções acima, podem ser definidos para as outras entidades sintáticas das linguagens de programação. Quando um sistema completo tiver sido definido para determinada linguagem, ele poderá ser usado para estabelecer o significado de programas completos nessa linguagem. Isso constitui um arcabouço para pensarmos em programação de uma maneira altamente rigorosa.

A semântica denotacional pode ser usada como um auxílio para o projeto de linguagens. Por exemplo, instruções para as quais a descrição é complexa e difícil podem indicar ao projetista que também poderá haver dificuldades para que os usuários da linguagem a entendam e que, talvez, seja oportuno pensar em um projeto alternativo.

Uma quantidade significativa de trabalho tem sido feita em relação à possibilidade de usar-se descrições de linguagem denotacionais para gerar compiladores automaticamente (Jones, 1980; Milos et al., 1984; Bodwin et al., 1982). Esses esforços mostraram que o método é viável, mas o trabalho jamais progrediu até o ponto de poder ser usado para gerar compiladores úteis.

Devido à complexidade das descrições denotacionais, elas têm pouca utilidade para os usuários da linguagem. Por outro lado, oferecem um excelente método de descrever uma linguagem de maneira concisa.

Não obstante o uso da semântica denotacional normalmente ser atribuído a Scott e Strachey (1971), a abordagem denotacional geral à descrição das linguagens pode remeter-se ao século XIX (Frege, 1892).

## RESUMO

A forma de Backus-Naur e as gramáticas livres de contexto são metalinguagens equivalentes quase idealmente adequadas para a tarefa de descrever a sintaxe das linguagens de programação. Elas não somente são métodos descritivos concisos, mas a árvore de análise que pode ser associada às suas ações generativas fornece uma evidência gráfica das estruturas sintáticas subjacentes. Além disso, elas estão naturalmente relacionadas com os dispositivos de reconhecimento das linguagens que geram, o que possibilita uma construção relativamente fácil de analisadores sintáticos de compiladores para essas linguagens.

Os grafos de sintaxe são simplesmente representações gráficas de gramáticas.

Uma gramática de atributos é um formalismo descritivo que pode descrever tanto a sintaxe como a semântica estática de uma linguagem.

Existem três métodos principais de descrição da semântica: operacional, axiomático e denotacional. A semântica operacional é um método para descrever o significado de construções de linguagem em termos de seus efeitos sobre uma máquina ideal. A semântica axiomática, que se baseia na lógica formal, foi idealizada como uma ferramenta para provar a exatidão de programas. Na semântica denotacional, objetos matemáticos são usados para representar os significados das construções de linguagem. As entidades de linguagem são convertidas nesses objetos matemáticos com funções recursivas.

## NOTAS BIBLIOGRÁFICAS

A descrição sintática, usando gramáticas livres de contexto e a BNF, é cuidadosamente discutida em Claveland e Uzgalis (1976). Os grafos de sintaxe foram desenvolvidos na Burroughs em um projeto de desenvolvimento de compilador para serem usados como uma descrição compacta da sintaxe do ALGOL 60 (Taylor et al., 1961). Mais tarde, eles foram modificados por A. Schai, diretor do centro de computadores da ETH em Zurique. Essa versão modificada surgiu primeiro na forma impressa, em um livro sobre o ALGOL 60, de Rutishauser (1967).

A pesquisa sobre a semântica axiomática iniciou-se com Floyd (1967) e desenvolveu-se adicionalmente com Hoare (1969). A semântica de uma grande parte do Pascal foi descrita por Hoare e Wirth (1973) usando esse método. As partes que eles não concluíram envolviam os efeitos colaterais funcionais e as instruções goto. Essas eram consideradas as mais difíceis de serem descritas.

A técnica de usar pré-condições e pós-condições durante o desenvolvimento de programas é descrita (e defendida) por Dijkstra (1976) e também discutida detalhadamente em Gries (1981).

Uma boa introdução à semântica denotacional pode ser encontrada em Gordon (1979) e Stoy (1977).

Introduções a todos os três métodos de descrição da semântica discutidos neste capítulo podem ser encontradas em Marcotty et al. (1976). Outra boa referência para grande parte do material deste capítulo é Pagan (1981). A forma das funções semânticas denotacionais neste capítulo é similar à encontrada em Meyer (1990).

## QUESTÕES DE REVISÃO

1. Defina sintaxe e semântica.
2. A quem se destinam as descrições de linguagem?
3. Descreva a operação de um gerador de linguagem geral.
4. Descreva a operação de um reconhecedor de linguagem geral.
5. Qual é a diferença entre uma sentença e uma forma semântica?
6. Defina uma regra de gramática recursiva à esquerda.
7. Quais três extensões são comuns à maioria das EBNFs?
8. Descreva semântica estática e dinâmica.
9. A qual propósito servem os predicados em uma gramática de atributos?
10. Qual é a diferença entre um atributo sintetizado e um herdado?
11. Como é determinada a ordem de avaliação de atributos para as árvores de uma dada gramática de atributos?
12. Qual é o principal uso das gramáticas de atributos?
13. Qual é o problema quando se usa um interpretador puro de software para a semântica operacional?
14. Explique o que as pré-condições e as pós-condições de determinada instrução significam na semântica axiomática.
15. Descreva a abordagem de usar a semântica axiomática para provar a exatidão de determinado programa.
16. Descreva o conceito básico da semântica denotacional.
17. De qual maneira a semântica operacional e a semântica denotacional diferem?

## PROBLEMAS

1. Os dois modelos matemáticos de descrição de linguagens são geração e reconhecimento. Descreva como cada um pode definir a sintaxe de uma linguagem de programação.
2. Escreva descrições EBNF e de grafo de sintaxe para cada item:
  - a. Uma instrução de cabeçalho (*header*) para definição de uma classe Java
  - b. Uma instrução de chamada a um método Java
  - c. Uma instrução **switch C**
  - d. Uma definição **union C**
  - e. Literais **float C**
3. Usando a gramática do Exemplo 3.2, mostre uma árvore de análise e uma derivação à extrema esquerda das seguintes instruções:
  - a.  $A = A * (B + (C * A))$
  - b.  $B = C * (A * C + B)$
  - c.  $A = A * (B + (C))$
4. Usando a gramática do Exemplo 3.4, mostre uma árvore de análise e uma derivação à extrema esquerda das seguintes instruções:
  - a.  $A = (A + B) * C$
  - b.  $A = B + C + A$
  - c.  $A = A * (B + C)$
  - d.  $A = B * (C * (A + B))$
5. Prove que a seguinte gramática é ambígua:
$$\begin{aligned} <\$> &\rightarrow <A> \\ <A> &\rightarrow <A> + <A> \mid <\text{id}> \\ <\text{id}> &\rightarrow a \mid b \mid c \end{aligned}$$
6. Modifique a gramática do Exemplo 3.4 para adicionar um operador menos unário que tenha uma precedência mais alta do que + ou \*.

7. Descreva, em português, a linguagem definida pela seguinte gramática:

$$\begin{aligned} <S> &\rightarrow <A> <B> <C> \\ <A> &\rightarrow a \quad | \quad a \\ <B> &\rightarrow b \quad | \quad b \\ <C> &\rightarrow c \quad | \quad c \end{aligned}$$

8. Considere a seguinte gramática:

$$\begin{aligned} <S> &\rightarrow <A> a <B> b \\ <A> &\rightarrow <A> b \quad | \quad b \\ <B> &\rightarrow a \quad | \quad a \end{aligned}$$

Quais das seguintes sentenças estão na linguagem gerada por essa gramática?

- a. baab
- b. bbbab
- c. bbaaaaa
- d. bbaab

9. Considere a seguinte gramática:

$$\begin{aligned} <S> &\rightarrow a <S> c <B> \mid <A> \mid b \\ <A> &\rightarrow c <A> \mid c \\ <B> &\rightarrow d \mid <A> \end{aligned}$$

Quais das seguintes sentenças estão na linguagem gerada por essa gramática?

- a. abed
- b. acccbd
- c. acccbcc
- d. acd
- e. accc

10. Escreva uma gramática para a linguagem consistindo em seqüências que têm  $n$  cópias da letra a seguida do mesmo número de cópias da letra b, em que  $n > 0$ . Por exemplo, as seqüências ab, aaaabbba e aaaaaaaaaaaaaaaaaaaa estão na linguagem, mas a, abb, ba e aaabb não estão.

11. Desenhe árvores de análise para as sentenças aabcc e aaabbcc, como derivadas da gramática do Problema 10.

12. Usando as instruções da máquina virtual dada na Seção 3.5.1.1, apresente uma definição semântica operacional do seguinte:

- a. **repeat** do Pascal
- b. **for** da Ada
- c. DO do FORTRAN da forma: DO N K = start, end, step
- d. **if-then-else** do Pascal
- e. **for** do C
- f. **switch** do C

13. Compute a pré-condição mais fraca para cada uma das seguintes instruções de atribuição e pós-condições:

- a.  $a = 2 * (b - 1) - 1 \quad \{a > 0\}$
- b.  $b = (c + 10) / 3 \quad \{b > 6\}$
- c.  $a = a + 2 * b - 1 \quad \{a > 1\}$
- d.  $x = 2 * y + x - 1 \quad \{x > 11\}$

14. Compute a pré-condição mais fraca para cada uma das seguintes seqüências de instruções de atribuição e suas pós-condições:

- a.  $a = 2 * b + 1;$   
 $b = a - 3$   
 $\{b < 0\}$
- b.  $a = 3 * (2 * b + a);$   
 $b = 2 * a - 1$   
 $\{b > 5\}$

15. Escreva a função de correspondência da semântica denotacional para as seguintes instruções:

- a. **for** da Ada
- b. **repeat** do Pascal
- c. expressões booleanas do Java

d. **for** do C

e. **switch** do C

16. Qual é a diferença entre um atributo intrínseco e um atributo sintetizado?
17. Escreva uma gramática de atributos cuja base BNF seja a do Exemplo 3.6 na Seção 3.4.5, mas cujas regras de linguagem sejam as seguintes: tipos de dados não podem ser misturados em expressões, mas as instruções de atribuição não precisam ter os mesmos tipos em ambos os lados do operador de atribuição.
18. Escreva uma gramática de atributos cuja base BNF seja a do Exemplo 3.2 e cujas regras de tipo sejam as mesmas do exemplo de instrução de atribuição da Seção 3.4.5.
19. Prove que o seguinte programa está correto:

```
{x = Vx, and y = Vy}
```

```
temp = x;
```

```
x = y;
```

```
y = temp;
```

```
{x = Vy and y = Vx}
```

20. Prove que o seguinte programa está correto:

```
{n > 0}
```

```
cont = n;
```

```
soma = 0;
```

```
while cont <> 0 do
```

```
    soma = soma + cont;
```

```
    cont = cont - 1;
```

```
end
```

```
{soma = 1 + 2 + ... + n}
```

Hidden page

## Capítulo 4

# Análise Léxica e Sintática



### **Larry Wall**

Como um lingüista treinado no campo, a paixão de Larry Wall é desenvolver a cultura da computação junto com sua tecnologia. Seu *m*, metaconfiguração e programas de conserto foram fundamentais no sucesso do movimento de código aberto, mas ele é mais conhecido pelo projeto "empedernidamente antropológico" da Perl. Trabalha como membro sênior da O'Reilly and Associates, que publica o infame *Camel Book* de programação Perl.

- 4.1** Introdução
- 4.2** Análise Léxica
- 4.3** O Problema da Análise Sintática
- 4.4** Análise Descendente Recursiva
- 4.5** Análise Baixo-Cima

Uma investigação séria de projeto de compiladores requer ao menos um semestre de estudo intensivo, incluindo o projeto e a implementação de um compilador para uma pequena mas realística linguagem de programação. Metade ou dois terços de tal curso é devotada às análises léxica e sintática. Contudo, uma significativa e valiosa introdução pode ser apresentada em um único capítulo, como este.

Este capítulo começa com uma introdução à análise léxica, incluindo um exemplo simples. A seguir, o problema geral da análise sintática é discutido, apresentando as duas principais abordagens e a complexidade da análise. A técnica de implementação descendente recursiva para analisadores EE é introduzida, assim como dois exemplos de partes de um analisador descendente recursivo. A última seção discute a análise baixo-cima e o algoritmo de análise ED. Essa seção inclui um exemplo de uma pequena tabela de análise ED e a análise de uma cadeia usando o processo ED.

## 4.1 Introdução

Três diferentes abordagens de implementação de linguagens de programação foram introduzidas no Capítulo 1: compilação, interpretação pura e implementação híbrida. A compilação usa um programa tradutor, o compilador, que traduz os programas escritos em linguagem de alto nível para código de máquina. A compilação geralmente é usada para implementar linguagens de programação utilizadas em grandes aplicações industriais, frequentemente escritas em C++ e COBOL. Os sistemas de interpretação pura não fazem tradução; os programas são interpretados em sua forma original por um software interpretador. A interpretação pura é usada em pequenos sistemas, nos quais a eficiência de execução não é fundamental, como scripts embutidos em documentos HTML, escritos em linguagens como a JavaScript. Os sistemas de implementação híbrida traduzem os programas escritos em linguagens de alto nível para formas intermediárias, que são interpretadas. Eses sistemas são amplamente usados, em grande parte pela popularidade do Java e da Perl. Tradicionalmente, os sistemas híbridos levam a execuções de programa muito mais lentas do que os sistemas compilados. Contudo, avanços recentes na tecnologia de interpretação tornaram os sistemas híbridos competitivos com os sistemas compilados, pelo menos no caso do Java.

Os analisadores sintáticos, ou *parsers*, são quase sempre baseados em uma descrição formal da sintaxe dos programas. O formalismo mais usual para descrição de sintaxe é a gramática livre de contexto ou BNF, introduzida no Capítulo 3. Usar a BNF traz pelo menos três vantagens imediatas. Primeira, suas descrições de sintaxe de programa são claras e concisas, tanto para leitores humanos como para os sistemas de software que as utilizam. Segunda, a descrição BNF pode ser usada como a base do analisador sintático. A terceira vantagem é que as implementações baseadas na BNF são relativamente fáceis de manter, devido à sua clara modularidade.

Praticamente todos os compiladores separam a tarefa da análise sintática em duas partes distintas, análise léxica e sintática, embora essa terminologia seja confusa. O analisador léxico trata as construções de pequena escala da linguagem, tais como nomes e literais numéricos. O analisador sintático trata as construções de larga escala, tais como expressões, instruções e unidades de programa. Na Seção 4.2, é introduzida a análise léxica, e nas Seções 4.3, 4.4 e 4.5 são abordados os analisadores sintáticos.

Existem várias razões pelas quais a análise léxica deve ser separada da análise sintática, como descrito a seguir:

1. Simplicidade — As técnicas para a análise léxica são menos complexas do que as exigidas para análise sintática; assim, o processo de análise léxica pode ser mais simples se for separado. Também, ao se remover os detalhes de baixo nível inerentes à análise léxica do analisador sintático, ele se torna menor e mais limpo.
2. Eficiência — Embora valha a pena otimizar o analisador léxico, uma vez que a análise léxica exige uma porção significativa do tempo total de compilação, não é vantajoso otimizar o analisador sintático. A separação facilita essa otimização seletiva.
3. Portabilidade — Uma vez que o analisador léxico lê arquivos de entrada contendo o programa e freqüentemente usa retentores na operação de leitura, ele é de certa forma dependente da plataforma. O analisador sintático, porém, pode ser independente da plataforma. É sempre bom isolar partes dependentes da máquina em qualquer sistema de software.

## 4.2 Análise Léxica

Um analisador léxico é essencialmente um “casamenteiro” de padrões. Ele tenta encontrar uma subcadeia de uma dada cadeia de caracteres que casa com o padrão dado. O casamento de padrões é uma parte tradicional da computação. Um de seus primeiros usos foi com editores de texto, como o *ed*, introduzido em uma das primeiras versões do UNIX. Desde então, o casamento de padrões tem aparecido em algumas linguagens de programação, como Perl e JavaScript.

Um analisador léxico serve como linha de frente para o analisador sintático. Técnicamente, a análise léxica faz parte da análise sintática. O analisador léxico faz a análise sintática no nível mais baixo da estrutura do programa. Um programa de entrada para um compilador aparece como uma única cadeia de caracteres. O analisador léxico coleta caracteres em agrupamentos lógicos e atribui códigos internos aos agrupamentos de acordo com sua estrutura. Os agrupamentos de caracteres são chamados *lexemas*. Os códigos internos são chamados *símbolos (tokens)*. Os lexemas são identificados pelo casamento da cadeia de caracteres de entrada com os padrões. Embora os símbolos sejam usualmente identificados com valores inteiros, com freqüência são referenciados por constantes nomeadas, para fins de legibilidade do analisador. Considere o seguinte exemplo de uma instrução de atribuição:

```
Soma = SomaVelha - Valor / 100;
```

A seguir estão os símbolos e os lexemas dessa instrução:

Símbolo	Lexema
IDENTIFICADOR	Soma
OP_ATRIBUIÇÃO	=
IDENTIFICADOR	SomaVelha
OP_SUBTRAÇÃO	-
IDENTIFICADOR	Valor
OP_DIVISÃO	/
LITERAL_INT	100
PONTOeVIRGULA	;

Os analisadores léxicos extraem lexemas de uma dada cadeia de entrada e produzem os respectivos símbolos. Nos primeiros compiladores, os analisadores léxicos processavam um arquivo inteiro com o programa fonte e produziam um arquivo contendo símbolos e lexemas. Atualmente, a maioria dos analisadores léxicos é subprograma que produz o próximo lexema e seu símbolo associado a partir da entrada e os retorna para o chamador, que é o analisador sintático. A única visão do programa de entrada que o analisador sintático tem é a saída do analisador léxico, um lexema de cada vez.

O processo do analisador léxico inclui saltar comentários e brancos fora dos lexemas, uma vez que eles não são relevantes para o significado do programa. Ele faz também a inserção dos lexemas de nomes definidos pelo programador na tabela de símbolos, que é usada nas fases posteriores da compilação. Finalmente, o analisador léxico detecta erros de sintaxe nos símbolos, tais como literais de números reais inválidos, e reporta esses erros ao usuário.

Existem três abordagens para se construir um analisador léxico:

1. Escrever uma descrição formal dos padrões de símbolos da linguagem, utilizando uma linguagem de descrição relacionada com expressões regulares e usar uma ferramenta de software para gerar automaticamente o analisador. Existem muitas ferramentas disponíveis para isso. A mais antiga e mais acessível, chamada lex, normalmente está incluída nos sistemas UNIX.
2. Construir um diagrama de transição de estados que descreva os padrões de símbolos da linguagem e escrever um programa que implemente o diagrama.
3. Construir um diagrama de transição de estados que descreva os padrões de símbolos da linguagem e construir manualmente uma implementação do diagrama dirigida por tabela.

Um diagrama de transição de estados, ou apenas diagrama de estados, é um grafo relacionado aos grafos sintáticos, introduzidos no Capítulo 3. Os vértices do diagrama de estado são rotulados com os nomes dos estados. Os arcos são rotulados com os caracteres de entrada que causam as transições. Um arco pode incluir também as ações que o analisador léxico deve executar quando a transição é efetuada.

Os diagramas de estados da forma usados pelos analisadores léxicos são representações de uma classe de máquinas matemáticas chamada *autômato finito*. Os autômatos finitos podem ser projetados para reconhecer uma classe de linguagens chamada *linguagem regular*. Expressões regulares e gramáticas regulares são dispositivos geradores para linguagens regulares. Os símbolos de uma linguagem de programação constituem uma linguagem regular.

Ilustraremos agora a construção de um analisador léxico com um diagrama de estados e o código que o implementa. O diagrama de estados poderia incluir simplesmente os estados e as transições para cada padrão de símbolo. Contudo, isso resulta em um diagrama complexo e muito grande. Portanto, consideremos os modos de simplificá-lo. Suponha que necessitemos de um analisador que reconheça apenas nomes de programas, palavras reservadas e literais inteiros. Os nomes neste exemplo consistem de cadeias de letras maiúsculas, minúsculas e de dígitos, mas devem começar com uma letra. Eles não têm limitação de tamanho. A primeira coisa a se observar é que os nomes e as palavras reservadas têm padrões similares. Embora seja possível construir um diagrama de estados para reconhecer qualquer palavra reservada específica de uma linguagem de programação, isso resultaria em um diagrama de estados grande. É muito mais simples e mais rápido fazer com que o analisador reconheça nomes e palavras reservadas com o mesmo padrão, e pesquisar uma tabela para determinar quais nomes são palavras reservadas.

Outra oportunidade para simplificar o diagrama de transição é com os símbolos de literais inteiros. Existem dez caracteres diferentes que podem iniciar um lexema de literal inteiro. Isso exigiria dez transições a partir do estado inicial do diagrama. Uma vez que os dígitos específicos não interessam ao analisador léxico, podemos construir um diagrama de estados muito mais compacto se definirmos uma classe de caracteres para dígitos e utilizarmos uma única transição de cada caráter dessa classe para o estado que coleta literais inteiros.

A maneira mais importante de simplificar o diagrama de transições para padrões de símbolos é usar uma classe de caracteres para letras, uma vez que existem 52 letras, todas tratadas da mesma forma. Assim, definimos uma nova classe de letras.

A seguir, definimos alguns subprogramas utilitários para as tarefas comuns, internas ao analisador léxico. Primeiro, necessitamos de um subprograma que podemos chamar `ObtemCarater`, que tem várias obrigações. Quando chamado, `ObtemCarater` pega o próximo caráter do programa fonte e o coloca na variável global `ProximoCarater`. Isso pode exigir a leitura de uma nova linha ou de novo retentor de entrada. `ObtemCarater` deve também determinar a classe a que o caráter pertence e colocá-la na variável global `ClasseCarater`. O lexema que está sendo construído pelo analisador, que pode ser uma cadeia ou um vetor de caracteres, será chamado `lexema`. Implementamos o processo de colocar o caráter contido em `ProximoCarater` na variável `lexema` no subprograma `AcrescentaCarater`. Esse subprograma deve ser explicitamente chamado, uma vez que os programas incluem muitos caracteres que não necessitam ser postos em `lexemas`, como comentários e espaços em branco entre `lexemas`. Finalmente, necessitamos de um subprograma chamado `pesquisa` para determinar se o conteúdo corrente de `lexema` é uma palavra reservada ou um nome. Esse subprograma retorna zero se o `lexema` não for uma palavra reservada, mas retornará o código do símbolo de qualquer palavra reservada. Isso pressupõe que o código de símbolo para nomes seja zero. Os códigos de símbolo são números arbitrariamente atribuídos a símbolos pelo escritor do compilador.

O diagrama de estados na Figura 4.1 (ver p. 158) descreve os padrões para os nossos símbolos e inclui as ações exigidas em cada transição do diagrama.

A implementação desse diagrama de estados é relativamente fácil. A seguinte função C é um exemplo.

```
/* lex-um simples analisador léxico */

int lex () {
    ObtemCarater ();
    switch (ClasseCarater) {

        /* analisa identificadores e palavras reservadas */

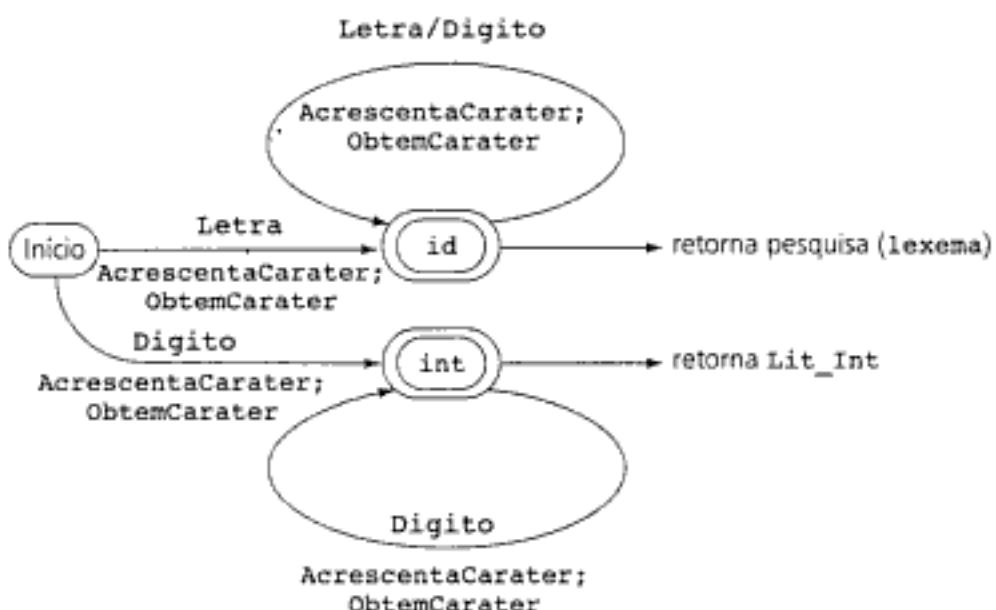
        case LETRA:
            AcrescentaCarater ();
            ObtemCarater ();
            while (ClasseCarater == LETRA ||
                   ClasseCarater == DIGITO) {
                AcrescentaCarater ();
                ObtemCarater ();
            }
            return pesquisa (lexema);
            break;
    }
}
```

```

/* analisa literais inteiros */

case DIGITO:
    AcrescentaCarater ();
    ObtemCarater ();
    while (ClasseCarater == DIGITO) {
        AcrescentaCarater ();
        ObtemCarater ();
    }
    return LIT_INT;
    break;
} /* Fim do switch */
} /* Fim da função lex */

```



**FIGURA 4.1** Um diagrama de estados para reconhecer nomes, palavras reservadas e literais inteiros.

Esta função ilustra a relativa simplicidade dos analisadores léxicos. Evidentemente deixamos de fora as funções utilitárias que executam a maioria do trabalho exigido, bem como vários detalhes. Além disso, é um conjunto muito pequeno e simples de símbolos.

### 4.3 O Problema da Análise Sintática

A parte do processo de análise que se refere à sintaxe geralmente é chamada de *parsing*.

Nesta seção discutiremos o problema geral da análise sintática e serão introduzidas as duas principais categorias de algoritmos de análise, cima-baixo e baixo-cima, bem como a complexidade do processo de análise.

### 4.3.1 Introdução à Análise Sintática

Os analisadores das linguagens de programação constroem árvores de análise para os programas dados. Em alguns casos, a árvore de análise é implicitamente construída, significando que talvez apenas seja gerado o resultado de percorrer a árvore. Mas em todos os casos a informação necessária para construir a árvore é criada durante o processo de análise. As árvores de análise e as derivações incluem toda a informação sintática necessária para um processador da linguagem. Há duas metas distintas na análise sintática. A primeira é que a análise sintática deve verificar o programa de entrada para determinar se ele está sintaticamente correto. Quando um erro é encontrado, o analisador deve produzir uma mensagem de diagnóstico e retomar o processamento. Essa retomada significa que ele deve voltar a um estado normal e continuar a análise do programa de entrada. Isso é exigido para que o compilador encontre tantos erros quanto possível durante uma única análise do programa de entrada. Se não for bem feita, a retomada pode criar mais erros, ou pelo menos mais mensagens de erro. A segunda meta da análise sintática é produzir uma árvore de análise completa ou pelo menos esboçar a sua estrutura. Nos dois casos, o resultado é usado como a base para a tradução.

Os analisadores são classificados de acordo com a direção na qual eles constroem a árvore de análise. As duas grandes classes são **cima-baixo**, na qual a árvore é construída da raiz para as folhas, e **baixo-cima** na qual a árvore é construída das folhas para a raiz.

Todos os algoritmos de análise comumente utilizados operam sob a obrigação de que eles jamais olharão à frente mais do que um símbolo no programa de entrada. Isso é suficiente para permitir a análise. Como resultado, tem-se um analisador elegante e altamente eficiente.

Neste capítulo usaremos um pequeno conjunto de convenções de notação para símbolos gramaticais e cadeias, tornando a discussão menos confusa. Para as linguagens formais, são as seguintes:

Símbolos terminais — letras minúsculas do início do alfabeto (a, b,...)

Símbolos não-terminais — letras maiúsculas do início do alfabeto (A, B,...)

Terminais ou não-terminais — letras maiúsculas do final do alfabeto (W, X, Y, Z)

Cadeias de terminais — letras minúsculas do fim do alfabeto (w, x, y, z)

Cadeias mistas (terminais e/ou não-terminais) — letras minúsculas gregas (α, β, δ, γ)

Nas linguagens de programação, os símbolos terminais são as construções sintáticas de pequena escala da linguagem, incluindo os símbolos de pontuação e os literais numéricos. Os símbolos não-terminais são usualmente nomes conotativos ou abreviações entre colchetes angulados. Por exemplo, <instrução\_while>, <expressão> e <def\_função>. Os símbolos terminais e não-terminais serão usados quando discutirmos os algoritmos de análise. As sentenças de uma linguagem (programas, no caso de uma linguagem de programação) são cadeias de terminais. As cadeias mistas são usadas nos algoritmos de análise.

### 4.3.2 Analisadores Cima-Baixo

A derivação que é feita durante a análise cima-baixo é mais à esquerda. Um analisador cima-baixo esboça ou constrói a árvore de análise em pré-ordem. O caminhamento na árvore de análise em pré-ordem se inicia pela raiz. Cada vértice é visitado antes que seus desvios sejam seguidos. Estes são percorridos na ordem esquerda-direita.

Em termos de derivação, um analisador cima-baixo pode ser descrito como se segue: dada uma forma sentencial, que é parte de uma derivação mais à esquerda, a tarefa do

analisador é encontrar a próxima forma sentencial nessa derivação. A forma geral de uma forma sentencial à esquerda é  $xA_$ , em que, de acordo com nossa convenção notacional,  $x$  é uma cadeia de símbolos terminais,  $A$  é um não-terminal e  $_$  é uma cadeia mista. Uma vez que  $x$  contém somente terminais,  $A$  é o não-terminal mais à esquerda na forma sentencial, então ele deve ser expandido para se obter a próxima forma sentencial na derivação mais à esquerda. A determinação da próxima forma sentencial depende da escolha da regra gramatical correta que tem  $A$  como seu lado esquerdo (LE). Por exemplo, se a forma sentencial corrente é

$xAa$

e as regras de  $A$  são  $A \rightarrow bB$ ,  $A \rightarrow cBb$  e  $A \rightarrow a$ , um analisador cima-baixo deve escolher entre estas três regras, para obter a próxima forma sentencial, que pode ser  $xbBa$ ,  $xcBba$  ou  $xaa$ .

Sob a restrição de apenas um símbolo considerado à frente, um analisador cima-baixo deve escolher o LD correto baseado no próximo símbolo do programa de entrada.

Os algoritmos de análise cima-baixo mais comuns são estreitamente relacionados. Um analisador descendente recursivo é uma versão codificada de um analisador sintático baseado diretamente na descrição BNF da linguagem. A alternativa mais comum à recursão descendente é usar uma tabela de análise ao invés do código para implementar as regras BNF. Ambas, chamadas *algoritmos EE*, são igualmente poderosas, significando que elas funcionam no mesmo subconjunto de gramáticas. O primeiro E em EE especifica a varredura da esquerda para a direita na entrada; o segundo E especifica que uma derivação mais à esquerda é gerada. A Seção 4.4 introduz a abordagem descendente recursiva para implementar um analisador EE.

### 4.3.3 Analisadores Baixo-Cima

Um analisador baixo-cima constrói uma árvore de análise começando das folhas e progredindo até a raiz. Ele produz o inverso de uma derivação mais à direita. Em termos de derivação, o analisador baixo-cima pode ser descrito como se segue: dada uma forma sentencial à direita  $\alpha$ , o analisador deve determinar qual subcadeia de  $\alpha$  é o lado direito (LD) de alguma regra na gramática que deve ser reduzida ao seu LE para produzir a forma sentencial dada na derivação mais à direita. Por exemplo, o primeiro passo para um analisador baixo-cima é determinar qual subcadeia da sentença inicial é o LD a ser reduzido ao seu correspondente LE para obter a penúltima forma sentencial na derivação.

Os algoritmos mais comuns para analisadores baixo-cima pertencem à família ED, na qual o E especifica uma varredura da esquerda para a direita na entrada e o D especifica que uma derivação mais à direita é gerada.

### 4.3.4 A Complexidade da Análise

Os algoritmos de análise que funcionam em qualquer gramática não-ambígua são complexos e inefficientes. Na verdade, a complexidade de tais algoritmos é  $O(n^3)$ , o que significa que o tempo gasto para a análise é da ordem do cubo do comprimento da cadeia a ser analisada. Essa quantidade de tempo relativamente grande é necessária porque esses algoritmos freqüentemente fazem cópias e reanalismam partes da sentença que está sendo analisada. Esses algoritmos não são úteis em processos práticos, como a análise sintática de um compilador, que deve ser executada com freqüência. Nessa situação, os cientistas da com-

putação procuram algoritmos mais rápidos, ainda que não sejam tão gerais. Isso significa um compromisso entre generalidade e eficiência. Em termos de análise, têm sido encontrados algoritmos mais rápidos que funcionam apenas em um subconjunto de todas as gramáticas possíveis. Eles são aceitáveis, desde que o subconjunto inclua gramáticas que descrevam linguagens de programação (de fato, como discutido no Capítulo 3, a classe inteira das gramáticas não é adequada para descrever toda a sintaxe da maioria das linguagens de programação).

Todos os algoritmos usados como analisadores sintáticos de compiladores têm complexidade  $O(n)$ , o que significa que o tempo gasto é linearmente relacionado ao comprimento da cadeia a ser analisada. Isso é muito mais eficiente que os algoritmos  $O(n^3)$ .

## 4.4 Análise Descendente Recursiva

Essa seção introduz o processo de implementação do analisador cima-baixo, descendente recursivo.

### 4.4.1 O Processo da Análise Descendente Recursiva

Um analisador descendente recursivo é assim chamado porque consiste de uma coleção de subprogramas, muitos dos quais são recursivos, e porque produz uma árvore de análise cima-baixo (descendente). Essa recursão reflete a natureza das linguagens de programação, que incluem várias estruturas recursivas diferentes. A sintaxe dessas estruturas é naturalmente descrita com regras gramaticais recursivas. Por exemplo, as instruções são freqüentemente aninhadas em outras instruções, assim como os parênteses nas expressões também devem ser propriamente aninhados.

A EBNF é muito adequada para analisadores descendentes recursivos. Lembre-se do Capítulo 3, no qual as principais extensões da EBNF são as chaves, que especificam que a parte nelas contida pode aparecer zero ou mais vezes, e colchetes, que especificam que os símbolos neles contidos podem aparecer zero ou uma vez. Note que, em ambos os casos, os símbolos envolvidos são opcionais. Por exemplo:

```
<instrução_if> → if <expressão_lógica> <instrução> [else <instrução>]
<lista_ident> → ident{, ident}
```

Na primeira regra, a cláusula **else** de uma instrução **if** é opcional. Na segunda, uma **<lista\_ident>** é um identificador seguido por zero ou mais repetições de uma vírgula e um identificador.

Um analisador descendente recursivo tem um subprograma para cada não-terminal da gramática. A responsabilidade deste subprograma é a seguinte: dada uma cadeia de entrada, ele investiga a árvore que pode ter como raiz o não-terminal e cujas folhas casam com a cadeia de entrada. Isto é, um subprograma de um analisador descendente recursivo é um analisador para a linguagem (conjunto de cadeias) que pode ser gerada por seu não-terminal associado. Na discussão que se segue, assumimos que cada não-terminal possui uma única regra, possivelmente com múltiplos LDs separados por operadores OU.

Considere a seguinte descrição EBNF de expressões aritméticas simples:

```
<expr> → <termo> { (+ | -) <termo> }
```

```
<termo> → <fator> {(* | /) <fator>}
<fator> → ident | (<expr>)
```

No exemplo que se segue da função descendente recursiva `expr`, o analisador léxico é uma função apropriadamente chamada `lex`. Ela obtém o próximo lexema e coloca seu código de símbolo na variável global `proximoSimbolo`. Os códigos de símbolo são definidos como constantes nomeadas. Por exemplo, `Codigo_Mais` é uma constante nomeada para o código do símbolo mais.

Um subprograma descendente recursivo para uma regra com um único LD é relativamente simples. Para cada símbolo terminal no LD é feita uma comparação com `proximoSimbolo`. Se eles não casam, isso é um erro de sintaxe. Se eles casam, o analisador léxico é chamado para obter o próximo símbolo de entrada. Para cada não-terminal, o subprograma de análise correspondente é chamado.

O subprograma descendente recursivo para a primeira regra na gramática do exemplo acima, escrito em C, é

```
/* função expr
   Analisa cadeias na linguagem gerada pela regra:
   <expr> -> <termo> {(+ | -) <termo> }
   */

void expr () {

    /* analisa o primeiro termo */

    termo ();

    /* enquanto o próximo símbolo for + ou -, chama lex para obter
       o próximo símbolo e analisa o próximo termo */

    while (proximoSimbolo == Codigo_Mais ||
           proximoSimbolo == Codigo_Menos) {
        lex ();
        termo ();
    }
}
```

Os subprogramas de análise descendente recursiva são escritos segundo a convenção de que cada um deixa o próximo símbolo da entrada em `proximoSimbolo`. Assim, sempre que uma função do analisador se inicia é certo que `proximoSimbolo` contém o símbolo mais à esquerda da entrada que ainda não foi usado no processo de análise.

A parte da linguagem que a função `expr` analisa consiste de um ou mais termos, separados pelo operador + ou -. Essa é a linguagem gerada pelo não-terminal `<expr>`. Portanto, ela chama primeiro a função que analisa termos (`<termo>`). Então ela continua a chamar esta última enquanto encontra os símbolos de `Codigo_Mais` ou `Codigo_Menos` (que são passados pela chamada a `lex`). Esta função descendente recursiva é mais simples do que a maioria porque essa regra possui apenas um LD. Além disso, ela não inclui código para detecção de erros de sintaxe, porque não existem erros detectáveis associados à regra gramatical.

Um subprograma de análise descendente recursiva para um não-terminal cuja regra possui mais de um LD começa com um código para determinar qual LD deve ser analisado.

Cada LD é examinado (quando se constrói o compilador) para determinar o conjunto de símbolos terminais que podem aparecer no início das sentenças que ele pode gerar. Comparando esses conjuntos com o próximo símbolo da entrada, o analisador pode escolher o LD correto.

A função para o não-terminal <fator> da nossa gramática de expressões aritméticas deve escolher um de seus LDs. Ela também inclui detecção de erros. Na função para <fator> a reação ao ser detectado um erro de sintaxe é simplesmente chamar a função `erro`. Em um analisador real, uma mensagem de diagnóstico deve ser produzida quando um erro é detectado. Além disso, a maioria dos compiladores deve retomar a execução de forma segura, para que o processo de análise possa continuar.

```
/* Função fator
   Analisa cadeias na linguagem gerada pela regra:
   <fator> -> ident | (<expr>)

   */
void fator () {

    /* determina qual LD */
    if (proximoSimbolo == codigo_ident)

        /* para o LD ident apenas chama lex */

        lex ();

    /* se o LD é (<expr>) - chama lex para passar o parêntese
       esquerdo, chama expr e confere o parêntese
       direito */

    else if (proximoSimbolo == parent_esq) {
        lex ();
        expr ();
        if (proximoSimbolo == parent_dir)
            lex ();
        else
            erro ();
    } /* fim do else if (proximoSimbolo == ... */

    /* o símbolo não é ident nem parêntese esquerdo */

    else erro ();

}

}
```

O objetivo dessa breve discussão é convencê-lo de que um analisador descendente recursivo pode ser escrito facilmente se uma gramática apropriada estiver disponível para a linguagem. A questão-chave agora é a caracterização de uma gramática “apropriada” para a análise descendente recursiva.

#### 4.4.2 A Classe de Gramáticas EE

Uma característica simples das gramáticas que causa um problema catastrófico para os analisadores EE é a recursão à esquerda. Por exemplo, considere a seguinte regra:

$$A \rightarrow A + B$$

Um subprograma analisador descendente recursivo para A imediatamente chama a si próprio para analisar o primeiro símbolo em seu LD. Essa ativação do subprograma analisador A imediatamente chama a si próprio novamente, e assim por diante. É fácil ver que isso leva a lugar nenhum.

A recursão indireta à esquerda apresenta o mesmo problema que à direita. Por exemplo, suponha que se tenha

$$A \rightarrow B \text{ a } A$$

$$B \rightarrow A \text{ b}$$

Um analisador descendente recursivo para essas regras faria com que o subprograma A imediatamente chamassem o subprograma B, o qual chamaria imediatamente A. Assim, acontece o mesmo problema da recursão direta à esquerda. O problema da recursão à esquerda não se restringe à abordagem descendente recursiva para construir analisadores Cima-Baixo. É um problema para todos os algoritmos deste tipo de análise. Felizmente, a recursão à esquerda não é problema para os algoritmos de análise Baixo-Cima.

Existe um algoritmo para modificar uma dada gramática que remove a recursão à esquerda, tanto direta quanto indiretamente (Aho et al., 1986), mas não o abordaremos aqui. Quando se escreve uma gramática para uma linguagem de programação é preciso evitar a inclusão de qualquer recursão à esquerda.

A recursão à esquerda não é a única característica da gramática que impede a análise Cima-Baixo. Uma outra é se o analisador poderá sempre escolher o LD correto baseando-se apenas no próximo símbolo da entrada. Existe um teste relativamente simples de gramáticas recursivas não à esquerda que indica se isso pode ser feito, chamado de **teste de disjunção paritária**. Esse teste requer que possamos determinar um conjunto com base nos LDs de um dado símbolo não-terminal em uma gramática. Esses conjuntos, chamados de PRIMEIRO, são definidos como

$$\text{PRIMEIRO}(\alpha) = \{ \alpha \mid \alpha \Rightarrow^* a \beta \} \quad (\text{se } \alpha \Rightarrow^* \epsilon, \epsilon \text{ está em } \text{PRIMEIRO}(\alpha))$$

Um algoritmo para computar PRIMEIRO para cada cadeia mista a pode ser encontrado em Aho et al. (1986). Para nossos propósitos, PRIMEIRO pode ser computado por inspeção na gramática. O teste de disjunção paritária é

Para cada não-terminal, A, na gramática que tenha mais de um LD, para cada par de regras,  $A \rightarrow \alpha_i$  e  $A \rightarrow \alpha_j$ , deve ser constatado que  $\text{PRIMEIRO}(\alpha_i) \cap \text{PRIMEIRO}(\alpha_j) = \emptyset$

(A interseção dos dois conjuntos PRIMEIRO( $\alpha_i$ ) e PRIMEIRO( $\alpha_j$ ) deve ser o conjunto vazio.)

Em outras palavras, se um não-terminal A possui mais de um LD, o primeiro símbolo terminal que pode ser gerado em uma derivação para cada um deles deve ser apenas o daquele LD. Considere as seguintes regras

$$A \rightarrow aB \mid bAb \mid c$$

Os conjuntos PRIMEIRO para os LDs dessas regras são {a}, {b} e {c}, que são claramente disjuntos. Portanto, essas regras passam no teste da disjunção paritária. Isso significa

que, em termos de um analisador descendente recursivo, o código do subprograma para análise do não-terminal A pode escolher qual LD ele está tratando olhando apenas o primeiro símbolo terminal da entrada que é gerado pelo não-terminal. Agora considere as regras

$$A \rightarrow aB \mid aAb$$

Os conjuntos PRIMEIRO para os LDs dessas regras são  $\{a\}$  e  $\{a\}$  que claramente não são disjuntos. Assim, essas regras não passam no teste de disjunção paritária. Em termos do analisador, o subprograma para A não teria como determinar qual LD estava sendo analisado olhando apenas para o próximo símbolo da entrada porque, se for um a, poderia ser qualquer um dos LDs.

Em muitos casos, uma gramática que não passa pelo teste da disjunção paritária pode ser modificada. Por exemplo, considere as regras

$$\langle \text{variável} \rangle \rightarrow \text{identificador} \mid \text{identificador} [\langle \text{expressão} \rangle]$$

Elas declaram que  $\langle \text{variável} \rangle$  é um identificador ou um identificador seguido por uma expressão entre colchetes (um subscrito). Essas regras claramente não passam no teste de disjunção paritária porque os LDs se iniciam com o mesmo terminal, identificador. Esse problema pode ser resolvido com um processo chamado **fatoração à esquerda**.

Daremos agora uma olhada informal na fatoração à esquerda. Considere as regras anteriores para  $\langle \text{variável} \rangle$ . Ambos LDs começam com identificador. As partes que seguem identificador são  $\epsilon$  (uma cadeia vazia) e  $[\langle \text{expressão} \rangle]$ . As duas regras podem ser substituídas por

$$\langle \text{variável} \rangle \rightarrow \text{identificador} \langle \text{nova} \rangle$$

onde  $\langle \text{nova} \rangle$  é definida como

$$\langle \text{nova} \rangle \rightarrow \epsilon \mid [\langle \text{expressão} \rangle]$$

Não é difícil verificar que juntas, essas duas regras geram a mesma linguagem com a qual começamos. Contudo, essas duas regras passam no teste de disjunção paritária.

Se a gramática estiver sendo usada como base para um analisador descendente recursivo, existe uma alternativa para a fatoração à esquerda. Usando uma extensão EBNF, o problema desaparece de uma maneira bastante similar à solução da fatoração à esquerda. Considere novamente as primeiras regras para  $\langle \text{variável} \rangle$ . O subscrito pode ser opcional, desde que colocado entre colchetes como em

$$\langle \text{variável} \rangle \rightarrow \text{identificador} [[\langle \text{expressão} \rangle]]$$

Nesta regra, os colchetes mais de fora são metassímbolos indicando que o seu conteúdo é opcional. Os colchetes internos são símbolos terminais da linguagem de programação que está sendo descrita. O importante é que substituímos duas regras por uma, que gera a mesma linguagem, mas passa no teste de disjunção paritária.

Um algoritmo formal para a fatoração à esquerda pode ser encontrado em Aho et al. (1986). A fatoração à esquerda não consegue resolver todos os problemas de disjunção paritária das gramáticas. Em alguns casos, as regras devem ser reescritas de outras maneiras para eliminar o problema.

## 4.5 Análise Baixo-Cima

Nesta seção, introduzimos o processo geral de análise Baixo-Cima e a descrição de um algoritmo analisador ED.

### 4.5.1 O Problema da Análise Baixo-Cima

Considere a seguinte gramática que gera expressões aritméticas com operadores de adição e multiplicação, parênteses e o operando, id.

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

Note que essa gramática gera as mesmas expressões aritméticas do exemplo na Seção 4.4. A diferença é que essa gramática é recursiva à esquerda, e isso é aceitável para analisadores Baixo-Cima. Note também que gramáticas para analisadores Baixo-Cima normalmente não incluem metassímbolos como os usados para especificar extensões à BNF. A seguinte derivação mais à direita ilustra essa gramática:

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

A parte sublinhada de cada forma sentencial nessa derivação é o LD que é reescrito como o seu correspondente LE para se obter a forma sentencial anterior. O processo de análise Baixo-Cima produz o inverso da derivação mais à direita. Assim, na derivação acima, o analisador Baixo-Cima começa com a última forma sentencial (a sentença de entrada) e produz a sequência de formas sentenciais desde aquela até restar apenas o símbolo de início, que nessa gramática é E. Em cada passo, a tarefa do analisador Baixo-Cima é encontrar o LD específico na forma sentencial que deve ser reescrito para se obter a próxima (anterior) forma sentencial. Em alguns casos, uma forma sentencial à direita pode incluir mais de um LD. Por exemplo, a forma sentencial à direita

$$E + T * id$$

incluirá três LDs, E + T, T e id. Somente um deles é o correto para ser reescrito. Por exemplo, se o E + T fosse escolhido para ser reescrito nessa forma sentencial, a forma resultante seria E \* id, mas E \* id não é uma forma sentencial à direita legal para a gramática dada. O LD correto em uma dada forma sentencial à direita a ser reescrito para se obter a forma sentencial anterior é chamado **manipulador**. O manipulador de uma forma sentencial à direita é único. A tarefa de um analisador Baixo-Cima é encontrar o manipulador de qualquer forma sentencial à direita dada, que possa ser gerada por sua gramática associada. Formalmente, o manipulador é definido como

Definição:  $\beta$  é o **manipulador** de uma dada forma sentencial à direita  $\gamma = \alpha\beta w$  se e somente se  $S \Rightarrow^* rm \alpha Aw \Rightarrow^* rm \alpha\beta w$

Nessa definição  $\Rightarrow^*$  rm especifica um passo na derivação mais à direita e  $\Rightarrow^0$  rm especifica zero ou mais passos na derivação mais à direita. Embora a definição de um manipulador seja matematicamente concisa, ela ajuda pouco na busca do manipulador de uma dada forma sentencial à direita. A seguir, forneceremos as definições de várias subcadeias de formas sentenciais que são relacionadas com os manipuladores. O propósito disso é oferecer alguma intuição a respeito dos manipuladores.

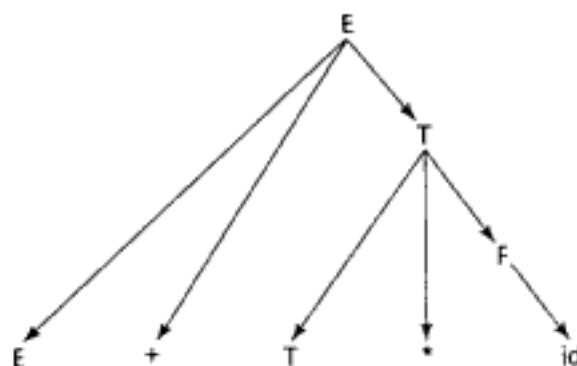
Definição:  $\beta$  é uma **frase** da forma sentencial à direita  $\gamma$  se e somente se  $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1 \beta \alpha_2$

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

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

Se você comparar essas duas definições cuidadosamente descobrirá que elas diferem apenas na última especificação da 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 frase simples parecem possuir a mesma falta de utilidade prática que a de um manipulador, mas isso não é verdade. Considere como a frase está relacionada à árvore de análise. Ela é a cadeia consistindo de todas as folhas da árvore de análise parcial cuja raiz é um vértice interno particular da árvore de análise completa. Uma frase simples é uma frase que necessita um único passo na derivação a partir de seu vértice não-terminal. Em termos de uma árvore de análise, 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 um único nível da árvore. Considere a árvore de análise mostrada na Figura 4.2



**FIGURA 4.2** Uma árvore de análise para  $E + T * id$ .

As folhas da árvore de análise na Figura 4.2 incluem a forma sentencial  $E + T * id$ . Uma vez que existem três vértices internos, existem três frases. Cada vértice interno é a raiz de uma subárvore cujas folhas são uma frase. O vértice raiz da árvore completa, E, gera todas as formas sentenciais resultantes,  $E + T * id$ , que é uma frase. O vértice interno T gera as folhas  $T * id$ , que é outra frase. Finalmente, o vértice interno F gera  $id$ , que também é uma frase. Assim, as frases da forma sentencial  $E + T * id$  são  $E + T * id$ ,  $T * id$  e  $id$ . Note que as frases não são necessariamente LDs na gramática subjacente.

As frases simples são um subconjunto das frases. No exemplo acima, a única frase simples é  $id$ .

A razão de se discutir frase e frase simples é esta: o manipulador de uma forma sentencial mais à direita é a frase simples mais à esquerda. Temos agora uma maneira

altamente intuitiva para descobrir o manipulador de qualquer forma sentencial à direita, desde que tenhamos a gramática e desenhemos a árvore de análise. Essa abordagem para encontrar manipuladores obviamente não é prática para um analisador (se você já tem uma árvore de análise, por que necessitaria de um analisador?). Seu único propósito é fornecer ao leitor um sentimento intuitivo sobre o que é um manipulador em relação a uma árvore de análise, o que acreditamos ser mais fácil do que tentar pensar sobre os manipuladores em termos de formas sentenciais.

Podemos agora pensar a análise Baixo-Cima em termos de árvore de análise, embora sabendo que o objetivo do analisador é produzir a árvore. Dada a árvore de análise para uma sentença inteira, você pode facilmente encontrar o manipulador, que é a primeira coisa a ser reescrita na sentença para obter a forma sentencial anterior. Então, o manipulador pode ser extraído da árvore de análise e o processo é repetido. Continuando em direção à raiz da árvore de análise a derivação mais à direita pode ser construída.

#### 4.5.2 Algoritmos Desloca-Reduz

Os analisadores Baixo-Cima são freqüentemente chamados *algoritmos desloca-reduz* porque deslocamento e redução constituem as ações mais comuns que eles especificam. A ação de deslocamento move o próximo símbolo da entrada na pilha do analisador. Uma ação de redução substitui um LD no topo da pilha pelo seu correspondente LE. Um analisador de qualquer espécie é um autômato à pilha. Você não precisa ter intimidade com os autômatos para entender como um analisador Baixo-Cima funciona, embora isso ajude. Um autômato à pilha é uma máquina matemática muito simples que faz uma varredura nas cadeias de símbolos, da esquerda para a direita. Ele é assim chamado porque usa uma pilha como memória. Ele pode ser utilizado como um reconhecedor de linguagem. Dada uma cadeia de símbolos pertencentes a algum alfabeto, o autômato projetado para esse fim determina se a cadeia é ou não uma sentença da linguagem. No processo, ele pode produzir a informação necessária para a construção da árvore de análise para a sentença.

A cadeia de entrada será examinada, um símbolo de cada vez, da esquerda para a direita. A entrada é tratada como se estivesse armazenada em outra pilha, porque o autômato nunca enxerga além do símbolo mais à esquerda na cadeia.

#### 4.5.3 Analisadores ED

Muitos algoritmos de análise baixo-cima foram inventados, a maioria deles são variações do processo chamado ED, no qual o E significa que a varredura na cadeia de entrada é da esquerda para a direita e o D significa que ele produz uma derivação mais à direita. Os analisadores ED usam uma quantidade relativamente pequena de código e uma tabela de análise. O algoritmo original foi inventado por Donald Knuth, que o publicou em 1965. Esse algoritmo, que é às vezes chamado *ED canônico*, não foi usado nos anos imediatamente seguintes à publicação porque a produção da tabela de análise exigia grande quantidade de tempo computacional e de memória. A partir daí, variações no processo de construção da tabela foram desenvolvidas. Elas se caracterizam por duas propriedades: (1) elas requerem muito menos recursos computacionais para produzir a tabela de análise, e (2) elas trabalham com classes de gramática menores que as do algoritmo canônico.

Existem várias vantagens no analisador ED:

1. Ele funciona em praticamente todas as gramáticas usadas para descrever linguagens de programação.
2. Ele funciona em uma classe maior de gramáticas do que os outros algoritmos baixo-cima, mas é tão eficiente quanto qualquer outro analisador baixo-cima.
3. Ele consegue detectar erros de sintaxe tão logo isso seja possível.
4. A classe de gramáticas ED engloba a dos analisadores EE.

A única desvantagem da análise ED é a dificuldade de produzir manualmente a tabela de análise para uma dada gramática. Isso não é desvantagem séria, contudo, pois existem muitos programas disponíveis que tomam a gramática como entrada e produzem a tabela de análise (Aho, et al., 1986).

Antes do aparecimento do algoritmo de análise ED havia algoritmos que encontravam os manipuladores das formas sentenciais à direita olhando à esquerda e à direita da subcadeia da forma sentencial suspeita de ser o manipulador. A descoberta de Knuth foi que você poderia olhar efetivamente à esquerda do manipulador suspeito sempre na base da pilha de análise, para determinar se ele era o manipulador correto. Mas toda informação na pilha relevante ao processo de análise poderia ser representada em um único estado, que poderia ficar no topo da pilha. Em outras palavras, Knuth descobriu que independentemente do comprimento da cadeia de entrada, do comprimento da forma sentencial ou da profundidade da pilha, haveria um número pequeno de situações diferentes na forma como o processo de análise é entendido. Cada situação poderia ser representada por um estado e guardada na pilha de análise, um estado para cada símbolo gramatical na pilha. No topo da pilha estaria sempre um símbolo de estado que representaria a informação relevante da história completa da análise até aquele momento. Usaremos S maiúsculo com subscrito para representar os estados do analisador.

A Figura 4.3 mostra a estrutura de um analisador ED.

O conteúdo da pilha de análise para um analisador ED possui a seguinte forma:

$S_0 X_1 S_1 X_2 \dots X_m S_m$  (topo)

em que os Ss são símbolos de estado e os Xs são símbolos gramaticais. Uma configuração de analisador ED é um par de cadeias (pilha, entrada) com a forma detalhada

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

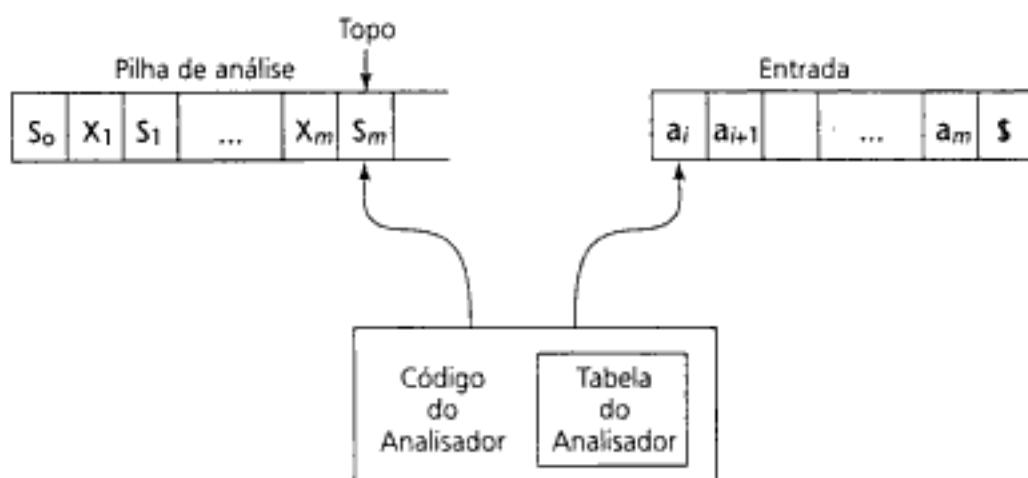


FIGURA 4.3 A estrutura de um analisador ED.

Note que a cadeia de entrada tem um cífrão em seu fim no lado direito. Ele é colocado ali durante a inicialização e é usado para o término normal do analisador. Usando essa configuração para o analisador, podemos definir formalmente o processo de análise ED, que é baseado na tabela de análise. A tabela possui duas partes, chamadas AÇÃO e Vá\_Para. A primeira especifica o que o analisador faz, tem símbolos de estado como rótulos de suas linhas e símbolos terminais da gramática como rótulos de suas colunas. Dado um estado corrente de análise, que é representado pelo símbolo de estado no topo da pilha e o próximo símbolo da entrada, a tabela especifica o que o analisador deve fazer.

As duas principais ações do analisador são deslocar e reduzir. O analisador ou desloca o próximo símbolo de entrada na pilha de análise ou ele já tem o manipulador no topo da pilha, que é então reduzido para o LE da regra cujo LD seja o mesmo que o manipulador. Duas outras ações são possíveis; aceitar, que significa que a análise da entrada foi completada com sucesso, e erro, que significa que o analisador encontrou um erro de sintaxe.

As linhas da parte Vá\_Para de uma tabela de análise têm símbolos de estado como rótulos. Essa parte da tabela tem não-terminais como rótulos de coluna. Os valores na parte Vá\_Para da tabela indicam que símbolos de estado devem ser empilhados após a redução ter se completado, o que significa que o manipulador foi removido da pilha de análise e o novo não-terminal foi empilhado. O símbolo específico é encontrado na linha cujo rótulo é o símbolo de estado no topo da pilha após a remoção do manipulador e de seus símbolos associados. A coluna da tabela Vá\_Para que é usada é aquela com o rótulo que é o LE da regra usada na redução.

A configuração inicial de um analisador ED é

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

As ações do analisador são definidas como segue:

- Se AÇÃO  $[S_m, a_i] = \text{Desloca } S$ , 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 deslocar é simples. O próximo símbolo da entrada é empilhado juntamente com o símbolo de estado que é parte da especificação Deslocar na tabela AÇÃO.

- Se AÇÃO  $[S_m, a_i] = \text{Reduz } A \rightarrow b \text{ e } S = \text{Vá_Para } [S_{m-r}, A]$ , onde  $r$  é o comprimento de  $\beta$ , a próxima configuração é

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

Essa é uma ação muito mais complicada. Para uma ação reduzir, o manipulador deve ser retirado da pilha. Uma vez que cada símbolo da gramática na pilha possui um símbolo de estado, o número de símbolos removidos da pilha é o dobro do número de símbolos no manipulador. Depois de remover o manipulador e seus símbolos de estado associados, o LE da regra é empilhado. Finalmente, a tabela Vá\_Para é usada, com o rótulo da linha sendo o símbolo que ficou exposto ao se remover o manipulador e seus símbolos de estado, e o rótulo da coluna sendo o não-terminal que é o LE da regra usada na redução. Assim, na nova configuração, o símbolo de cima vem da parte Vá\_Para da tabela e o símbolo gramatical mais acima é o LE da regra usada na redução.

- Se AÇÃO  $[S_m, a_i] = \text{Aceitar}$ , a análise está completa e não foi detectado erro.
- Se AÇÃO  $[S_m, a_i] = \text{Erro}$ , o analisador chama uma rotina para tratamento de erro.

Embora existam muitos algoritmos de análise baseados no conceito ED, eles diferem apenas na construção da tabela de análise. Todos analisadores ED usam o mesmo algoritmo de análise.

Talvez a melhor maneira de se familiarizar com o processo de análise ED seja através de um exemplo. Considere a gramática clássica para expressões aritméticas que se segue:

1.  $E \rightarrow E + T$
2.  $E \rightarrow T$
3.  $T \rightarrow T * T$
4.  $T \rightarrow F$
5.  $F \rightarrow ( E )$
6.  $F \rightarrow id$

As regras dessa gramática são numeradas, de modo que apenas um número deve aparecer na tabela de análise.

A Figura 4.4 (p. 172) mostra a tabela de análise ED para essa gramática. Note que usamos abreviações para as ações e nos referimos aos estados por números. R4 significa reduzir usando a regra 4. S6 significa deslocar o próximo símbolo de entrada na pilha e empilhar o estado 6. Posições vazias na tabela AÇÃO indicam erro de sintaxe. Em um analisador completo, elas poderiam conter chamadas para rotinas de tratamento de erro.

A seguir está o rastreamento de uma análise da cadeia  $id + id * id$ , usando o algoritmo de análise ED e a tabela mostrada na Figura 4.4.

Pilha	Entrada	Ação
0	$id + id * id \$$	Desloca 5
0 id 5	$+ id * id \$$	Reduz 6 (usa Vá_Para [0, F])
0 F 3	$+ id * id \$$	Reduz 4 (usa Vá_Para [0, T])
0 T 2	$+ id * id \$$	Reduz 2 (usa Vá_Para [0, E])
0 E 1	$+ id * id \$$	Desloca 6
0 E 1 + 6	$id * id \$$	Desloca 5
0 E 1 + 6 id 5	$* id \$$	Reduz 6 (usa Vá_Para [6, F])
0 E 1 + 6 F 3	$* id \$$	Reduz 4 (usa Vá_Para [6, T])
0 E 1 + 6 T 9	$* id \$$	Desloca 7
0 E 1 + 6 T 9 * 7	$id \$$	Desloca 5
0 E 1 + 6 T 9 * 7 id 5	$\$$	Reduz 6 (usa Vá_Para [7, F])
0 E 1 + 6 T 9 * 7 F 10	$\$$	Reduz 3 (usa Vá_Para [6, T])
0 E 1 + 6 T 9	$\$$	Reduz 1 (usa Vá_Para [0, E])
0 E 1	$\$$	Aceita

Os algoritmos para gerar a tabela de análise ED para uma gramática dada são descritos em Aho et. al. (1986). Existem diferentes sistemas de software disponíveis para gerar tabelas de análise ED. Um dos mais antigos é o sistema yacc, que normalmente integra o UNIX.

Estado	id	AÇÃO						Vá_Para		
		+	*	(	)	\$		E	T	F
0	S5		S4					1	2	3
1		S6				aceita				
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 4.4** Tabela de análise ED para uma gramática de expressões aritméticas.

## RESUMO

A análise sintática é uma parte comum às linguagens, independentemente da abordagem usada na implementação. Ela é normalmente baseada em uma descrição formal da linguagem que está sendo implementada. O mecanismo mais comum para descrever a sintaxe é uma gramática livre de contexto, também chamada BNF. A tarefa da análise sintática é usualmente dividida em duas partes: análise léxica e análise sintática. Existem várias razões substanciais para separar a análise léxica: simplicidade, eficiência e portabilidade.

Um analisador léxico é um “casamenteiro” de padrões que isola as partes de pequena escala de um programa, chamadas lexemas. Esses ocorrem em categorias, tais como literais inteiros e nomes, chamadas de símbolos. A cada símbolo é atribuído um código numérico que, juntamente com o lexema, é o que o analisador léxico produz. Existem três abordagens distintas para construir um analisador léxico: usar uma ferramenta de software para gerar a tabela para um analisador dirigido por tabela, construir essa tabela manualmente e escrever um código para implementar uma descrição do diagrama de estados dos símbolos da linguagem que está sendo implementada. O diagrama de estado para os símbolos pode ser razoavelmente pequeno se forem usadas classes de caracteres para transição, em vez de haver transição para cada possível caráter de cada vértice de estado. O uso de uma pesquisa em tabela para reconhecer palavras reservadas simplifica o diagrama de estado.

Os analisadores sintáticos têm duas metas: detectar erros de sintaxe em um determinado programa e produzir a árvore de análise ou a informação requerida para construir tal árvore, para um programa dado. Eles são cima-baixo, significando que constroem derivações mais à esquerda e uma árvore de análise na ordem de cima para baixo; ou baixo-cima,

caso em que constroem o inverso de uma derivação mais à direita e uma árvore de análise na ordem de baixo para cima. Analisadores que trabalham com gramáticas não-ambíguas gerais têm complexidade  $O(n^3)$ . Contudo, os analisadores usados para implementação de linguagens de programação trabalham em subclasses das gramáticas não-ambíguas e têm complexidade  $O(n)$ .

Um analisador descendente recursivo é um analisador EE que é implementado escrevendo-se o código diretamente da gramática da linguagem fonte. A EBNF é ideal como base para os analisadores descendentes recursivos, os quais tem um subprograma para cada não-terminal na gramática. O código para uma dada regra gramatical é simples se ela possui um único LD. O LD é examinado da esquerda para a direita. Para cada não-terminal, o código chama o subprograma associado àquele não-terminal, que analisa o resultado gerado pelo não-terminal. Para cada terminal, o código o compara com o próximo símbolo da entrada. Se eles coincidem, o código simplesmente chama o analisador léxico para obter o novo símbolo. Caso contrário, o subprograma sinaliza erro de sintaxe. Se uma regra possui mais de um LD, o subprograma deve primeiro determinar qual LD ele deve analisar. Tem de ser possível fazer essa determinação baseando-se no próximo símbolo da entrada.

Duas características distintas da gramática impedem a construção de um analisador descendente recursivo baseado na gramática. Uma delas é a recursão à esquerda. Embora não tenha sido coberto aqui, existe um algoritmo para remover de uma gramática a recursão à esquerda tanto direta como indireta. O outro problema é detectado com o teste da disjunção paritária, que testa se um subprograma de análise pode determinar qual LD está sendo analisado tomando por base o próximo símbolo da entrada. Algumas gramáticas podem ser modificadas para passar no teste usando fatoração à esquerda.

O problema dos analisadores baixo-cima é encontrar a subcadeia da forma sentencial corrente que deve ser reduzida a seu LE associado para obter a próxima (anterior) forma sentencial na derivação mais à direita. Essa subcadeia é chamada manipulador da forma sentencial. As árvores de análise podem ser usadas para esclarecer a definição de um manipulador. Um analisador baixo-cima é um algoritmo desloca-reduz porque, na maioria dos casos, ele desloca o próximo lexema da entrada na pilha de análise ou reduz o manipulador que está no topo da pilha.

A família ED de analisadores desloca-reduz é a abordagem mais comumente usada em analisadores baixo-cima para linguagens de programação porque os analisadores dessa família têm várias vantagens em relação aos alternativos. Um analisador ED usa uma pilha de análise, que contém símbolos gramaticais e símbolos de estado para manter o estado do analisador. O símbolo mais acima na pilha é sempre um símbolo de estado que representa toda a informação na pilha que é relevante ao processo de análise. Os analisadores ED usam duas tabelas de análise; AÇÃO e Vá\_Para. A parte AÇÃO especifica o que o analisador deve fazer, dado o símbolo de estado no topo da pilha de análise e o próximo símbolo da entrada. A tabela Vá\_Para é usada para determinar qual símbolo de estado deve ser colocado na pilha de análise após a 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 símbolo (token).
- Quais são as principais tarefas de um analisador léxico?
- Descreva brevemente as três abordagens para a construção de um analisador léxico.
- O que é diagrama de transição de estados?

7. Por que são usadas classes de caracteres em vez de caracteres individuais nas transições de um diagrama de estado em um analisador léxico?
8. Quais são as duas metas distintas da análise sintática?
9. Descreva as diferenças entre analisadores cima-baixo e baixo-cima.
10. Descreva o problema de análise para um analisador cima-baixo.
11. Descreva o problema de análise para um analisador baixo-cima.
12. Explique por que os compiladores usam algoritmos de análise que funcionam somente em um subconjunto de todas as gramáticas.
13. Por que são usadas constantes nomeadas em vez de números para os códigos de símbolos?
14. Descreva como um subprograma analisador descendente recursivo é escrito para uma regra com um único LD.
15. Explique as duas características da gramática que proíbem seu uso como base para um analisador cima-baixo.
16. Qual é o conjunto PRIMEIRO para uma dada gramática e uma dada forma sentencial?
17. Descreva o teste da disjunção paritária.
18. O que é fatoração à esquerda?
19. O que é frase de uma forma sentencial?
20. O que é frase simples de uma forma sentencial?
21. O que é manipulador de uma forma sentencial?
22. Qual é a máquina matemática na qual os analisadores baixo-cima se baseiam?
23. Descreva três vantagens dos analisadores ED.
24. Qual foi a descoberta de Knuth ao desenvolver a técnica de análise ED?
25. Descreva o propósito da tabela AÇÃO de um analisador ED.
26. Descreva o propósito da tabela VÁ\_PARA de um analisador ED.
27. A recursão à esquerda é problema para os analisadores ED?

## PROBLEMAS

1. Faça um diagrama de estado que reconheça uma forma de comentários das linguagens baseadas no C, aquela que começa com /\* e termina com \*/.
2. Faça um diagrama de estado para reconhecer literais de números reais na sua linguagem de programação favorita.
3. Escreva e teste o código para implementar o diagrama de estado do Problema 1.
4. Escreva e teste o código para implementar o diagrama de estado do Problema 2.
5. Para as seguintes regras gramaticais, faça o teste de disjunção paritária. Para aquelas que passarem no teste, escreva um subprograma analisador recursivo descendente que analisa a linguagem gerada pelas regras. Assuma que tenha disponível um analisador léxico chamado `lex` e um subprograma para manipulação de erros denominado `erro`, chamado sempre que um erro de sintaxe é detectado.
  - $A \rightarrow aB \mid b \mid BB$
  - $A \rightarrow aB \mid bA \mid aBb$
  - $A \rightarrow aaA \mid b \mid caB$
6. Dada a seguinte gramática e a forma sentencial à direita, desenhe a árvore de análise e mostre as frases e frases simples, bem como o manipulador.  
 $S \rightarrow aAb \mid aBA \quad A \rightarrow ab \mid aAB \quad B \rightarrow bB \mid b$ 
  - $aaabbb$
  - $bbBaabB$
  - $aaBbBb$
7. Mostre uma análise completa, incluindo o conteúdo da pilha de análise, a cadeia de entrada e a ação para a cadeia  $id^* (id + id)$ , usando a gramática e a tabela de análise da Seção 4.5.3.
8. Mostre uma análise completa, incluindo o conteúdo da pilha de análise, a cadeia de entrada e a ação para a cadeia  $(id + id)^* id$ , usando a gramática e a tabela de análise da Seção 4.5.3.

## Capítulo 5

# Nomes, Vinculações, Verificação de Tipos e Escopos



### Bjarne Stroustrup

Bjarne Stroustrup, um membro do staff técnico da AT&T Bell Labs desde 1979, projetou a linguagem C++ que tornou a programação orientada a objeto acessível e disponível a desenvolvedores de software do mundo real. Em grande parte devido à popularidade do C++, a programação orientada a objeto tornou-se o paradigma de desenvolvimento universal no início da década de 90. Stroustrup é o autor de *A Linguagem de Programação C++* e *The Design and Evolution of C++*.

- 5.1** Introdução
- 5.2** Nomes
- 5.3** Variáveis
- 5.4** O Conceito de Vinculação
- 5.5** Verificação de Tipos
- 5.6** Tipificação Forte
- 5.7** Compatibilidade de Tipos
- 5.8** Escopo
- 5.9** Escopo e Tempo de Vida
- 5.10** Ambientes de Referenciamento
- 5.11** Constantes Nomeadas
- 5.12** Inicialização de Variáveis

Este capítulo apresenta as questões semânticas fundamentais das variáveis. O mais básico destes tópicos será abordado primeiro: a natureza dos nomes e das palavras especiais nas linguagens de programação. Os atributos das variáveis, incluindo tipo, endereço e valor também serão discutidos. A questão dos apelidos (*aliases*) será incluída nesta discussão. Os importantes conceitos de vinculação e de tempos de vinculação serão apresentados. Os diferentes tempos de vinculação possíveis para os atributos de variáveis definem quatro diferentes categorias de variáveis. Suas descrições são seguidas de uma minuciosa investigação sobre a verificação de tipos, tipificação forte e regras de compatibilidade de tipos. Duas regras muito diferentes de determinação do escopo para nomes, a estática e a dinâmica, também são discutidas, juntamente com o conceito de ambiente de referenciamento de uma instrução. Finalmente, serão descritas técnicas de constantes nomeadas e de inicialização de variáveis.

## 5.1 Introdução

---

As linguagens de programação imperativas são, em graus variados, abstrações da arquitetura do computador de von Neumann subjacente. Dois dos componentes principais da arquitetura são a sua memória, que armazena tanto instruções como dados, e o 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 são muito próximas das características das células; um exemplo disso é uma variável inteira, que, normalmente, é representada exatamente como uma palavra individual de memória do hardware. Em outros casos, as abstrações estão afastadas das células, como acontece com uma matriz tridimensional, que requer uma função de mapeamento de software para suportar a abstração.

Uma variável pode ser caracterizada por um conjunto de propriedades, ou de atributos, sendo o mais importante o tipo, um conceito fundamental nas linguagens de programação. O projeto dos tipos de dados de uma linguagem exige que uma variedade de questões seja levada em consideração. Entre as mais importantes estão o escopo e o tempo de vida das variáveis. Relacionam-se com elas as questões de verificação de tipos e de inicialização. O conhecimento de todos esses conceitos é um requisito para entender as linguagens imperativas. A compatibilidade de tipos é outra parte importante no projeto de tipos de dados de uma linguagem.

No restante do livro, freqüentemente nos referiremos a famílias de linguagens como se elas fossem uma única linguagem. Por exemplo, quando nos referirmos ao FORTRAN, queremos dizer todas as versões dele. Isso também acontece com o Pascal e com a Ada. As referências ao C incluem a versão original dele e do ANSI C. Quando nos referirmos a uma versão específica de uma linguagem, é porque ela é diferente dos outros membros da família.

## 5.2 Nomes

---

Antes de podermos iniciar nossa discussão a respeito das variáveis, precisamos discutir um dos seus atributos fundamentais — os nomes, que têm um uso mais amplo do que simplesmente para variáveis. Os nomes também estão associados a rótulos, a subprogramas, a

parâmetros formais e a outras construções de programa. O termo *identificador* é muitas vezes usado de forma intercambiável com *nome*.

### 5.2.1 Questões de Projeto

A seguir, constam as principais questões de projeto para nomes:

- Qual é o tamanho máximo de um nome?
- Caracteres de conexão podem ser usados em nomes?
- Os nomes fazem distinção entre maiúsculas e minúsculas?
- As palavras especiais são palavras reservadas ou palavras-chave?

Tais questões serão discutidas nas duas seções seguintes, que também incluem exemplos de diversas opções de projeto.

### 5.2.2 Formas de Nomes

Um **nome** é uma cadeia de caracteres usada para identificar alguma entidade de um programa. As primeiras linguagens de programação usavam nomes de um único caractere. Isso era natural porque a programação primitiva era fundamentalmente matemática, e os matemáticos há muito tempo usam nomes de um único caractere para incógnitas em suas notações formais.

O FORTRAN I rompeu com a tradição do nome de um único caractere, permitindo até seis em seus nomes. O FORTRAN 77 ainda restringe os nomes a seis caracteres, mas o FORTRAN 90 e o C permitem até 31; os nomes em Java e Ada não têm limite de tamanho, e todos são significativos. Algumas linguagens, como por exemplo o C++, não especificam um limite de tamanho para os nomes, ainda que os implementadores dessas linguagens, às vezes, o façam. Eles optam, por isso, para que a tabela de símbolos em que os identificadores são armazenados durante a compilação não precise ser muito grande e, também, para simplificar a sua manutenção.

A forma de nomes comumente aceitável é uma cadeia com um limite de tamanho razoavelmente longo, se for o caso, com algum caractere de conexão, como um sublinhado (\_) incluído, servindo ao mesmo propósito que um espaço no texto em português, mas sem finalizar a cadeia do nome em que ele é colocado. As linguagens contemporâneas permitem caracteres de conexão em nomes.

Em algumas linguagens, como o C, o C++ e o Java, letras maiúsculas e minúsculas em nomes são distintas; ou seja, nomes nessas linguagens **fazem distinção entre maiúsculas e minúsculas**. Por exemplo, os três nomes seguintes são distintos em C++: rosa, ROSA e Rosa. Para alguns, esse é um sério prejuízo à legibilidade, porque nomes muito semelhantes denotam, de fato, entidades diferentes. Nesse sentido, a distinção entre maiúsculas e minúsculas viola o princípio de projeto segundo o qual construções de linguagem de aparência semelhante devem ter o mesmo significado.

Evidentemente, nem todo mundo concorda que a distinção entre maiúsculas e minúsculas é ruim para os nomes. No C, este problema pode ser evitado por meio do uso exclusivo de minúsculas para nomes. Em Java, entretanto, não é possível evitá-lo porque muitos dos nomes predefinidos incluem as duas formas. Por exemplo, o método Java para converter uma cadeia em um valor inteiro é `parseInt`, e grafias como `ParseInt` e `parseInt` não são reconhecidas. Esse é um problema de capacidade de escrita, não de legibilidade, porque a necessidade de lembrar grafias estranhas dificulta a escrita correta de

programas. É um tipo de intolerância da parte do projetista da linguagem que é reforçada pelo compilador.

Em versões do FORTRAN anteriores ao 90, somente letras maiúsculas podiam ser usadas em nomes, o que era uma restrição desnecessária. A origem disso é que as máquinas perfuradoras de cartão tinham apenas letras maiúsculas. À semelhança do FORTRAN 90, muitas implementações da versão 77 permitem letras minúsculas; elas simplesmente as convertem para maiúsculas para uso interno durante a compilação.

### 5.2.3 Palavras Especiais

As palavras especiais nas linguagens de programação são usadas para tornar os programas mais legíveis ao dar nome a ações que devem ser executadas. Elas também são usadas para separar as entidades sintáticas dos programas. Na maioria das linguagens, tais palavras são classificadas como reservadas, mas, em algumas, são somente palavras-chave.

Uma **palavra-chave** de uma linguagem de programação é especial somente em certos contextos. As palavras especiais no FORTRAN são palavras-chave. Nele, REAL, quando encontrada no início de uma instrução e seguida de um nome, é considerada uma palavra-chave que indica uma instrução declarativa. Porém, se a palavra REAL for seguida do operador de instrução, ela será considerada um nome de variável. Esses dois usos são ilustrados a seguir:

```
REAL APPLE
REAL = 3.4
```

Os compiladores FORTRAN e os leitores dos programas devem reconhecer a diferença entre nomes e palavras especiais pelo contexto.

Uma **palavra reservada** é especial e não pode ser usada como um nome. Como uma opção de projeto de uma linguagem, as palavras reservadas são melhores do que as palavras-chave porque a capacidade de redefiní-las pode acarretar problemas de legibilidade. Por exemplo, no FORTRAN, poderíamos ter as instruções

```
INTEGER REAL
REAL INTEGER
```

que declaram ser a variável de programa REAL do tipo INTEGER e a variável INTEGER do tipo REAL. Além da aparência estranha dessas instruções de declaração, a presença de REAL e de INTEGER como nomes de variáveis em outra parte do programa poderia ser enganosa para os leitores.

Nos exemplos de código de programa deste livro, as palavras reservadas são apresentadas em negrito.

Muitas linguagens incluem nomes predefinidos, que estão, em certo sentido, entre palavras reservadas e nomes definidos pelo usuário. Eles têm significados predefinidos, mas podem ser redefinidos pelo usuário. Por exemplo, os nomes dos tipos de dados incorporados da Ada, como INTEGER e FLOAT, são predefinidos. Esses nomes não são reservados; eles podem ser redefinidos por qualquer programa Ada.

As definições dos nomes predefinidos no Pascal e na Ada devem ser visíveis aos seus compiladores devido à verificação de tipos ser feita durante a compilação. Em ambas as linguagens, os exemplos de nomes predefinidos acima são implicitamente visíveis ao compilador. Na Ada, outros nomes predefinidos, como os subprogramas de entrada e saída padrão, GET e PUT, tornam-se explicitamente visíveis pelas instruções **with** escritas pelo usuário.

No C e no C++, muitos nomes são predefinidos em bibliotecas usadas pelos programas do usuário. Por exemplo, os nomes de função de entrada e saída C, `printf` e `scanf`, são definidos na biblioteca `stdio`. O acesso aos nomes predefinidos em bibliotecas é colocado à disposição do compilador por meio de arquivos de cabeçalhos, que contêm declarações para nomes definidos em bibliotecas.

## 5.3 Variáveis

Uma variável de programa é uma abstração de uma célula ou de um conjunto de células de memória do computador. Os programadores freqüentemente pensam nas variáveis como nomes para localizações da memória, mas há muito mais coisas nelas do que apenas um nome. A mudança das linguagens de máquina para as linguagens de montagem foi, em grande parte, uma substituição dos endereços de memória numéricos por nomes, tornando os programas bem mais legíveis e, portanto, mais fáceis de serem escritos e mantidos. Esse passo também proporcionou uma saída do problema do endereçamento absoluto, porque o tradutor que convertia os nomes para os endereços reais também os escolhia.

Uma variável pode ser caracterizada como um sétuplo de atributos: nome, endereço, valor, tipo, tempo de vida e escopo. Isso constitui a maneira mais clara de explicar os vários aspectos das variáveis, ainda que possa parecer demasiadamente complicado para um conceito aparentemente tão simples.

Nossa discussão a respeito dos atributos das variáveis nos levará a examinar os importantes conceitos relacionados dos apelidos, da vinculação, dos tempos de vinculação, das declarações, da verificação de tipos, da tipificação forte, das regras de escopo e dos ambientes de referenciamento.

Os atributos de nome, de endereço, de tipo e de valor das variáveis serão discutidos nas subseções seguintes. Os atributos de tempo de vida e de escopo serão discutidos nas Seções 5.4.3 e 5.8, respectivamente.

### 5.3.1 Nome

Nomes de variáveis são os nomes mais comuns nos programas. Eles foram extensamente discutidos na Seção 5.2 no contexto geral dos nomes de entidade em programas. A maioria das variáveis tem nomes. Aquelas que não têm serão discutidas na Seção 5.4.3.3. Os nomes freqüentemente são chamados de identificadores.

### 5.3.2 Endereço

O **endereço** de uma variável é o mesmo da memória à qual ela está associada. Essa associação não é tão simples como poderia parecer a princípio. Em muitas linguagens, é possível que o mesmo nome seja associado a diferentes endereços em diferentes lugares e em diferentes tempos no programa. Por exemplo, um programa pode ter dois subprogramas, `sub1` e `sub2`, cada um dos quais define uma variável que usa o mesmo nome, digamos `soma`. Uma vez que essas duas variáveis são independentes uma da outra, uma referência a `soma` em `sub1` não se relaciona com uma referência a `soma` em `sub2`. Similarmente, a maioria das linguagens permite que o mesmo nome seja associado a diferentes endereços

diferentes tempos durante a execução do programa. Por exemplo, um programa chamado recursivamente tem múltiplas versões de cada variável declarada localmente, uma para cada ativação. O processo de associar variáveis a endereços será discutido adicionalmente na Seção 5.4.3. Um modelo de implementação para subprogramas e para suas ativações para linguagens assemelhadas ao ALGOL será discutido no Capítulo 10.

O endereço de uma variável, às vezes, é chamado de seu **valor-l**, porque é isso que se exige quando a variável aparece no lado esquerdo (*left side*) de uma instrução de atribuição.

### 5.3.2.1 Apelidos

É possível fazer com que múltiplos identificadores façam referência ao mesmo endereço. Quando mais de um nome de variável pode ser usado para acessar uma única localização de memória, os nomes são chamados apelidos. A criação de apelidos é um problema para a legibilidade, porque permite que uma variável tenha seu valor modificado por uma atribuição a uma variável diferente. Por exemplo, se as variáveis **A** e **B** forem apelidos, qualquer mudança em **A** também modificará **B** e vice-versa. Um leitor do programa sempre deve lembrar que **A** e **B** são nomes diferentes para a mesma célula da memória. Na prática, isso é muito difícil, uma vez que pode haver qualquer número de apelidos em um programa. O uso de apelidos também torna a verificação de programas mais difícil.

Os apelidos também podem ser criados de diversas maneiras diferentes. No FORTRAN, podem ser explicitamente criados com a instrução **EQUIVALENCE**. Eles também podem ser criados usando-se as estruturas de registro variantes de algumas linguagens, incluindo Pascal e Ada, e usando-se os tipos de união do C e do C++. Os apelidos criados por esses tipos de dados destinam-se a poupar espaço de armazenamento, permitindo que as mesmas localizações sejam usadas por diferentes tipos de dados em diferentes tempos. Também podem ser usados para contornar as regras de tipo de algumas das linguagens em que são oferecidos. Os registros variantes e as uniões serão discutidos detalhadamente no Capítulo 6.

Duas variáveis de ponteiro são apelidos quando apontam para a mesma localização de memória. O mesmo é verdadeiro em relação às variáveis de referência. Esse tipo de múltiplos nomes não se destina a conservar espaço de armazenamento, mas é simplesmente 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. Essa e outras características dos ponteiros e das referências serão discutidas no Capítulo 6.

Os apelidos podem ser criados em muitas linguagens por meio de parâmetros de subprograma. Esses tipos de apelidos serão discutidos no Capítulo 9.

Algumas das justificativas para os apelidos não existem mais. Quando uma construção de linguagem cria apelidos com o propósito de reutilizar o espaço de armazenamento, ele pode ser substituído por um esquema de gerenciamento dinâmico de armazenamento, o que permite a reutilização do espaço de armazenamento, mas não os criará necessariamente. Além disso, a memória dos computadores é bem maior agora do que quando linguagens como o FORTRAN foram desenvolvidas; assim, agora a memória não é um bem tão escasso.

O momento em que uma variável associa-se a um endereço é muito importante para um entendimento das linguagens de programação. Esse assunto será discutido na Seção 5.4.3.

### 5.3.3 Tipo

O **tipo** de uma variável determina a faixa de valores que ela pode ter e o conjunto de operações definidas para os valores do tipo. Por exemplo, o tipo `INTEGER` de algumas implementações FORTRAN especifica uma faixa de valores de -32.768 a 32.767 e operações aritméticas para adição, para subtração, para multiplicação, para divisão e para expo-  
nenciação, juntamente com algumas funções de biblioteca para operações como valor absoluto.

### 5.3.4 Valor

O valor de uma variável é o conteúdo da célula ou das células de memória associadas à variável. É conveniente imaginarmos a memória do computador em termos de células *abstratas*, em vez de físicas. As células ou unidades individualmente endereçáveis da maioria das memórias de computador contemporâneas são do tamanho de bytes, tendo cada um destes usualmente oito bits de tamanho, o que é muito pequeno para a maioria das variá-  
veis de programa. Definimos uma célula de memória abstrata para que tenha o tamanho exigido pela variável a que ela está associada. Por exemplo, ainda que os valores de núme-  
ros reais possam ocupar quatro bytes físicos em uma implementação particular de uma linguagem particular, imaginamos um valor real como se estivesse ocupando uma única célula de memória abstrata. Consideramos que o valor de cada tipo simples não-estrutura-  
da ocupa uma única célula abstrata. A partir de agora, quando usarmos o termo *célula de memória*, queremos dizer célula de memória abstrata.

O valor de uma variável, às vezes, é chamado de **valor-r**, pois é o que se exige quando ela é usada no lado direito (*right side*) de uma instrução de atribuição. Para acessar o **valor-r**, o **valor-l** deve ser determinado primeiro. Essas determinações nem sempre são simples. Por exemplo, as regras de escopo podem complicar muito as coisas, conforme discutiremos na Seção 5.8.

## 5.4 O Conceito de Vinculação

Em termos gerais, uma **vinculação** é uma associação, como, por exemplo, entre um atrí-  
buto e uma entidade ou entre uma operação e um símbolo. O momento em que uma vincu-  
lação se desenvolve é chamado de **tempo de vinculação**. Vinculação e tempos de  
vinculação são conceitos proeminentes na semântica das linguagens de programação. As  
vinculações podem acontecer no tempo de projeto da linguagem, no tempo de implemen-  
tação da linguagem, no tempo de compilação, no tempo de ligação (*link*), no tempo de carre-  
gamento ou no tempo de execução. Por exemplo, o símbolo de asterisco (\*) normalmente é  
vinculado à operação de multiplicação no tempo de projeto da linguagem. Um tipo de  
dados, como `INTEGER` no FORTRAN, é vinculado a uma variedade de valores possíveis no  
tempo de implementação da linguagem. No tempo de compilação, uma variável em um  
programa Java é vinculada a um tipo de dados particular. Uma chamada a um subprograma

de biblioteca é vinculada ao código do subprograma no tempo de ligação. Uma variável pode ser vinculada a uma célula de armazenamento quando o programa é carregado na memória. Essa mesma vinculação não acontece até o tempo de execução, em alguns casos, como acontece com variáveis declaradas em subprogramas Pascal e em funções C (se a definição não incluir o qualificador **static**).

Considere a seguinte instrução de atribuição C++, cuja variável **cont** foi definida da seguinte maneira:

```
int cont;
...
cont = cont + 5;
```

Algumas das vinculações e seus tempos de vinculação para as partes dessa instrução de atribuição são as seguintes:

- Conjunto dos tipos possíveis para **cont**: vinculado no tempo de projeto da linguagem.
- Tipo de **cont**: vinculado no tempo de compilação.
- Conjunto dos valores possíveis de **cont**: vinculado no tempo de projeto do compilador.
- Valor de **cont**: vinculado no tempo de execução com esta instrução.
- Conjunto dos significados possíveis para o símbolo do operador **+**: vinculado no tempo de definição da linguagem.
- Significado do símbolo do operador **+** nessa instrução: vinculado no tempo de compilação.
- Representação interna do literal 5: vinculada no tempo de projeto do compilador.

Um entendimento completo dos tempos de vinculação para os atributos de entidades de programa é um requisito prévio para compreender a semântica de uma linguagem de programação. Por exemplo, para entender o que um subprograma faz, é preciso entender como os parâmetros reais de uma chamada são vinculados aos parâmetros formais existentes em sua definição. Para determinar o valor atual de uma variável, talvez você precise saber quando a variável foi vinculada ao armazenamento.

#### 5.4.1 Vinculação de Atributos a Variáveis

Uma vinculação é **estática** se ocorrer antes do tempo de execução e permanecer inalterada ao longo da execução do programa. Se ela ocorrer durante a execução ou puder ser modificada no decorrer da execução de um programa, será chamada **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 endereço no qual a célula reside pode ser deslocada(o) para dentro e para fora da memória muitas vezes durante a execução do programa. Em certo sentido, essas variáveis são vinculadas e desvinculadas repetidamente. Entretanto, essas vinculações são mantidas pelo hardware do computador, e as alterações são invisíveis para o programa e para o usuário. Uma vez que elas não são importantes para a discussão, não nos preocuparemos com essas vinculações de hardware. O ponto fundamental é distinguir entre vinculações estáticas e dinâmicas.

## 5.4.2 Vinculações de Tipos

Antes que uma variável possa ser referenciada em um programa, ela deve ser vinculada a um tipo de dados. Os dois aspectos importantes dessa vinculação são a maneira como o tipo é especificado e quando a vinculação ocorre. Tipos podem ser especificados estaticamente por meio de alguma forma de declaração explícita ou implícita.

### 5.4.2.1 Declarações de variáveis

Uma **declaração explícita** é uma instrução em um programa que lista nomes de variáveis e especifica que elas são de um tipo particular. Uma **declaração implícita** é um meio de associar variáveis a tipos por convenções padrão em vez de por instruções de declaração. Nesse caso, a primeira ocorrência de um nome de variável em um programa constitui sua declaração implícita. Tanto as declarações explícitas como as implícitas criam vinculações estáticas a tipos.

A maioria das linguagens de programação projetada desde os meados da década de 60 exige declarações explícitas de todas as variáveis (a Perl e a ML são duas exceções). Diversas linguagens amplamente usadas cujos projetos iniciais foram feitos antes da década de 60, notavelmente o FORTRAN, a PL/I e o BASIC, têm declarações implícitas. Por exemplo, no FORTRAN, um identificador que aparece em um programa e não é explicitamente declarado fica implicitamente declarado de acordo com a seguinte convenção: se o identificador for iniciado com uma das letras I, J, K, L, M ou N, ele será implicitamente declarado como do tipo INTEGER; caso contrário, ele será implicitamente declarado como do tipo REAL.

Ainda que sejam uma pequena conveniência para os programadores, as declarações implícitas podem ser prejudiciais à legibilidade porque impedem que o processo de compilação detecte alguns erros tipográficos e do programador. Variáveis deixadas accidentalmente sem declarar pelo programador recebem tipos-padrão e atributos inesperados, que poderiam causar erros sutis difíceis de serem diagnosticados.

Alguns dos problemas com as declarações implícitas podem ser evitados exigindo-se que os nomes para tipos específicos iniciem com caracteres especiais particulares. Por exemplo, em Perl, qualquer nome que se inicie com \$ é uma escalar, que pode armazenar uma cadeia de caracteres ou um valor numérico; se o nome iniciar-se com @, ele será uma matriz; se iniciar com um %, será uma estrutura hash<sup>1</sup>. Isso cria diferentes espaços de nome para diferentes tipos de variáveis. Nesse cenário, os nomes @apple e %apple não têm relação, porque cada um pertence a um espaço de nome diferente. Além disso, um leitor de programa sempre conhece o tipo de uma variável quando lê seu nome.

No C e no C++, às vezes, deve-se distinguir entre declarações e definições. As declarações especificam tipos e outros atributos, mas não causam alocação de armazenamento. As definições especificam atributos e causam alocação de armazenamento. Para um nome específico, um programa C pode ter qualquer número de declarações compatíveis, mas somente uma única definição. Uma finalidade das declarações de variáveis em C é fornecer o tipo de uma variável definida externamente a uma função e usada na

<sup>1</sup>N. de T. Hash: Técnica de pesquisa em tabela. No caso, uma matriz associativa de duas colunas na qual se entra com um valor chave na primeira coluna e retira-se o correspondente da segunda.

mesma. Ela diz ao compilador qual é o tipo de variável e que ela está definida em outro lugar. A idéia transporta-se para as funções em C e em C++, cujos protótipos declaram nomes e interfaces, mas não o código de funções. As definições de função, por outro lado, são completas.

#### 5.4.2.2 Vinculação Dinâmica de Tipos

Com a vinculação dinâmica de tipos, o tipo não é especificado por uma instrução de declaração. Em vez disso, a variável é vinculada a ele quando lhe é atribuído um valor em uma instrução de atribuição. Quando a instrução de atribuição é executada, a variável que está sendo atribuída é vinculada ao tipo do valor, da variável ou da expressão no lado direito da atribuição.

As linguagens que vinculam dinamicamente os tipos apresentam diferenças drásticas em relação às com os tipos estaticamente vinculados. A principal vantagem da vinculação dinâmica de variáveis a tipos é que ela proporciona muita flexibilidade de programação. Por exemplo, um programa para processar uma lista de dados em uma linguagem com vinculação dinâmica de tipos pode ser escrito como um programa genérico, significando que ele será capaz de lidar com dados de qualquer tipo. Sem fazer distinções entre os dados de tipos introduzidos, eles serão aceitos, porque as variáveis em que devem ser armazenados podem ser vinculadas ao tipo correto quando os dados forem atribuídos a elas, após a entrada. Em comparação, devido à vinculação estática de tipos, não se pode escrever um programa C++ ou Java para processar uma lista de dados sem se conhecer seu tipo.

Em APL, SNOBOL4 e em JavaScript, a vinculação de uma variável a um tipo é dinâmica. Por exemplo, um script em JavaScript pode conter a seguinte instrução:

```
LIST = [10.2 5.1 0.0]
```

Independentemente do tipo anterior da variável chamada LIST, essa atribuição faz com que ela se torne uma matriz unidimensional de elementos reais de tamanho 3. Se a instrução

```
LIST = 47
```

viesse depois da instrução acima, LIST iria tornar-se uma variável escalar inteira.

Há duas desvantagens na vinculação dinâmica de tipos. Primeiro, a capacidade de detecção de erros do compilador é diminuída em relação ao compilador de uma linguagem que possui vinculação de tipos estática, porque dois tipos quaisquer podem aparecer em lados opostos do operador de atribuição. Tipos incorretos nos lados direitos de atribuições não são detectados como erros; ao contrário, o tipo do lado esquerdo é simplesmente mudado para o incorreto. Por exemplo, suponhamos que em um programa particular, i e x sejam variáveis inteiros, e que y seja uma matriz de números reais. Suponhamos, ainda, que o programa precise da instrução de atribuição

```
i = x
```

mas que, devido a um erro de digitação, ele tenha a instrução de atribuição

```
i = y
```

Em uma linguagem com vinculação dinâmica de tipos, nenhum erro será detectado pelo compilador ou pelo sistema em tempo de execução. Assim, i é simplesmente mudado para o tipo matriz de números reais. Entretanto, uma vez que foi usado y em vez da variável x

correta, os resultados serão errôneos. Em uma linguagem com vinculação estática de tipos, o compilador detectaria o erro e o programa não chegaria à execução.

Note que essa desvantagem também está presente, até certo ponto, em algumas linguagens que usam vinculação estática de tipos, como, por exemplo, o FORTRAN, o C e o C++, que, em muitos casos, convertem automaticamente o tipo do LD de uma atribuição para o tipo do LE.

A outra desvantagem da vinculação dinâmica de tipos é que o custo para implementar a vinculação dinâmica de atributos é considerável, especialmente durante a execução. A verificação de tipos deve ser feita em tempo de execução. Além disso, toda variável deve ter um descritor associado a ela para manter o tipo atual. A memória usada para o valor de uma variável deve ter um tamanho variável porque diferentes valores de tipo exigem diferentes quantidades de armazenamento.

As linguagens com vinculação dinâmica de tipos para variáveis freqüentemente são implementadas usando interpretadores em vez de compiladores. Isso se deve, parcialmente, ao fato de ser difícil mudar dinamicamente os tipos de variáveis em código de máquina. Além disso, o tempo para fazer a vinculação dinâmica deles é oculto pelo tempo global de interpretação, de modo que ele parece menos custoso nesse ambiente. Por outro lado, linguagens com vinculação estática de tipo raramente são implementadas por meio de interpretação, porque os programas, nestas linguagens, podem ser facilmente traduzidos para versões de código de máquina muito eficientes.

#### 5.4.2.3 Inferência de Tipo

A ML é uma linguagem relativamente recente que suporta tanto programação funcional como imperativa (Milner et al., 1990). Ela emprega um mecanismo de inferência de tipo interessante, em que tipos da maioria das expressões podem ser determinados sem exigir que o programador especifique os tipos das variáveis. Por exemplo, a declaração de função

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

especifica uma função que leva um argumento real e produz um resultado real. Os tipos são inferidos do tipo da constante na expressão. De maneira semelhante, na função

```
fun vezes10(x) = 10 * x;
```

o argumento e o valor funcional são inferidos para serem do tipo inteiro.

O sistema ML rejeita a função

```
fun quadrado(x) = x * x;
```

porque o tipo para o operador \* não pode ser inferido. Nesses casos, o programador pode oferecer uma dica, como, por exemplo, a seguinte, em que o tipo retornado pela função é especificado para ser int.

```
fun quadrado(x) : int = x * x;
```

O fato do valor funcional ser tipificado como um inteiro é suficiente para inferir que o argumento também é um inteiro. As seguintes definições também são legais:

```
fun quadrado(x : int) = x * x;  
fun quadrado(x) = (x : int) * x;  
fun quadrado(x) = x * (x : int);
```

A inferência de tipo também é usada nas linguagens puramente funcionais Miranda e Haskell.

### 5.4.3 Vinculações de Armazenamento e Tempo de Vida

O caráter de uma linguagem de programação é, em grande parte, determinado pelo projeto das vinculações de armazenamento para suas variáveis. Portanto, é importante ter um claro entendimento dessas vinculações.

A célula de memória à qual uma variável é vinculada deve, de alguma maneira, ser tomada de um conjunto de memória disponível. Esse processo é chamado de **alocação**. **Desalocação** é o processo de devolver uma célula de memória desvinculada de uma variável ao conjunto de memória disponível.

O **tempo de vida** de uma variável é o tempo durante o qual esta é vinculada a uma localização de memória específica. Sendo assim, seu tempo de vida inicia-se quando ela é vinculada a uma célula específica e encerra-se quando ela é desvinculada dessa célula. Para investigar as vinculações de armazenamento de variáveis, é conveniente separar as variáveis escalares (não-estruturadas) em quatro categorias, de acordo com seus tempos de vida: estáticas, dinâmicas na pilha, dinâmicas no monte explícitas e dinâmicas no monte implícitas. Nas seções seguintes, discutiremos os significados das quatro, juntamente com seus propósitos, vantagens e desvantagens.

#### 5.4.3.1 Variáveis Estáticas

As **variáveis estáticas** são aquelas vinculadas a células da memória antes que a execução do programa inicie e que assim permanecem até que a execução do programa encerre. Variáveis estaticamente vinculadas a memória têm diversas aplicações valiosas para a programação. Evidentemente, variáveis globalmente acessíveis são usadas com freqüência no decorrer da execução de um programa, tornando, assim, necessário fazer com que elas sejam vinculadas ao mesmo armazenamento durante essa execução. Às vezes, é conveniente ter variáveis declaradas em subprogramas como **sensíveis à história**; ou seja, fazer com que elas retenham valores entre as execuções do subprograma. Isso é característico de uma variável estaticamente vinculada ao armazenamento.

Outra vantagem das variáveis estáticas é a eficiência. Todo o seu endereçamento pode ser direto; outros tipos de variáveis exigem endereçamento indireto, mais lento. Além disso, não se incorre em nenhuma sobretaxa em tempo de execução para alocação e desalocação de variáveis estáticas.

Uma desvantagem da vinculação de armazenamento estática é a reduzida flexibilidade; em especial, em uma linguagem com somente variáveis estaticamente vinculadas ao armazenamento, subprogramas recursivos não são suportados. Outra desvantagem é que o armazenamento não pode ser compartilhado entre variáveis. Por exemplo, suponhamos que um programa tenha dois subprogramas e que ambos exijam matrizes grandes não-relacionadas. Se elas forem estáticas, seu armazenamento não poderá ser compartilhado.

Nos FORTRAN I, II e IV, todas as variáveis eram estáticas. O C, o C++ e o Java permitem que os programadores incluam o especificador **static** em uma definição de variável local, tornando estáticas as variáveis que ele define. O Pascal não oferece variáveis estáticas.

#### 5.4.3.2 Variáveis Dinâmicas na Pilha

As **variáveis dinâmicas na pilha** são aquelas cujas vinculações de armazenamento criam-se a partir da elaboração de suas instruções de declaração, mas cujos tipos são esta-

ticamente vinculados. A **elaboração** dessas declarações refere-se ao processo de alocação e de vinculação de armazenamento indicado pela declaração, o qual se desenvolve quando a execução atinge o código em que ela se encontra. Portanto, a elaboração ocorre em tempo de execução. Por exemplo, um procedimento Ada consiste de uma seção de declaração e de uma seção de código. A primeira é elaborada imediatamente antes que a execução da seção do código se inicie, o que acontece quando o procedimento é chamado. O armazenamento para as variáveis na seção de declaração é alocado no momento de elaboração e desalocado quando o procedimento devolve o controle ao seu chamador. Em outras linguagens, como o Java, no qual as declarações não são confinadas a seções, a elaboração se dá quando a instrução de declaração é encontrada durante a execução.

Como seu nome indica, as variáveis dinâmicas na pilha são alocadas na pilha mantida pelo sistema em tempo de execução.

O projeto do ALGOL 60 e das linguagens que o sucederam permite subprogramas recursivos. Para serem úteis, pelo menos na maioria dos casos, os subprogramas recursivos exigem alguma forma de armazenamento local dinâmico, a fim de que cada cópia ativa do subprograma recursivo tenha sua própria versão das variáveis locais. Essas necessidades são convenientemente atendidas pelas variáveis dinâmicas na pilha. Mesmo na ausência de recursão, existem méritos em se ter armazenamento dinâmico na pilha para subprogramas porque todos os subprogramas compartilham do mesmo espaço de memória para suas variáveis locais. As desvantagens são a sobretaxa de alocação e de desalocação em tempo de execução e o fato das variáveis locais não poderem ser sensíveis à história.

O FORTRAN 90 permite que os implementadores usem variáveis dinâmicas na pilha para as locais, mas incluem uma instrução

`SAVE list`

que permite ao programador especificar que algumas ou todas as variáveis (aqueles que estão na lista) do subprograma em que `SAVE` é colocado sejam estáticas.

No Java e no C++ as variáveis locais são assumidas como dinâmicas na pilha. No Pascal e na Ada, todas as variáveis definidas em subprogramas que não sejam dinâmicas no monte serão dinâmicas na pilha.

Todos os atributos exceto os de armazenagem são estaticamente vinculados às variáveis escalares dinâmicas na pilha. Isso não acontece com alguns tipos estruturados, como veremos no Capítulo 6. A implementação do processo de alocação/desalocação para variáveis dinâmicas na pilha será discutida no Capítulo 10.

#### 5.4.3.3 Variáveis Dinâmicas no Monte Explícitas

As **variáveis dinâmicas no monte explícitas** são células de memória sem nome (abstratas) alocadas e desalocadas por instruções explícitas em tempo de execução, especificadas pelo programador. Essas variáveis alocadas no monte e desalocadas para o monte só podem ser referenciadas por meio de variáveis de ponteiro ou de referência. O monte é um conjunto de células de armazenamento altamente desorganizado devido à imprevisibilidade de seu uso. Uma variável dinâmica no monte explícita é criada por meio de um operador (por exemplo, na Ada e no C++) ou por meio de uma chamada a um subprograma de sistema fornecido para essa finalidade (por exemplo, no C).

No C++, o operador de alocação, chamado `new`, usa um nome de tipo como seu operando. Quando executada, uma variável dinâmica no monte explícita do tipo do operando é criada, e um ponteiro para ela é retornado. Uma vez que uma variável dinâmica no

monte é vinculada a um tipo durante a compilação, essa vinculação é estática. Porém, essas variáveis são vinculadas ao armazenamento no momento em que são criadas, o que acontece em tempo de execução.

Além de um subprograma ou de um operador para criar variáveis dinâmicas no monte explícitas, algumas linguagens incluem um meio para destruí-las.

Como um exemplo de variáveis dinâmicas no monte explícitas, considere o seguinte segmento de código C++:

```
int *intnode;  
...  
intnode = new int; /* aloca uma célula int */  
...  
delete intnode;    /* desaloca a célula para a qual  
                    intnode aponta */
```

Nesse exemplo, uma variável dinâmica no monte explícita do tipo `int` é criada pelo operador `new`. Ela poderá, então, ser referenciada pelo ponteiro, `intnode`. Depois, a variável é desalocada pelo operador `delete`.

Em Java, todos os dados, exceto os escalares primitivos, são objetos. Os objetos Java são dinâmicos no monte explícitos e acessados por meio de variáveis de referência. O Java não tem nenhuma maneira de destruir explicitamente uma variável dinâmica no monte; em vez disso, é usada a coleta de lixo implícita.

As variáveis dinâmicas no monte explícitas são usadas freqüentemente para estruturas dinâmicas, como, por exemplo, para listas encadeadas e para árvores que precisem crescer e/ou encolher durante a execução. Essas estruturas podem ser construídas convenientemente usando-se ponteiros ou referências e variáveis dinâmicas no monte explícitas.

As desvantagens das variáveis dinâmicas no monte explícitas são a dificuldade de usar variáveis de ponteiro ou de referência corretamente, além do custo das referências para as variáveis, para as alocações e para as desalocações. Essas considerações, tipos de dados de ponteiro e referência e de métodos de implementação para variáveis dinâmicas no monte explícitas serão discutidas extensamente no Capítulo 6.

#### 5.4.3.4 Variáveis Dinâmicas no Monte Implícitas

As **variáveis dinâmicas no monte implícitas** são vinculadas ao armazenamento no monte somente quando lhe são atribuídos valores. De fato, todos os seus atributos vinculam-se todas as vezes que lhe são atribuídos valores. Em certo sentido, são apenas nomes que se adaptam a qualquer uso que sejam solicitados a servir. A vantagem dessas variáveis é que elas têm o mais elevado grau de flexibilidade, permitindo que se escreva um código altamente genérico. A desvantagem é a sobretaxa para manter todos os atributos dinâmicos em tempo de execução, que poderia incluir tipos de subscrito de matriz e de faixas, entre outras coisas. Outra desvantagem é a perda de certa capacidade de detecção de erros pelo compilador, conforme discutimos na Seção 5.4.2.2; lá estão também exemplos de variáveis dinâmicas no monte implícitas em APL. As cadeias de caracteres e matrizes em Perl e JavaScript são dinâmicas no monte implícitas. Seus exemplos serão mostrados no Capítulo 6.

## 5.5 Verificação de Tipos

Para nossa discussão sobre a verificação de tipos, generalizamos o conceito de operandos e de operadores para incluirmos subprogramas e instruções de atribuição. Imaginamos os subprogramas como operadores cujos operandos sejam seus parâmetros. O símbolo de atribuição será imaginado como um operador binário, sendo sua variável alvo e sua expressão os operandos.

A **verificação de tipos** é a atividade de assegurar que os operandos de um operador sejam de tipos compatíveis. Um tipo **compatível** é aquele válido para o operador ou com permissão, nas regras da linguagem, para ser convertido pelo código gerado pelo compilador para um tipo válido. A essa conversão automática chama-se **coerção**. Um **erro de tipo** é a aplicação de um operador a um operando de tipo impróprio.

Se todas as vinculações de variáveis a tipos forem estáticas em uma linguagem, a verificação de tipos quase sempre poderá ser feita estaticamente. A vinculação dinâmica de tipos requer a verificação de tipo em tempo de execução, o que é chamado de verificação dinâmica de tipo.

Algumas linguagens, como o JavaScript e a APL, devido à sua vinculação dinâmica de tipos, permitem somente a verificação dinâmica destes. É muito melhor detectar erros durante a compilação do que na execução porque, quanto mais cedo for feita a correção, normalmente, menos custoso será. A penalidade para a verificação estática é a reduzida flexibilidade do programador. Um número menor de atalhos e de truques é possível. Essas técnicas, entretanto, agora detêm pouca estima.

A verificação de tipos é complicada quando uma linguagem permite que uma célula de memória armazene valores de diferentes tipos em diversos momentos durante a execução. Isso pode ser feito com registros variantes Ada e Pascal, EQUIVALENCE FORTRAN e uniões C e C++. Nesses casos, a verificação de tipos, caso seja feita, deve ser dinâmica e exige que o sistema mantenha, em tempo de execução, o tipo do valor atual dessas células de memória. Assim, ainda que todas as variáveis sejam estaticamente vinculadas a tipos em linguagens como o C e o Pascal, nem todos os erros de tipo podem ser detectados pela verificação estática de tipos.

## 5.6 Tipificação Forte

Uma das novas idéias no projeto de linguagens que se tornou proeminente na chamada revolução da programação estruturada da década de 70 é a **tipificação forte**, muito reconhecida como um conceito altamente valioso. Infelizmente, muitas vezes, ela é definida de maneira livre e é usada na literatura de informática sem ser definida de modo algum.

A seguir, apresentamos uma definição simples, mas incompleta, de uma linguagem com tipificação forte: cada nome de um programa escrito na linguagem tem um único tipo associado a ela. Esse tipo é conhecido no momento da compilação. A essência dessa definição é que todos os tipos são vinculados estaticamente. A fragilidade reside no fato dela ignorar a possibilidade que, não obstante um tipo da variável poder ser conhecido, a loca-

lização do armazenamento ao qual ele está vinculado pode armazenar valores de diferentes tipos em diferentes tempos. Para levarmos essa possibilidade em consideração, definimos uma linguagem de programação como sendo **fortemente tipificada** se erros de tipo sempre forem detectados. Isso exige que os tipos de todos os operandos possam ser determinados durante a compilação ou em tempo de execução. A importância da tipificação forte reside em sua capacidade de detectar todos os usos equivocados de variáveis que resultam em erros de tipo. Uma linguagem fortemente tipificada também permite a detecção, em tempo de execução, da utilização dos valores de tipo incorretos em variáveis que podem armazenar valores de mais de um tipo.

O FORTRAN não é fortemente tipificado porque a relação entre parâmetros reais e formais não é verificada quanto ao tipo. Além disso, o uso de **EQUIVALENCE** entre variáveis de tipos diferentes permite que uma variável de um tipo se refira a um valor de um tipo diferente sem que o sistema seja capaz de verificar o tipo do valor quando uma das variáveis em **EQUIVALENCE** é referenciada ou atribuída. De fato, a verificação de tipos de variáveis em **EQUIVALENCE** eliminaria a maior parte de sua utilidade.

O Pascal é quase fortemente tipificado, mas falha em seu projeto de registros variantes porque permite a omissão da marca que armazena o tipo atual de uma variável, que oferece os meios de verificação dos tipos de valor corretos. Discutiremos os registros variantes e seus problemas potenciais no Capítulo 6.

A Ada é quase fortemente tipificada. As referências a variáveis em registros variantes são verificadas dinamicamente quanto aos valores de tipo corretos. Isso é muito melhor do que o Pascal, no qual a verificação nem mesmo é possível, muito menos exigida. Porém, a Ada permite que os programadores violem suas regras de verificação de tipos ao exigir, especificamente, que a verificação seja suspensa para uma conversão particular. Essa suspensão temporária da verificação de tipos pode ser feita somente quando a função de biblioteca **UNCHECKED\_CONVERSION** é usada. Tal função, da qual pode haver uma versão para cada tipo de dados, pega uma variável de seu tipo como parâmetro e retorna a cadeia de bits do valor atual dessa variável. Nenhuma conversão real desenvolve-se; é, simplesmente, um meio de extrair o valor de uma variável de um tipo e usá-lo como se fosse de tipo diferente. Isso pode ser útil para operações de alocação e de desalocação de armazenamento definidas pelo usuário, nas quais endereços são manipulados como números inteiros, mas devem ser usados como ponteiros. Uma vez que nenhuma verificação é feita em **UNCHECKED\_CONVERSION**, cabe ao programador assegurar que o uso de um valor obtido seja significativo.

O Modula-3 tem um procedimento predefinido chamado **LOophole** que serve ao mesmo propósito que o **UNCHECKED\_CONVERSION** da Ada.

O C e o C++ não são linguagens fortemente tipificadas porque ambas permitem funções para as quais os parâmetros não são verificados quanto ao tipo. Além disso, os tipos união dessas linguagens não são verificados quanto à compatibilidade.

A ML é fortemente tipificada, mas em um sentido ligeiramente diferente daquele das linguagens imperativas. A ML tem variáveis cujos tipos são todos estaticamente conhecidos a partir das declarações ou a partir de suas regras de inferência de tipos, conforme discutimos na Seção 5.4.2.3.

O Java, não obstante ser livremente baseado no C++, é fortemente tipificado no mesmo sentido que a Ada. Tipos podem ser explicitamente convertidos, o que poderia resultar em um erro de tipo. Porém, não há nenhuma maneira implícita pela qual os erros de tipo não possam ser detectados.

As regras de coerção de uma linguagem têm um efeito importante sobre o valor da verificação de tipos. Por exemplo, as expressões são fortemente tipificadas em Java. Porém,

um operador aritmético com um operando real e um operando inteiro é legal. O valor deste último é convertido em real, e uma operação com números reais desenvolve-se. Normalmente, isso é o que o programador pretende. Porém, a coerção também resulta na perda de parte da razão para a tipificação forte — detecção de erros. Assim, o valor da tipificação forte é enfraquecido pela coerção. As linguagens com muita coerção, como o FORTRAN, o C e o C++, são significativamente menos confiáveis do que aquelas com pouca, como a Ada. O Java tem a metade dos tipos de coerção em atribuições que o C++, logo sua detecção de erros é melhor que a do C++, mas não tão efetiva quanto a da Ada. A questão da coerção será examinada detalhadamente no Capítulo 7.

## 5.7 Compatibilidade de Tipos

A idéia da compatibilidade de tipos foi definida quando se introduziu a questão da verificação de tipos. Nesta seção, investigaremos os vários tipos de regras de compatibilidade de tipos das linguagens. O projeto das regras de compatibilidade de tipos de uma linguagem é importante porque influencia o projeto dos tipos de dados e as operações fornecidas para os seus valores. Talvez o resultado mais importante de duas variáveis serem de tipos compatíveis é que qualquer uma delas pode ter seu valor atribuído à outra.

Há dois métodos diferentes de compatibilidade de tipos: a compatibilidade de nome e a compatibilidade de estrutura. A **compatibilidade de nome** significa que duas variáveis têm tipos compatíveis somente se estiverem na mesma declaração ou em declarações que usam o mesmo nome de tipo. A **compatibilidade de estrutura** significa que duas variáveis têm tipos compatíveis se os seus tipos tiverem estruturas idênticas. Há algumas variações desses dois métodos e a maioria das linguagens usa combinações de diferentes técnicas.

A compatibilidade de tipo de nome é fácil de implementar, mas é altamente restritiva. Em uma interpretação estrita, uma variável subfaixa dos números inteiros não seria compatível com uma variável do tipo inteiro. Por exemplo, supondo que Pascal usasse compatibilidade estrita de nomes de tipo, considere o seguinte:

```
type indextype = 1..100; {um tipo subfaixa }
var
  cont : integer;
  indice : indextype;
```

As variáveis `cont` e `indice` não são compatíveis; `cont` não seria atribuída a `indice` e vice-versa.

Outro problema com a compatibilidade de tipo de nome surge quando um tipo estruturado é passado entre subprogramas por meio de parâmetros. Esse tipo deve ser definido somente uma vez, globalmente. Um subprograma não pode declarar o tipo desses parâmetros formais em termos locais. Isso acontece com a versão original do Pascal.

A compatibilidade de tipo de estrutura é mais flexível do que a de nome, mas é mais difícil de implementar. Nesta, somente os dois nomes de tipo devem ser comparados para determinar a compatibilidade. Naquela, entretanto, as estruturas inteiras dos dois tipos devem ser comparadas. Essa comparação nem sempre é simples (considere uma estrutura de dados que se refira a seu próprio tipo, como, por exemplo, uma lista encadeada). Outra questão também pode surgir. Por exemplo, dois tipos de registro ou de estrutura são compatíveis, se tiverem a mesma estrutura, mas nomes de campo diferentes? Dois tipos de

matrizes unidimensionais em um programa Pascal ou Ada são compatíveis, se tiverem o mesmo tipo de elemento, mas faixas de subscrito iguais a `0..10` e `1..11`? Dois tipos de enumeração são compatíveis, se tiverem o mesmo número de componentes, mas grafarem os literais de maneira diferente?

Outra dificuldade com a compatibilidade de tipos estruturada é que ela desautoriza diferenciar entre tipos que têm a mesma estrutura. Por exemplo, considere as seguintes declarações assemelhadas ao Pascal:

```
type celsius = real;
        fahrenheit = real;
```

As variáveis desses dois tipos são consideradas compatíveis sob a compatibilidade de tipo de estrutura, permitindo que elas sejam misturadas em expressões, o que, certamente, é indesejável nesse caso. Em geral, tipos com nomes diferentes têm a probabilidade de ser abstrações de diferentes categorias de valores de problema e não devem ser considerados equivalentes.

A definição original do Pascal (Wirth, 1971) não especifica claramente quando a compatibilidade de nome ou de estrutura deve ser usada. Isso é altamente prejudicial para a portabilidade, porque um programa correto em uma implementação poderia ser ilegal em outra. A ISO Standard Pascal (ISO, 1982) define as regras de sua compatibilidade de tipos com clareza: nem completamente pelo nome, nem pela estrutura. Esta é usada na maioria dos casos, enquanto aquele é usado para parâmetros formais e em algumas outras situações. Por exemplo, considere as seguintes declarações:

```
type
    tipol = array [1..10] of integer;
    tipo2 = array [1..10] of integer;
    tipo3 = tipo2;
```

Nesse exemplo, `tipol` e `tipo2` não são compatíveis, demonstrando que a compatibilidade de tipo de estrutura não é usada. Além disso, `tipo2` é compatível com `tipo3`, ilustrando que a mesma equivalência de nomes não é, tampouco, estritamente usada. Essa forma de compatibilidade, às vezes, é chamada de **equivalência de declaração**, porque quando um tipo é definido com o nome de outro tipo, os dois são compatíveis, ainda que não sejam compatíveis quanto ao nome.

A Ada usa a compatibilidade de nome, mas fornece duas construções de tipo: subtipos e tipos derivados, que evitam os problemas com esse método. Um **tipo derivado** é um novo que se baseia em algum anteriormente definido, com o qual ele é incompatível, ainda que possa ter uma estrutura idêntica. Eles herdam todas as propriedades de seus tipos-pai. Considere o seguinte exemplo:

```
type celsius is new FLOAT;
type fahrenheit is new FLOAT;
```

Variáveis desses dois tipos derivados não são compatíveis, ainda que suas estruturas sejam idênticas. Além disso, variáveis de nenhum desses tipos são compatíveis com qualquer outra de números reais. Os literais são uma exceção à regra. Um literal, como `3.0`, por exemplo, tem o tipo universal `real` e é compatível com qualquer `real`. Os tipos derivados também podem incluir restrições a faixas no tipo-pai, ao mesmo tempo que herdam todas as operações deste.

Um **subtipo** Ada é uma possível reversão de um tipo existente com limitações em seus limites. Um subtipo é compatível com seu tipo-pai. Por exemplo, considere a seguinte declaração:

```
subtype SMALL_TYPE is INTEGER range 0..99;
```

Variáveis do tipo `SMALL_TYPE` são compatíveis com variáveis `INTEGER`.

As regras de compatibilidade de tipos para a Ada são mais importantes do que aquelas para as linguagens com muitas coerções entre tipos. Por exemplo, os dois operandos de um operador de adição em C podem ter, virtualmente, qualquer combinação de tipos numéricos na linguagem. Um dos operandos simplesmente será convertido para o tipo do outro. Mas, na Ada, não existem coerções dos operandos de um operador aritmético.

O C usa equivalência estrutural para todos os tipos, exceto para as estruturas (registros do C) e uniões, para os quais o C usa equivalência de declaração. Essa regra modifica-se, entretanto, quando duas estruturas ou uniões são definidas em dois arquivos diferentes. Nesse caso, é usada a equivalência estrutural de tipos.

O C++ usa equivalência de nomes. Note que `typedef` no C e no C++ não introduz um tipo novo, simplesmente define um novo nome para um tipo existente.

Variáveis podem ser declaradas em muitas linguagens sem usar nomes de tipos, criando variações anônimas destes. Considere os seguintes exemplos Ada:

```
A : array (1..10) of INTEGER;
```

Nesse caso, A tem um tipo anônimo, mas único. Se também tivéssemos

```
B : array (1..10) of INTEGER;
```

A e B seriam de tipos anônimos, mas distintos e incompatíveis, ainda que fossem estruturalmente idênticos. A declaração múltipla

```
C, D : array (1..10) of INTEGER;
```

cria dois tipos anônimos, um para C e um para D, ambos incompatíveis. Essa declaração é, de fato, tratada como se fosse as duas declarações seguintes:

```
C : array (1..10) of INTEGER;
```

```
D : array (1..10) of INTEGER;
```

O resultado disso é que C e D não são compatíveis. Se, em vez disso, tivéssemos escrito

```
type LIST_10 is array (1..10) of INTEGER;
```

```
C, D : LIST_10;
```

C e D seriam compatíveis.

Em linguagens que não permitem aos usuários definirem e nomearem tipos, como o FORTRAN e o COBOL, a equivalência de nomes evidentemente não pode ser usada.

As linguagens orientadas a objeto, como o Java e o C++, trazem consigo a questão da compatibilidade de objetos e sua relação com a hierarquia de herança. Isso será discutido no Capítulo 12.

A compatibilidade de tipos em expressões será discutida no Capítulo 7; a compatibilidade de tipos para parâmetros de subprograma será discutida no Capítulo 9.

## 5.8 Escopo

Um dos fatores mais importantes a ter efeito sobre o entendimento das variáveis é o escopo. O **escopo** de uma variável de programa é a faixa de instruções na qual a variável é visível. Uma variável é **visível** em uma instrução se puder ser referenciada nessa instrução.

As regras de escopo de uma linguagem determinam como uma ocorrência particular de um nome está associada à variável. Em particular, as regras de escopo determinam como as referências a variáveis declaradas fora do subprograma ou do bloco que estão atualmente em execução são associadas a suas declarações e, assim, a seus atributos (os blocos serão discutidos na Seção 5.8.2). Portanto, o completo conhecimento dessas regras para uma linguagem é fundamental para nossa capacidade de escrever ou de ler programas nessa linguagem.

Conforme definimos na Seção 5.4.3.2, uma variável é local em uma unidade ou em um bloco de programa se for declarada aí (para este capítulo, consideramos as unidades como unidades do programa principal ou como subprogramas; unidades como as classes C++ e Java serão discutidas no Capítulo 11). As **variáveis não-locais** de uma unidade ou de um bloco de programa são as visíveis dentro daquela ou deste, mas não são declaradas aí.

### 5.8.1 Escopo Estático

O ALGOL 60 introduziu o método de vincular nomes a variáveis não-locais, chamado **escopo estático**, o qual foi copiado pela maioria das linguagens imperativas subsequentes, e por muitas linguagens não-imperativas também. O escopo estático recebeu esse nome porque o escopo de uma variável pode ser determinado estaticamente, ou seja, antes da execução.

A maior parte dos escopos estáticos nas linguagens imperativas está associada a definições de unidade de programa, e, por enquanto, presumiremos que todos os escopos assim estão. Neste capítulo, também presumimos que a determinação do escopo é o único método para acessar variáveis não-locais nas linguagens em discussão. Isso não é verdadeiro em relação a todas as linguagens. Nem mesmo é verdadeiro para todas aquelas que usam escopo estático, mas a suposição simplifica a discussão aqui. Métodos adicionais de acessar não-locais serão discutidos no Capítulo 9.

Em muitas linguagens, os subprogramas criam seus próprios escopos. No Pascal, na Ada e em JavaScript, os subprogramas podem ser aninhados dentro de outros subprogramas, os quais criam uma hierarquia de escopos em um programa.

Quando uma referência a uma variável é encontrada pelo compilador de uma linguagem de escopo estático, os atributos da variável são determinados localizando-se a instrução na qual ela é declarada. Em linguagens de escopo estático com subprogramas aninhados, esse processo pode ser imaginado da seguinte maneira: suponhamos que uma referência seja feita a uma variável *x* no subprograma *sub1*. A declaração correta é encontrada procurando-se, primeiro, as declarações do subprograma *sub1*. Se nenhuma declaração for encontrada para a variável que lá se encontra, a procura prosseguirá nas declarações do subprograma que declarou o subprograma *sub1*, o qual é chamado de **pai-estático** (*static parent*). Se uma declaração de *x* não for encontrada aí, a procura prosseguirá na próxima unidade envolvente maior (a unidade que declarou o pai de *sub1*) e assim por diante, até que uma declaração para *x* seja encontrada ou as declarações da unidade maior tenham sido pesquisadas sem sucesso. Nesse caso, um erro de variável não-declarada terá sido detectado. O pai-estático do subprograma *sub1*, seu pai-estático e assim por diante até o programa principal, inclusive, são chamados **ancestrais estáticos** de *sub1*. Note que as técnicas de implementação do escopo estático, que serão discutidas no Capítulo 10, são muito mais eficientes do que o processo que acabamos de descrever.

Considere o seguinte procedimento Pascal:

```
procedure big;
  var x: integer;
  procedure sub1;
```

```

begin { sub1 }
...
end; { sub1 }
procedure sub2;
var x: integer;
begin { sub2 }
...
end; { sub2 }
begin { big }
...
end; { big }

```

No escopo estático, a referência à variável *x* em *sub1* é ao *x* declarado no procedimento *big*. Isso é verdadeiro porque a procura por *x* inicia-se no procedimento em que a referência ocorre, *sub1*, mas nenhuma declaração para *x* é encontrada aí. Assim, a procura prossegue no pai-estático de *sub1*, *big*, no qual a declaração de *x* é encontrada.

A presença de nomes predefinidos, discutidos na Seção 5.2.3, complica bastante esse procedimento. Em alguns casos, um nome predefinido é como uma palavra-chave e pode ser redefinido pelo usuário. Em alguns casos, ele é usado somente se o programa do usuário não contiver uma redefinição. Em outros, ele pode ser reservado, o que significa que a procura pelo significado de determinado nome inicia-se com a lista de nomes predefinidos, mesmo antes que as declarações de escopo local sejam verificadas.

Em linguagens de escopo estático, algumas declarações de variáveis podem ser ocultas de alguns subprogramas. Por exemplo, considere o seguinte programa Pascal esquemático:

```

program main;
var x : integer;
procedure sub1;
var x : integer;
begin { sub1 }
...
end; { sub1 }
begin { main }
...
end. { main }

```

A referência a *x* em *sub1* é ao *x* declarado de *sub1*. Nesse caso, o *x* do programa *main* está oculto no código de *sub1*. Em geral, uma declaração para uma variável oculta efetivamente qualquer declaração de uma variável com o mesmo nome em um escopo envolvente maior.

Na Ada, as variáveis ocultas de escopos ancestrais podem ser acessadas com referências seletivas, que incluem o nome do escopo do ancestral. Por exemplo, no programa precedente, o *x* declarado no programa *main* (principal) pode ser acessado em *sub1* pela referência *main.x*.

O C e o C++ têm variáveis globais, ainda que não permitam que subprogramas sejam aninhados dentro de outras definições de subprogramas. Essas variáveis são declaradas fora de qualquer definição de subprograma. Variáveis locais podem ocultar as globais, como no Pascal. No C++, essas globais ocultas podem ser acessadas, usando-se o operador de escopo (`::`). Por exemplo, se *x* for uma global que é oculta em um subprograma por uma local chamada *x*, a global poderia ser referenciada como `::x`.

### 5.8.2 Blocos

Muitas linguagens permitem que novos escopos estáticos sejam definidos no meio de código executável. Esse conceito poderoso, introduzido no ALGOL 60, permite que uma seção de código tenha suas próprias variáveis locais cujo escopo é minimizado. Essas variáveis são tipicamente dinâmicas na pilha, de modo que têm seu armazenamento alocado quando a seção é iniciada, e desalocado quando ela é finalizada. Esse tipo de seção de código é chamado de **bloco**.

Na Ada, os blocos são declarados com cláusulas **declare**, como em

```
...
declare TEMP : integer;
begin
    TEMP := FIRST;
    FIRST := SECOND;
    SECOND := TEMP;
end;
...
```

Os blocos constituem a origem da frase **linguagem estruturada por blocos**. Não obstante o Pascal e o Modula-2 serem chamados linguagens estruturadas por blocos, eles não incluem os não-baseados em procedimentos.

O C, o C++ e o Java permitem que qualquer instrução composta (uma seqüência de instruções contornadas por chaves emparelhadas) tenha declarações e, dessa forma, defina um novo escopo. Essas instruções compostas são blocos. Por exemplo, se *list* fosse uma matriz de números inteiros, poderíamos escrever

```
if (list[i] < list[j]) {
    int temp;
    temp = list[i];
    list[i] = list[j];
    list[j] = temp;
}
```

Os escopos criados por blocos são tratados exatamente como aqueles criados por subprogramas. Referências a variáveis de um bloco, que não estejam aí declaradas, são conectadas a declarações, procurando escopos envolventes em ordem de tamanho.

O C++ e o Java permitem que definições de variáveis apareçam em qualquer lugar em funções. Quando uma definição aparece em uma posição que não no início de uma função, o escopo dessa variável é a partir de sua instrução de definição até o final daquela.

As instruções **for** de C++ e de Java permitem definições de variáveis em suas expressões de inicialização. Nas primeiras versões do C++, o escopo desse tipo de variável partia de sua definição até o final do menor bloco envolvente. Na versão padrão, entretanto, o escopo restringe-se à construção **for**, como acontece com o Java.

As definições de classe e de método nas linguagens orientadas a objeto também criam escopos estáticos aninhados. Isso será discutido no Capítulo 12.

### 5.8.3 Avaliação do Escopo Estático

O escopo estático constitui um método de acesso não-local que funciona bem em muitas situações. Porém, ele tem seus problemas. Considere o programa cuja estrutura esquemáti-



**FIGURA 5.1** A estrutura de um programa.

ca é mostrada na Figura 5.1. Para esse exemplo, supomos que todos os escopos são criados pelas definições do programa principal e pelos procedimentos.

Esse programa contém um escopo global para `main`, com dois procedimentos que definem os escopos dentro de `main`, `A` e `B`. Dentro de `A`, há escopos para os procedimentos `C` e `D`. Dentro de `B`, está o escopo do procedimento `E`. Supomos que o acesso necessário a dados e a procedimentos tenha determinado a estrutura desse programa. O acesso a procedimentos exigido é o seguinte: `main` pode chamar `A` e `B`, `A` pode chamar `C` e `D`, e `B` pode chamar `A` e `E`.

É conveniente visualizar a estrutura do programa como uma árvore na qual cada vértice representa um procedimento e, assim, um escopo. Uma representação em árvore do programa da Figura 5.1 é mostrada na Figura 5.2. A estrutura deste programa talvez pareça ser uma organização de programa muito natural que reflete claramente as necessidades de projeto. Entretanto, um grafo das chamadas potenciais a procedimentos desse sistema, exposto na Figura 5.3 (ver p. 198), mostra que é possível uma grande quantidade de oportunidades de chamada além do exigido.

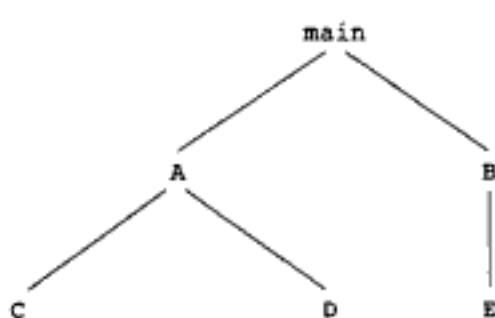
A Figura 5.4 (ver p. 198) mostra as chamadas desejadas do programa de exemplo. A diferença entre as Figuras 5.3 e 5.4 ilustra o número de chamadas possíveis que não são necessárias nessa aplicação específica.

Um programador poderia erroneamente chamar um subprograma que não deveria ter sido chamado, o que não seria detectado como um erro pelo compilador. Isso retarda a detecção do erro até a execução, o que poderia tornar sua correção mais custosa. Portanto, o acesso a procedimentos deve restringir-se aos necessários.

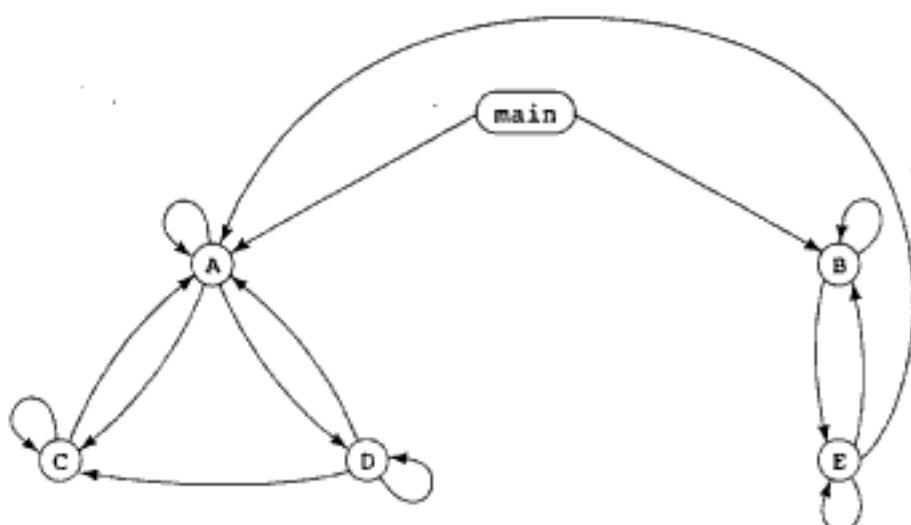
Demasiado acesso a dados é um problema estreitamente relacionado. Por exemplo, todas as variáveis declaradas no programa principal são visíveis a todos os procedimentos, quer isto seja desejado ou não, e não há nenhuma maneira de evitá-lo.

Para ilustrar outro tipo de problema com a determinação de escopo estático, considere o cenário a seguir. Suponhamos que, depois que o programa tiver sido desenvolvido e testado, uma modificação de sua especificação seja necessária. Em particular, suponhamos que o procedimento `E` deva ter acesso a algumas variáveis do escopo de `D`. Uma maneira de oferecer tal recurso é mover `E` para dentro do escopo de `D`. Mas, então, `E` não mais poderá acessar o escopo de `B`, o que presumivelmente ele precisará fazer (de outro modo, por que ele estaria aí?). Outra solução é mover as variáveis definidas em `D` que sejam necessárias a `E` para `main`. Isso permitiria o acesso por todos os procedimentos, o

que seria mais do que o necessário e, assim, cria a possibilidade de acessos incorretos. Por exemplo, um identificador grafado erroneamente em um procedimento pode ser tomado como uma referência a um identificador em algum escopo envolvente, em vez de ser detectado como um erro. Além disso, suponhamos que a variável movida para `main` seja chamada `x`, e que seja necessária para `D` e `E`. Mas, suponhamos que haja uma variável chamada `x` declarada em `A`. Isso ocultaria o `x` correto de seu proprietário original, `D`.



**FIGURA 5.2** A estrutura em árvore do programa da Figura 5.1.

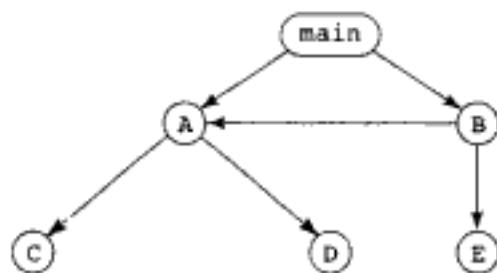


**FIGURA 5.3** O grafo de chamada potencial do programa da Figura 5.1.

Um problema final com a ação de mover a declaração de `x` para `main` é o fato de ser prejudicial para a legibilidade ter a declaração das variáveis tão distante de seus usos.

Os problemas associados à visibilidade das variáveis com escopo estático também estão presentes no acesso a subprogramas. Na árvore da Figura 5.2, suponhamos que, devido a alguma mudança de especificação, o procedimento `B` precisasse chamar o procedimento `D`. Isso somente poderia ser realizado movendo-se `D` para que se aninhe diretamente em `main`, supondo-se que ele também seja necessário para `A` ou `C`. Ele também perderia acesso às variáveis definidas em `A`. Essa solução, quando usada repetidamente, resulta em programas que se iniciam com listas longas de procedimentos de pouca utilidade.

Desse modo, contornar as restrições do escopo estático pode levar a projetos de programa que sustentam pouca semelhança com o original, mesmo em áreas do programa em que não ocorreram mudanças. Os projetistas são encorajados a usar bem mais globais do que o necessário. Todos os procedimentos podem acabar sendo aninhados no mesmo nível, no programa principal (`main`), usando globais em vez de níveis mais profundos de aninhamento. Além disso, o projeto final pode ser deselegante e complicado, além de não refletir o projeto conceitual subjacente. Esses e outros defeitos do escopo estático são discutidos detalhadamente em Clarke, Wileden e Wolf (1980). Uma solução para os problemas do escopo estático é uma construção de encapsulamento, discutida no Capítulo 9 e incluída em muitas linguagens mais recentes.



**FIGURA 5.4** O grafo das chamadas desejáveis no programa da Figura 5.1.

### 5.8.4 Escopo Dinâmico

O escopo das variáveis em APL, em SNOBOL4 e nas primeiras versões do LISP é dinâmico. Perl também permite que as variáveis sejam declaradas com escopo dinâmico. O **escopo dinâmico** baseia-se na seqüência de chamada de subprogramas, não em suas relações espaciais um com o outro. Dessa forma, o escopo pode ser determinado somente em tempo de execução.

Considere novamente o procedimento *big* da Seção 5.8.1, mostrado novamente aqui:

```
procedure big;
    var x : integer;
    procedure sub1;
        begin { sub1 }
        ... x ...
        end; { sub1 }
    procedure sub2;
        var x: integer;
        begin { sub2 }
        ...
        end; { sub2 }
    begin { big }
    ...
end; { big }
```

Suponhamos que as regras de escopo dinâmico apliquem-se a referências não-locais. O significado do identificador *x* referenciado em *sub1* é dinâmico — ele não pode ser determinado durante a compilação. Ele pode referenciar a variável a partir de qualquer uma das declarações de *x*, dependendo da seqüência de chamada.

Uma maneira pela qual o significado correto de *x* pode ser determinado em tempo de execução é iniciar a procura com as declarações locais. Essa também é a maneira pela qual se iniciou a determinação do escopo estático, mas é aqui que terminam as semelhanças entre as duas técnicas. Quando a procura de declarações locais falha, as declarações do pai-dinâmico ou do procedimento de chamada são pesquisadas. Se uma declaração para *x* não for encontrada aí, a procura prosseguirá no pai-dinâmico desse procedimento e assim por diante, até que uma declaração de *x* seja encontrada. Se nenhuma for encontrada em qualquer ancestral dinâmico, haverá um erro em tempo de execução.

Considere as duas diferentes seqüências de chamada para *sub1* no exemplo acima. Primeiro, *big* chama *sub2*, a qual chama *sub1*. Nesse caso, a procura prossegue a partir do procedimento local, *sub1*, para seu chamador, *sub2*, no qual uma declaração para *x* é encontrada. Assim, a referência a *x* em *sub1* nesse caso é ao *x* declarado em *sub2*. Em seguida, *sub1* é chamado diretamente de *big*. Nesse caso, o pai-dinâmico de *sub1* é *big* e a referência é para o *x* declarado em *big*.

### 5.8.5 Avaliação do Escopo Dinâmico

O efeito do escopo dinâmico sobre a programação é profundo. Os atributos corretos de variáveis não-locais visíveis a uma instrução de programa não podem ser determinados estaticamente. Além disso, essas variáveis nem sempre são as mesmas. Uma instrução em um subprograma que contenha uma referência a uma variável não-local pode se referir a

diferentes destas durante diversas execuções do subprograma. Vários tipos de problemas de programação seguem-se diretamente do escopo dinâmico.

Primeiro, durante o intervalo de tempo em que um subprograma inicia sua execução e termina quando esta se encerra, as variáveis locais do subprograma são todas visíveis a qualquer outro subprograma em execução, independentemente de sua proximidade textual. Não existe nenhuma maneira de proteger as variáveis locais dessa acessibilidade. Os subprogramas são sempre executados no ambiente imediato do chamador; portanto, o escopo dinâmico resulta em programas menos confiáveis do que os de escopo estático.

Um segundo problema com o escopo dinâmico é a incapacidade de, estaticamente, verificar o tipo das referências a não-locais. Isso resulta da incapacidade de determinar estaticamente a declaração para uma variável referenciada como não-local.

O escopo dinâmico dificulta muito mais a leitura dos programas porque a seqüência de chamada dos subprogramas deve ser conhecida para determinar o significado das referências a variáveis não-locais. Isso pode ser, virtualmente, impossível para um leitor humano.

Finalmente, os acessos a variáveis não-locais em linguagens com escopo dinâmico tomam mais tempo do que os acessos a não-locais quando o escopo estático é usado. A razão para isso será explicada no Capítulo 10.

Por outro lado, o escopo dinâmico tem seus méritos. Em alguns casos, os parâmetros passados de um suprograma a outro são simplesmente variáveis definidas no chamador. Nenhum deles precisa ser passado em uma linguagem de escopo dinâmico porque são implicitamente visíveis no subprograma chamado.

Não é difícil entender porque o escopo dinâmico não é tão usado quanto o estático. Os programas em linguagens de escopo estático são de leitura mais fácil, mais confiáveis e sua execução é mais rápida do que programas equivalentes em linguagens de escopo dinâmico. Exatamente por esses motivos que este último foi substituído por aquele na maioria dos dialetos atuais do LISP. Métodos de implementação para ambos serão discutidos no Capítulo 10.

## 5.9 Escopo e Tempo de Vida

---

Às vezes, o escopo e o tempo de vida de uma variável parecem estar relacionados. Por exemplo, considere uma variável declarada em um procedimento Pascal que não contenha nenhuma chamada a procedimentos. O escopo dela parte de sua declaração e vai até a palavra reservada **end** do procedimento. O tempo de vida da variável é o período que se inicia quando se introduz o procedimento e se encerra quando a execução do procedimento atinge o **end** (o Pascal não tem nenhuma instrução de retorno). Não obstante o escopo e o tempo de vida da variável evidentemente não serem os mesmos, porque o escopo estático é um conceito textual ou espacial enquanto o tempo de vida é um conceito temporal, nesse caso, pelo menos, eles parecem estar relacionados.

Essa aparente relação entre escopo e tempo de vida não se sustenta em outras situações. No C e no C++, por exemplo, uma variável declarada em uma função usando o especificador **static** está estaticamente vinculada ao escopo dela e, também, estaticamente vinculada ao armazenamento. Assim, o escopo é estático e local para a função, mas seu tempo de vida estende-se ao longo de toda a execução do programa do qual ela faz parte.

Escopo e tempo de vida também não têm relação quando chamadas a subprogramas estão envolvidas. Considere as seguintes funções C++:

```

void imprimecabecalho() {
    ...
} /* fim de imprimecabecalho */
void computa() {
    int soma;
    ...
imprimecabecalho();
} /*fim de computa */

```

O escopo da variável `soma` é completamente contido pela função `computa`. Ele não se estende para o corpo da função `imprimecabecalho`, ainda que `imprimecabecalho` execute no meio da execução de `computa`. Porém, o tempo de vida de `soma` estende-se ao longo do tempo durante o qual `imprimecabecalho` é executado. Seja qual for a localização de armazenagem à qual `soma` esteja vinculado antes da chamada a `imprimecabecalho`, essa vinculação prosseguirá durante e depois da execução de `imprimecabecalho`.

## 5.10 Ambientes de Referenciamento

**O ambiente de referenciamento** de uma instrução é o conjunto de todos os nomes visíveis na instrução. O ambiente de referenciamento de uma instrução em uma linguagem de escopo estático são as variáveis declaradas em seu escopo local mais o conjunto de todas as variáveis de seus escopos ancestrais visíveis. Em uma linguagem desse tipo, o ambiente de referenciamento de uma instrução é necessário enquanto esta está sendo compilada, de modo que código e estruturas de dados possam ser criados para permitir referências a variáveis a partir de outros escopos em tempo de execução. Técnicas para implementar referências a variáveis não-locais tanto em linguagens de escopo estático como em linguagens de escopo dinâmico serão discutidas no Capítulo 10.

No Pascal, cujos escopos são criados somente por definições de procedimentos, o ambiente de referência de uma instrução inclui as variáveis locais, mais todas as variáveis declaradas nos procedimentos nos quais a instrução está aninhada, somadas as variáveis declaradas no programa principal (excluindo-se aquelas em escopos não-locais ocultas por declarações em procedimentos mais próximos). Cada definição de procedimento cria um novo escopo e, assim, um novo ambiente. Considere o seguinte programa Pascal esquemático:

```

program example;
var a, b : integer;
...
procedure sub1;
    var x, y : integer;
    begin { sub1 }
        ...
    end; { sub1 }
procedure sub2;
    var x : integer;
    ...
procedure sub3;
    var x : integer;

```

```

begin { sub3 }
... ←—————2
end; { sub3 }
begin { sub2 }
... ←—————3
end; { sub2 }
begin { example }
... ←—————4
end. { example }

```

Os ambientes de referenciamento dos pontos de programa estão indicados da seguinte maneira:

Ponto	Ambiente de Referenciamento
1	x e y de sub1, a e b de example
2	x de sub3, (o x de sub2 está oculto), a e b de example
3	x de sub2, a e b de example
4	a e b de example

Considere, agora, as declarações de variáveis desse programa. Primeiro, note que, não obstante o escopo de sub1 estar em um nível mais elevado (aninhado menos profundamente) do que sub3, o escopo de sub1 não é um ancestral estático de sub3, de modo que sub3 não tem acesso às variáveis declaradas em sub1. Existe uma boa razão para isso. As variáveis declaradas em sub1 são dinâmicas na pilha, e não são vinculadas ao armazenamento se sub1 não estiver em execução. Uma vez que sub3 pode estar em execução quando sub1 não estiver, ela não pode receber permissão para acessar variáveis em sub1, as quais, não necessariamente, estariam vinculadas ao armazenamento durante a execução de sub3.

Um subprograma é **ativo** se sua execução tiver começado, mas ainda não tiver terminado. O ambiente de referenciamento de uma instrução em uma linguagem de escopo dinâmico são as variáveis declaradas localmente, mais as variáveis de todos os outros subprogramas atualmente ativos. Mais uma vez, algumas variáveis em subprogramas ativos podem ser ocultadas do ambiente de referenciamento. As ativações de subprogramas recentes podem ter declarações para variáveis que ocultam as de mesmos nomes existentes nas ativações anteriores.

Considere o seguinte programa de exemplo: suponhamos que as únicas chamadas a funções sejam main chama sub2, que chama sub1.

```

void sub1 () {
    int a, b;
    ... ←—————1
} /* end of sub1 */
void sub2() {
    int b, c;
    ... ←—————2
    sub1;
} /* end of sub2 */
void main() {
    int c, d;
    ... ←—————3
    sub2();
} / end of main */

```

O ambiente de referenciamento dos pontos de programa indicados apresentam-se da seguinte maneira:

Ponto	Ambiente de Referenciamento
1	a e b de sub1, c de sub2, d de main, (c de main e b de sub2 estão ocultos)
2	b e c de sub2, d de main, (c de main está oculto)
3	c e d de main

## 5.11 Constantes Nomeadas

Uma **constante nomeada** é uma variável vinculada a um valor somente no momento em que ela é vinculada a um armazenamento; seu valor não pode ser mudado pela instrução de atribuição ou por uma instrução de entrada. As constantes nomeadas são úteis como auxílios para a legibilidade e para a confiabilidade do programa. A legibilidade pode ser melhorada, por exemplo, usando-se o nome pi em vez da constante 3,14159.

Outro uso vantajoso das constantes nomeadas está em programas que processam um número fixo de valores de dados, digamos 100. Esses programas normalmente usam a constante 100 em uma grande quantidade de localizações para declarar faixas de subscripto de matriz, para limites de controle de laço e para outros usos. Considere o seguinte segmento de programa esquemático Java:

```
void exemplo() {
    int [] ListaInt = new int [100];
    String [] ListaCar = new String [100];
    ...
    for (indice = 0; indice < 100; indice++) {
        ...
    }
    ...
    for (indice = 0; indice < 100; indice++) {
        ...
    }
    ...
    media = soma / 100;
    ...
}
```

Quando esse programa precisar ser modificado para lidar com um número diferente de valores de dados, todas as ocorrências de 100 devem ser encontradas e mudadas. Em um programa grande, isso pode ser tedioso e propenso a erros. Um método mais fácil e mais confiável é usar uma constante nomeada, como em

```
void exemplo() {
    final int tamanho = 100;
    int [] ListaInt = new int [tamanho];
    String [] ListaCar = new String [tamanho];
```

```
...
for (indice = 0; indice < tamanho; indice++) {
...
}
...
for (indice = 0; indice < tamanho; indice++) {
...
}
...
media = soma / tamanho;
...
}
```

Agora, quando o tamanho precisar ser mudado, somente uma linha deverá ser mudada, independentemente do número de vezes que ela é usada no programa. Esse é outro exemplo dos benefícios da abstração. O nome `tamanho` é uma abstração do número de elementos em alguns vetores, e o número de iterações em alguns laços. Isso ilustra como as constantes nomeadas podem auxiliar na capacidade de modificação.

As declarações de constantes nomeadas Pascal exigem valores simples no lado direito do operador `=`. Porém, Modula-2 e FORTRAN 90 permitem o uso de expressões constantes que podem conter constantes nomeadas anteriormente declaradas, valores constantes e operadores. O motivo para a restrição a constantes e a expressões constantes em Pascal e em Modula-2, respectivamente, é que ambas usam vinculação estática de valores a constantes nomeadas. Estas últimas em linguagens que usam vinculação estática de valores, às vezes, são chamadas de **constantes manifestas**.

A Ada, o C++ e o Java permitem vinculação dinâmica de valores para constantes nomeadas. Isso faz com que expressões com variáveis sejam atribuídas a constantes nas declarações. Por exemplo, a instrução C++

```
const int max = 2 * WIDTH + 1;
```

declara que `MAX` deve ser uma constante nomeada do tipo inteiro cujo valor é tirado da expressão `2 * WIDTH + 1`, em que o valor da variável `WIDTH` deve ser visível quando `MAX` for alocado e vinculado ao seu valor.

A Ada também permite constantes nomeadas dos tipos enumeração e estruturados, que serão discutidos no Capítulo 6.

## 5.12 Inicialização de Variáveis

---

A discussão sobre a vinculação de valores a constantes nomeadas leva naturalmente ao tópico da inicialização de variáveis, porque a vinculação de um valor a uma constante nomeada é o mesmo processo, exceto que é permanente.

Em muitos casos, é conveniente que as variáveis tenham valores antes que o código do programa ou do subprograma no qual elas estão declaradas comece a executar. A vinculação de uma variável a um valor no momento em que ela é vinculada ao armazenamento é chamada **inicialização**. Se a variável estiver estaticamente vinculada ao armazenamento,

a vinculação e a inicialização ocorrerão antes da execução. Se a vinculação ao armazenamento for dinâmica, a inicialização também será dinâmica.

No FORTRAN, os valores iniciais das variáveis podem ser especificados em uma instrução DATA, como em

```
REAL PI
INTEGER SOMA
DATA SOMA /0/, PI /3.14159/
```

que inicializa SOMA em zero e PI em 3,14159. As inicializações reais desenvolvem-se no momento da compilação. Assim que a execução inicia-se, SOMA e PI são como quaisquer outras variáveis.

Em muitas linguagens, os valores iniciais das variáveis podem ser especificados na instrução de declaração, como na declaração C++

```
int soma = 0;
```

Nem o Pascal, nem o Modula-2 oferecem uma maneira de inicializar variáveis, exceto durante a execução com instruções de atribuição.

Em geral, a inicialização ocorre uma única vez para variáveis estáticas, mas é feita a cada alocação para as alocadas dinamicamente, como, por exemplo, as locais em uma função C++.

## RESUMO

A forma dos nomes de uma linguagem podem ter um impacto tanto sobre a sua legibilidade como sobre a sua capacidade de escrita. A relação de nomes com palavras especiais, que são palavras reservadas ou palavras-chave, também é uma decisão importante de projeto.

As variáveis podem ser caracterizadas pelo sextuplo de atributos: nome, endereço, valor, tipo, tempo de vida e escopo.

Os apelidos são dois ou mais nomes vinculados ao mesmo endereço de armazenamento. Eles são considerados prejudiciais para a legibilidade, mas difíceis de serem eliminados completamente de uma linguagem.

A vinculação é a associação de atributos a entidades do programa. O conhecimento dos tempos de vinculação de atributos a entidades é fundamental para que se possa entender a semântica das linguagens de programação. A vinculação pode ser estática ou dinâmica. As declarações, explícitas ou implícitas, constituem um meio de especificar a vinculação estática de variáveis a tipos. Em geral, a vinculação dinâmica permite uma maior flexibilidade, mas à custa da legibilidade, da eficiência e da confiabilidade.

As variáveis escalares podem ser separadas em quatro categorias, considerando seus tempos de vida. Essas categorias são: estática, dinâmica na pilha, dinâmica no monte explícita e dinâmica no monte implícita.

A tipificação forte é o conceito de exigir que todos os erros de tipos sejam detectados. A sua vantagem é o aumento da legibilidade.

As regras de compatibilidade de tipos de uma linguagem têm um efeito importante sobre as operações fornecidas para os seus valores. A compatibilidade de tipos geralmente é definida em termos de compatibilidade de nome ou de estrutura.

O escopo estático é um recurso fundamental do ALGOL 60 e da maioria de suas descendentes. Ele constitui um método eficiente de permitir visibilidade de variáveis

não-locais em subprogramas. O escopo dinâmico proporciona mais flexibilidade do que o escopo estático mas, novamente, à custa de legibilidade, de confiabilidade e de eficiência.

O ambiente de referenciamento de uma instrução é o conjunto de todas as variáveis visíveis para a instrução.

As constantes nomeadas são, simplesmente, variáveis vinculadas a valores somente quando elas são vinculadas ao armazenamento. A inicialização é a vinculação de uma variável a um valor no momento em que esta é vinculada ao armazenamento.

## **QUESTÕES DE REVISÃO**

---

1. Quais são as questões de projeto referentes a nomes?
2. Qual é o perigo potencial dos nomes que fazem distinção entre maiúsculas e minúsculas?
3. De que maneira as palavras reservadas são melhores do que as palavras-chave?
4. O que é um apelido?
5. Quais categorias de variáveis de referência do C++ são sempre apelidos?
6. Qual é o valor-l de uma variável? Qual é o valor-r?
7. Defina vinculação e tempo de vinculação.
8. Depois do projeto e da implementação da linguagem, quais são os quatro momentos em que vinculações podem desenvolver-se em um programa?
9. Defina vinculação estática e vinculação dinâmica.
10. Quais são as vantagens e as desvantagens das declarações implícitas?
11. Quais são as vantagens e as desvantagens da vinculação dinâmica de tipos?
12. Defina variáveis estáticas, dinâmicas na pilha, dinâmicas no monte explícitas e dinâmicas no monte implícitas. Quais são as vantagens e as desvantagens de cada uma?
13. Defina coerção, erro de tipo, verificação de tipos e tipificação forte.
14. Defina compatibilidade de tipo de nome e compatibilidade de tipo de estrutura. Quais são os méritos relativos de ambas?
15. Qual é a diferença entre um tipo derivado Ada e um subtipo Ada?
16. Defina tempo de vida, escopo, escopo estático e escopo dinâmico.
17. Como uma referência a uma variável não-local em um programa de escopo estático é vinculada à sua definição?
18. Qual é o problema geral com o escopo estático?
19. Qual é o ambiente de referenciamento de uma instrução?
20. O que é um ancestral estático de um subprograma? O que é um ancestral dinâmico de um subprograma?
21. O que é um bloco?
22. Quais são as vantagens e as desvantagens do escopo dinâmico?
23. Quais são as vantagens das constantes nomeadas?

## **PROBLEMAS**

---

1. Decida qual das seguintes formas de identificador é a mais legível e sustente essa decisão:  
SomaDeVendas  
soma\_de\_vendas  
SOMADEVENDAS
2. Algumas linguagens de programação são sem tipo. Quais são as vantagens e as desvantagens evidentes de não se ter tipos em uma linguagem?
3. Uma utilização comum de EQUIVALENCE do FORTRAN é a seguinte: um vetor grande de valores numéricos é colocado à disposição de um subprograma como um parâmetro. O vetor contém muitas variáveis diferentes não-relacionadas, em vez de uma coleção de repetições da

mesma variável. Ele é representado como um vetor para reduzir o número de nomes que precisam ser passados como parâmetros. Dentro de um subprograma, uma instrução EQUIVALENTE extensa é usada para criar nomes conotativos como apelidos para os vários elementos do vetor, o que aumenta a legibilidade do código do subprograma. Essa é uma boa ideia ou não? Quais alternativas para o uso de apelidos estão disponíveis?

4. Escreva uma instrução de atribuição simples com um operador aritmético em alguma linguagem que você conheça. Para cada componente da instrução, liste as várias vinculações que são necessárias para determinar a semântica quando a instrução é executada. Para cada vinculação, indique o tempo de vinculação usado para a linguagem.
5. A vinculação dinâmica de tipos está estreitamente relacionada com as variáveis dinâmicas no monte implícitas. Explique essa relação.
6. Descreva a situação em que uma variável sensível à história em um subprograma é útil.
7. Pesquise a definição de fortemente tipificada apresentada em Gehani (1983) e compare-a com a definição dada neste capítulo. De que maneira elas diferem?
8. Considere o seguinte programa Pascal esquemático.

```
program main;
  var x : integer;
  procedure sub3; forward;
  procedure sub1;
    var x : integer;
    procedure sub2;
      begin { sub2 }
      ...
      end; { sub2 }
    begin { sub1 }
    ...
    end; { sub1 }
  procedure sub3;
    begin { sub3 }
    ...
    end; { sub3 }
  begin { main }
  ...
  end. { main }
```

Suponhamos que a execução desse programa seja na seguinte ordem de unidades:

main chama sub1  
sub1 chama sub2  
sub2 chama sub3

- a. Supondo que esteja presente o escopo estático, qual declaração de x é a correta para a referência a x em:
  - i. sub1
  - ii. sub2
  - iii. sub3
- b. Repita a parte a, mas suponha a presença do escopo dinâmico.
9. Suponhamos que o seguinte programa tenha compilado e executado usando regras de escopo estático. Qual valor de x é impresso no procedimento sub1? De acordo com as regras de escopo dinâmico, qual valor de x é impresso no procedimento sub1?

```
program main;
  var x : integer;
  procedure sub1;
    begin { sub1 }
```

```
writeln('x =', x)
end; { sub1 }
procedure sub2;
var x : integer;
begin { sub2 }
x := 10;
sub1
end; { sub2 }
begin { main }
x := 5;
sub2
end. { main }
```

10. Considere o seguinte programa:

```
program main;
var x, y, z : integer;
procedure sub1;
var a, y, z : integer;
procedure sub2;
var a, b, z : integer;
begin { sub2 }
...
end; { sub2 }
begin { sub1 }
...
end; { sub1 }
procedure sub3;
var a, x, w : integer;
begin { sub3 }
...
end; { sub3 }
begin { main }
...
end. { main }
```

Liste todas as variáveis, juntamente com as unidades de programa em que elas foram declaradas que são visíveis nos corpos de sub1, sub2 e sub3, supondo que seja usado o escopo estático.

11. Considere o seguinte programa.

```
program main;
var x, y, z : integer;
procedure sub1;
var a, y, z : integer;
begin { sub1 }
...
end; { sub1 }
procedure sub2;
var a, x, w : integer;
procedure sub3;
var a, b, z : integer;
begin { sub3 }
...
end; { sub3 }
begin { sub2 }
...
end. { sub2 }
```

```

    end; { sub2 }
begin { main }
...
end. { main }

```

Liste todas as variáveis, juntamente com as unidades de programa em que elas estão declaradas, que são visíveis nos corpos de sub1, sub2 e sub3, supondo que seja usado o escopo estático.

12. Considere o seguinte programa C:

```

void fun(void) {
    int a, b, c; /* definição 1 */
    ...
    while (...) {
        int b, c, d; /* definição 2 */
        ... ←—————1
        while (...) {
            int c, d, e; /* definição 3 */
            ... ←—————2
            }
        ... ←—————3
    }
    ... ←—————4
}

```

Para cada um dos quatro pontos marcados nessa função, liste cada variável visível, juntamente com o número da instrução de definição que a define.

13. Considere o seguinte programa C esquemático:

```

void fun1(void); /* protótipo */
void fun2(void); /* protótipo */
void fun3(void); /* protótipo */
void main() {
    int a, b, c;
    ...
}
void fun1(void) {
    int b, c, d;
    ...
}
void fun2(void) {
    int c, d, e;
    ...
}
void fun3(void) {
    int d, e, f;
    ...
}

```

Dadas as seguintes seqüências de chamada e supondo-se que seja usado o escopo dinâmico, quais variáveis são visíveis durante a execução da última função chamada? Inclua, em cada variável visível, o nome da função em que 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.

14. Considere o seguinte programa:

```
program main;
  var x, y, z : integer;
  procedure sub1;
    var a, y, z : integer;
    begin { sub1 }
    ...
  end; { sub1 }
  procedure sub2;
    var a, b, z : integer;
    begin { sub2 }
    ...
  end; { sub2 }
  procedure sub3;
    var a, x, w : integer;
    begin { sub3 }
    ...
  end; { sub3 }
begin { main }
...
end. { main }
```

Dadas as seguintes seqüências de chamada e supondo-se que seja usado o escopo dinâmico, quais variáveis são visíveis durante a execução do último subprograma ativado? Inclua, em cada variável visível, o nome da unidade em que 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.

# Capítulo 6

## Tipos de Dados



### James Gosling

James Gosling, vice-presidente e membro da Sun Microsystems, desenvolveu o projeto do Java e implementou o primeiro compilador e a máquina virtual para ele. Gosling também serviu como engenheiro-chefe do sistema de janelas NeWS e projetou o editor de texto EMACS.

- 6.1** Introdução
- 6.2** Tipos de Dados Primitivos
- 6.3** Tipos Cadeia de Caracteres
- 6.4** Tipos Ordinais Definidos pelo Usuário
- 6.5** Tipos Matriz
- 6.6** Matrizes Associativas
- 6.7** Tipos Registro
- 6.8** Tipos União
- 6.9** Tipos Conjunto
- 6.10** Tipos Ponteiro

Este capítulo apresenta, em primeiro lugar, o conceito de tipo de dados e as características dos tipos de dados primitivos comuns. Depois, serão discutidos os projetos de tipos de enumeração e de subfaixa. Em seguida, os tipos de dados estruturados, especificamente, as matrizes, os registros e as uniões serão investigados. Os conjuntos serão, então, discutidos, seguidos de um exame em profundidade dos ponteiros.

Para cada uma das várias categorias de tipos de dados, serão declaradas as questões de projeto, e serão explicadas as opções feitas pelos projetistas das linguagens importantes. Esses projetos são então avaliados.

Métodos de implementação para tipos de dado têm freqüentemente um impacto significativo em seu projeto. Portanto, a implementação dos vários tipos de dados é outra parte importante deste capítulo, especialmente para a implementação de matrizes.

## 6.1 Introdução

Os programas de computador produzem resultados manipulando dados. Um fator importante para determinar a facilidade com que eles podem executar tal tarefa é quanto bem os tipos de dados coincidem com o espaço de problema do mundo real. Portanto, é crucial que uma linguagem suporte uma variedade de tipos de dados e de estruturas.

Os conceitos contemporâneos de tipificação de dados desenvolveram-se ao longo dos últimos 45 anos. Nas primeiras linguagens, as estruturas de dados do espaço de problema precisavam ser modeladas somente com algumas estruturas básicas suportadas na linguagem. Por exemplo, nos FORTRANs anteriores ao 90, as listas encadeadas e as árvores binárias comumente são modeladas com matrizes.

As estruturas de dados do COBOL deram o primeiro passo para sair do modelo FORTRAN I ao permitirem que os programadores especificassem a precisão dos valores de dados decimais e, também, por fornecerem um tipo de dados estruturado para registros de informações. A PL/I estendeu a capacidade de especificação da precisão para tipos inteiros e reais. Desde então, isso foi incorporado à Ada e ao FORTRAN 90. Os projetistas da PL/I incluíram muitos tipos de dados, com a intenção de suportar uma grande variedade de aplicações. Uma abordagem melhor, introduzida no ALGOL 68, é fornecer alguns tipos básicos e alguns operadores de definição de estrutura flexíveis que permitam ao programador modelar uma estrutura para o problema que está enfrentando com tipos de dados definidos pelo usuário. Esse foi claramente um dos mais importantes avanços em termos de evolução do projeto de tipos de dados. Os tipos definidos pelo usuário oferecem uma melhor legibilidade pelo uso de nomes de tipos significativos. Eles permitem a verificação de tipos das variáveis de uma categoria especial de uso, a qual, de outra forma, não seria possível. Também favorecem a "modificabilidade": o programador pode mudar o tipo de uma categoria de variáveis em um programa alterando somente uma instrução de declaração de tipo.

A Ada 83 incorporou os conceitos contemporâneos de projeto de tipos de dados do final da década de 70, que resultaram de uma extensão natural da idéia dos definidos pelo usuário. A filosofia destes últimos é que o usuário deve ter permissão para criar um tipo único para cada classe única de variáveis no espaço de problema. Além disso, a linguagem deve impor a singularidade dos tipos, os quais são, de fato, abstrações das variáveis do espaço de problema. Eis um conceito poderoso que tem um impacto significativo sobre o processo global do projeto de software. Levando-o a um passo adiante, chegamos aos tipos de dados abstratos, que podem ser simulados na Ada 83. A idéia fundamental deste tipo é

que o uso de um tipo é separado da representação e do conjunto de operações com valores deste último. Todos os tipos fornecidos por uma linguagem de programação de alto nível são abstratos. Os tipos de dados abstratos definidos pelo usuário serão discutidos detalhadamente no Capítulo 11.

Os dois tipos de dados estruturados (não-escalares) mais comuns são as matrizes e os registros. Esses e alguns outros tipos de dados são especificados pelos operadores de tipos ou de construtores, usados para formar expressões de tipos. Por exemplo, no C, os operadores de tipos são colchetes, parênteses e asteriscos, usados para especificar vetores, funções e ponteiros.

É conveniente, tanto lógica como concretamente, pensar nas variáveis em termos de descritores. Um **descritor** é o conjunto dos atributos de uma variável. Em uma implementação, um descritor é um conjunto de células de memória que armazenam atributos de variáveis. Se os atributos forem todos estáticos, os descritores serão exigidos somente no momento de compilar. Eles são construídos pelo compilador, usualmente como uma parte da tabela de símbolos, e usados durante a compilação. Para atributos dinâmicos, entretanto, parte do descritor ou todo ele deve ser mantido durante a execução. Assim, o descritor é usado pelo sistema em tempo de execução. Em todos os casos, os descritores são usados para verificação de tipos e pelas operações de alocação e de desalocação.

A palavra “objeto” freqüentemente é associada ao valor de uma variável e ao espaço que ela ocupa. Neste livro, entretanto, reservamos a palavra objeto exclusivamente para instâncias de tipos de dados abstratos definidos pelo usuário, em vez de usá-la, também, para os valores de variáveis de tipos predefinidos. Em linguagens orientadas a objeto, toda instância de classe, seja predefinida ou definida pelo usuário, é chamada de objeto. Os objetos serão discutidos detalhadamente nos Capítulos 11 e 12.

Nas seções seguintes, todos os tipos de dados comuns serão discutidos. Para a maioria, as questões de projeto particulares ao tipo são apresentadas. Para todas, um ou mais projetos de exemplo são descritos. Uma questão de projeto é fundamental a todos os tipos de dados: quais 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 tipos são chamados **tipos de dados primitivos**. Quase todas as linguagens de programação oferecem um conjunto destes últimos. Alguns dos tipos de dados primitivos são simplesmente reflexos de hardware; por exemplo, os tipos inteiros. Outros exigem somente um pequeno suporte de software para sua implementação.

Os tipos de dados primitivos de uma linguagem são usados, juntamente com um ou com mais construtores de tipos, para fornecer os tipos estruturados.

### 6.2.1 Tipos Numéricos

Muitas das primeiras linguagens de programação tinham somente tipos primitivos numéricos que ainda desempenham um papel fundamental entre os suportados pelas linguagens contemporâneas.

### 6.2.1.1 Inteiro

O tipo de dados primitivo numérico mais comum é o **inteiro**. Atualmente, muitos computadores suportam diversos tamanhos de inteiros, e essas capacidades são refletidas em algumas linguagens de programação. Por exemplo, a Ada permite que as implementações incluam até três tamanhos de inteiros: `SHORT INTEGER`, `INTEGER` e `LONG INTEGER`. Algumas linguagens, como o C++, incluem tipos inteiros sem sinal, usados freqüentemente com dados binários.

Um valor inteiro é representado em um computador por uma cadeia de bits, com um dos bits tipicamente na extrema esquerda, representando o sinal. Tipos inteiros são suportados diretamente pelo hardware.

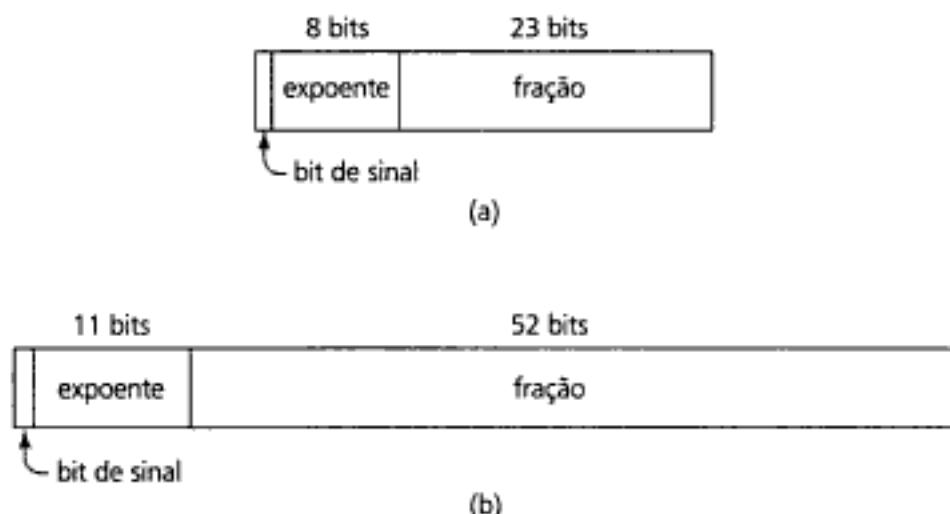
Um inteiro negativo poderia ser armazenado em uma notação de sinal-magnitude, na qual o bit de sinais é fixado para indicar o valor negativo, enquanto o restante da cadeia de bits representa o valor absoluto do número. A notação de sinal-magnitude, entretanto, não se presta para a aritmética computadorizada. A maioria dos computadores usa agora uma notação chamada complemento de dois para armazenar números inteiros negativos, o que é conveniente para a adição e para a subtração. Na notação de complemento de dois, a representação de um número inteiro negativo é formada, tomando-se o complemento lógico da versão positiva do número e adicionando-se um. A notação de complemento de um ainda é usada por alguns computadores. Na notação de complemento de um, o valor negativo de um número inteiro é armazenado como o complemento lógico de seu valor absoluto. A notação de complemento de um tem a desvantagem de possuir duas representações de zero. Leia qualquer livro sobre programação em linguagem de montagem para obter detalhes sobre as representações de inteiros.

### 6.2.1.2 Vírgula-Flutuante

Os tipos de dados com vírgula-flutuante modelam os números reais, mas as representações são somente aproximações para a maioria dos números reais. Por exemplo, nenhum dos números fundamentais  $\pi$  ou  $e$  (a base dos logaritmos naturais) pode ser representado corretamente na notação de vírgula-flutuante. Logicamente, nenhum desses pode ser representado com exatidão em qualquer espaço finito. Na maioria dos computadores, os números com vírgula-flutuante são armazenados em dígitos binários, o que exacerba o problema. Por exemplo, mesmo o valor 0,1 em decimal não pode ser representado por um número finito de dígitos binários. Outro problema é a perda de exatidão nas operações aritméticas. Para obter mais informação sobre os problemas da notação de vírgula-flutuante, leia Knuth (1981).

Os valores de vírgula-flutuante são representados como frações e como expoentes, uma forma emprestada da notação científica. Os computadores mais antigos usavam uma variedade de diferentes representações para valores de vírgula-flutuante. Porém, a maioria das máquinas mais novas usa o formato IEEE Floating-Point Standard 754. Os implementadores de linguagens usam qualquer representação suportada pelo hardware. A maioria das linguagens inclui dois tipos de vírgula-flutuante, freqüentemente chamados de **float** e **double**. O **float** é o tamanho-padrão, normalmente armazenado em quatro bytes de memória. O **double** é fornecido para situações em que se necessitam partes fracionárias maiores. As variáveis com precisão **double** normalmente ocupam duas vezes mais espaço do que as **float** e oferecem, pelo menos, o dobro do número de bits da fração.

O conjunto de valores que podem ser representados por um tipo de vírgula-flutuante é definido em termos de precisão e de faixa. Precisão é a exatidão da parte fracionária de um valor, medida como o número de bits. Faixa é uma combinação da faixa de frações e, o que é mais importante, da faixa de expoentes.



**FIGURA 6.1** Formatos de vírgula-flutuante IEEE: (a) precisão única, (b) precisão dupla.

A Figura 6.1 mostra o formato IEEE *Floating-Point Standard 754* para a representação da precisão única e dupla (IEEE, 1985). Detalhes dos formatos IEEE podem ser encontrados em Tanenbaum (1990).

#### 6.2.1.3 Decimais

A maioria dos grandes computadores projetados para suportar aplicações de sistemas comerciais têm suporte de hardware para tipos de dados **decimais**, que armazenam um número fixo de dígitos decimais, com a vírgula decimal em uma posição fixa no valor. Esses são os tipos de dados primários para processamento de dados comerciais e, portanto, são fundamentais para o COBOL.

Os tipos decimais têm a vantagem da capacidade de armazenar, com precisão, valores decimais, pelo menos os de dentro de uma faixa restrita, o que não pode ser feito em vírgula-flutuante. A desvantagem é que a faixa de valores é restrita porque não se permite nenhum expoente, e a representação dos mesmos na memória é um desperdício.

Os tipos decimais são armazenados de uma maneira muito semelhante à das cadeias de caracteres, usando códigos binários para os dígitos decimais. Essas representações são chamadas decimais codificadas em binário (*binary coded decimal* — BCD). Em alguns casos, elas são armazenadas com um dígito por byte, mas em outros são compactadas com dois dígitos por byte. De qualquer uma das maneiras, elas ocupam mais espaço do que as representações binárias. São necessários pelo menos 4 bits para codificar um dígito decimal. Portanto, para armazenar um número decimal codificado de seis dígitos é preciso 24 bits de memória. Porém, são necessários somente 20 bits para armazenar o mesmo número em dígitos binários. As operações com valores decimais são feitas em hardware em máquinas que têm essas capacidades; caso contrário, são simuladas em software.

#### 6.2.2 Tipos Booleanos

Os tipos **booleanos** são, talvez, os mais simples de todos. Sua faixa de valores tem somente dois elementos: um para verdadeiro e um para falso. Eles foram introduzidos no ALGOL 60 e têm sido incluídos na maioria das linguagens de propósito geral projetadas desde

1960. Uma exceção popular é o C, no qual expressões numéricas podem ser usadas como condicionais. Nessas expressões, todos os operandos com valores diferentes de zero são considerados verdadeiros, e zero é considerado falso. Ainda que o C++ tenha um tipo booleano, ele também permite que expressões numéricas sejam usadas como se fossem booleanas.

Tipos booleanos freqüentemente são usados para representar comutadores (*switches*) ou sinalizadores<sup>1</sup> em programas. Mesmo que outros tipos, como o inteiro, possam ser usados para essas finalidades, o uso de tipos booleanos aumenta a legibilidade.

Um valor booleano poderia ser representado por um único bit, mas, uma vez que um único bit de memória é difícil de ser acessado com eficiência em muitas máquinas, freqüentemente eles são armazenados na menor célula de memória eficientemente endereçável, tipicamente um byte.

### 6.2.3 Tipos Caractere

Os dados de caracteres são armazenados nos computadores como codificações numéricas. A codificação mais comumente usada é a ASCII (*American Standard Code for Information Interchange*), que usa valores de 0 a 127 para codificar 128 diferentes caracteres. Para fornecer o meio de processar codificações de caracteres únicos, muitas linguagens de programação incluem um tipo primitivo para elas.

Por causa da globalização dos negócios e da necessidade dos computadores comunicarem-se com outros computadores mundo afora, o conjunto de caracteres ASCII está tornando-se rapidamente inadequado. Um novo conjunto de caracteres de 16 bits chamado Unicode foi desenvolvido recentemente como uma alternativa e inclui os caracteres da maioria das linguagens naturais internacionais. Por exemplo, o Unicode inclui o alfabeto cirílico, usado na Sérvia, e os dígitos tailandeses. O Java é a primeira linguagem amplamente usada a introduzir o conjunto de caracteres Unicode, mas ele certamente chegará a outra linguagem de programação popular em breve.

## 6.3 Tipos Cadeia de Caracteres

---

Um **tipo cadeia de caracteres** é aquele cujos valores consistem em sequências de caracteres. Cadeias constantes são usadas para rotular a saída; e a entrada e saída de todos os tipos de dados freqüentemente é feita em termos de cadeias. Logicamente, as cadeias de caracteres também são um tipo fundamental para todos os programas com manipulação de caracteres.

### 6.3.1 Questões de Projeto

As duas questões de projeto mais importantes específicas aos tipos cadeias de caracteres são as seguintes:

<sup>1</sup>N. de T. Sinalizador: flag. Um indicador "sim/não" incorporado em determinado hardware, ou criado e controlado pelo programador.

- As cadeias devem ser, simplesmente, um tipo especial de vetor de caracteres ou de um tipo primitivo (sem nenhuma operação de subscrever,\* ao estilo dos vetores)?
- As cadeias devem ter um tamanho estático ou dinâmico?

### 6.3.2 As Cadeias e Suas Operações

Se as cadeias não forem definidas como um tipo primitivo, os seus dados normalmente serão armazenados em vetores de caracteres únicos e referenciados como tal na linguagem. Essa é a abordagem seguida pelo Pascal, pelo C, pelo C++ e pela Ada.

Na Ada, **STRING** é um tipo que é predefinido como uma matriz unidimensional de elementos **CHARACTER**. Referência a subcadeias, a concatenação, operadores relacionais e a atribuição são oferecidos para os tipos **STRING**. A referência à subcadeia permite que qualquer subcadeia de determinada cadeia seja tratada como um valor em uma referência ou como uma variável em uma atribuição. Uma referência àquela é denotada pela faixa de números inteiros entre parênteses, que indica a subcadeia desejada pela posição do caractere. Por exemplo,

```
NAME1(2:4)
```

especifica a subcadeia que consiste no segundo, no terceiro e no quarto caracteres do valor em NAME1.

A concatenação de cadeias de caracteres na Ada é uma operação especificada pelo E comercial (&). A instrução seguinte concatena NAME2 à extremidade direita de NAME1:

```
NAME1 := NAME1 & NAME2;
```

Por exemplo, se NAME1 tiver a cadeia "PEACE" e NAME2 tiver "FUL", depois que a instrução de atribuição for executada, NAME1 terá a cadeia "PEACEFUL".

O C e o C++ usam vetores **char** para armazenar cadeias de caracteres e para oferecer uma coleção de operações com cadeias por meio de uma biblioteca-padrão cujo arquivo de cabeçalhos é **string.h**. A maior parte dos usos de cadeias e a maioria das funções de biblioteca usam a convenção segundo a qual as cadeias de caracteres são finalizadas com um caractere especial nulo, que é representado por zero. Essa é uma alternativa para manter o tamanho da variável cadeia. As operações de biblioteca simplesmente executam suas operações até que o caractere nulo apareça na cadeia sobre a qual se está operando. As funções de biblioteca que constroem cadeias freqüentemente fornecem o caractere nulo. Os literais das cadeias de caracteres construídas pelo compilador têm o caractere nulo. Por exemplo, considere a seguinte declaração:

```
char *str = "apples";
```

Nesse exemplo, str é um ponteiro **char** definido para apontar para a cadeia de caracteres, apples, em que 0 é o caractere nulo. Essa inicialização de str é válida porque os literais da cadeia de caracteres são representados pelos ponteiros **char**, em vez de pela cadeia.

Algumas das funções de biblioteca mais comumente usadas para cadeias de caracteres em C e em C++ são **strcpy**, que move cadeias; **strcat**, que concatena duas cadeias

\*N. de R.T. Subscrever uma cadeia, no caso, significa acessá-la através de índices, como em cadeia[4], que acessaria o quarto caractere (ou o quinto em linguagens, a exemplo do C, que iniciam com a posição zero).

dadas; `strcmp`, que compara lexicograficamente (pela ordem de seus códigos) duas cadeias dadas; e `strlen`, que retorna o número de caracteres, sem contar o nulo, na cadeia dada. Os parâmetros e os valores de retorno para a maioria das funções de manipulação de cadeias são os ponteiros `char` que apontam para vetores `char`. Parâmetros também podem ser literais de cadeia.

O FORTRAN 77, o 90 e o BASIC tratam as cadeias como um tipo primitivo e oferecem atribuição, operadores relacionais, concatenação e operações de referência a subcadeia para elas.

Em Java, as cadeias são suportadas como um tipo primitivo pela classe `String`, cujos valores são cadeias constantes, e pela classe `StringBuffer`, cujos valores são mutáveis e assemelham-se muito mais a vetores de caracteres. Subscrever é permitido nas variáveis `StringBuffer`.

Em geral, tanto as operações de atribuição como as de comparação para cadeias de caracteres são complicadas pela possibilidade de atribuir-se e de comparar operandos de tamanhos diferentes. Por exemplo, o que acontece quando uma cadeia mais longa é atribuída a uma menor ou vice-versa? Normalmente, escolhas simples e sensatas são feitas nessas situações, mesmo que os usuários tenham problemas para lembrar-se delas.

Casar padrões também é outra operação fundamental de cadeias de caracteres. Muitas vezes, ela é fornecida por uma função de biblioteca, não como uma operação na linguagem. Há duas exceções importantes, uma das quais é o SNOBOL4, que tem uma elaborada operação de casamento de padrões incorporada àquela. O SNOBOL4 é, provavelmente, a linguagem de manipulação de cadeias definitiva.

Padrões de cadeias no SNOBOL4 são expressões que podem ser atribuídas a variáveis. Por exemplo, considere o seguinte:

```
LETRA = 'abcdefghijklmnopqrstuvwxyz'
WORDPAT = BREAK(LETRA) SPAN(LETRA) . WORD
```

`LETRA` é uma variável com o valor de uma cadeia contendo todas as letras minúsculas. `WORDPAT` é um padrão que descreve palavras da seguinte maneira: primeiro, pule até que uma letra seja encontrada, depois continue com as letras até que uma não-letra seja encontrada. Tal padrão também inclui um operador `". "`, o qual especifica que a cadeia coincidente com o padrão seja atribuída à variável `WORD`.

Esse padrão pode ser usado na instrução

```
TEXT WORDPAT
```

a qual tenta encontrar uma cadeia de letras no valor da variável `TEXT`.

A segunda linguagem importante que inclui operações de casamento de padrões incorporadas é a Perl. Nesse caso, as expressões de casamento de padrões baseiam-se de maneira bastante livre em expressões regulares matemáticas. De fato, muitas vezes elas são chamadas de expressões regulares. Elas evoluíram a partir do antigo editor de linhas UNIX, `ed`, e tornaram-se parte das linguagens de `shell` UNIX. Por fim, elas se desenvolveram para sua forma mais complexa na Perl. Seria necessário um capítulo inteiro de um livro sobre a Perl para explicar essas expressões. De fato, agora há um livro completo sobre esse tipo de expressões de casamento de padrões (Friedl, 1997). Nesta seção, apresentamos somente uma breve introdução ao estilo de tais expressões com a ajuda de dois exemplos relativamente simples. Considere o seguinte padrão:

```
/[A-Za-z][A-Za-z\d]+/
```

Ele descreve a forma de nome típico das linguagens de programação. Os colchetes envolvem classes de caracteres. A primeira classe especifica todas as letras; a segunda especifica todas as letras e todos os dígitos (um dígito é especificado com a abreviação \d). Se somente a segunda classe fosse incluída, não poderíamos evitar que um nome começasse com um dígito. O operador de mais, que vem depois da segunda categoria, especifica que deve haver um ou mais daquilo que está nesta categoria. Assim, o padrão inteiro casa cadeias que se iniciam com uma letra, seguida de uma ou mais letras ou dígitos.

Em seguida, considere a seguinte expressão de padrão:

```
/\d+\.,?\d*|\.,0\d+|
```

Este padrão casa literais numéricos. O \., especifica a vírgula decimal literal. O ponto de interrogação quantifica aquilo que vem depois para que apareça zero ou uma vez. A barra vertical ( | ) separa duas alternativas no padrão inteiro. A primeira casa as cadeias de um ou mais dígitos, possivelmente seguidos de uma vírgula decimal, seguido de zero ou mais dígitos; a segunda casa as cadeias que se iniciam com zero e uma vírgula decimal seguidos de um ou mais dígitos.

O estilo de padrões da Perl foi recentemente incorporado ao JavaScript.

### 6.3.3 Opções de Tamanho da Cadeia

Existem diversas opções de projeto, quanto ao tamanho dos valores da cadeia. Primeiro, o tamanho pode ser estático e especificado na declaração. Esse tipo é chamado **cadeia de tamanho estático**. Essa é a opção existente nas linguagens FORTRAN 90, COBOL, Pascal e Ada. Por exemplo, a seguinte instrução FORTRAN 90 declara NAME1 e NAME2 como sendo cadeias de caracteres de tamanho 15:

```
CHARACTER(LEN = 15) NAME1, NAME2
```

As cadeias de tamanho estático são sempre cheias; se uma cadeia mais curta for atribuída a uma variável, os caracteres vazios normalmente serão ajustados como brancos (espaços). A segunda opção é permitir que as cadeias tenham tamanhos variáveis até um máximo declarado e fixo estabelecido pela definição da variável, conforme é exemplificado pelas cadeias em C e em C++. Estas últimas, são chamadas de **cadeias de tamanho dinâmico limitado**. Tais variáveis podem armazenar qualquer número de caracteres entre zero e o máximo. Lembre-se de que as cadeias em C e em C++ usam um caractere especial para indicar o seu final, em vez de manterem o seu tamanho.

A terceira opção é permitir que as cadeias tenham tamanhos variáveis sem nenhum máximo, como no SNOBOL4, no JavaScript e na Perl. Essas são chamadas de **cadeias de tamanho dinâmico**. Tal opção exige a sobretaxa de alocação e de desalocação dinâmica de armazenamento, mas proporciona máxima flexibilidade.

### 6.3.4 Avaliação

Os tipos cadeia são importantes para a capacidade de escrita de uma linguagem. Tratá-las como vetores pode ser mais incômodo do que lidar com um tipo de cadeia primitivo. A sua adição como um tipo primitivo a uma linguagem não é custosa, tanto em termos de complexidade da linguagem como do compilador. Portanto, é difícil justificar a omissão de tipos de

cadeia primitivos em algumas linguagens contemporâneas. Logicamente, a disponibilidade de bibliotecas padrão de subprogramas de manipulação de cadeias pode superar essa deficiência quando estas não forem incluídas como um tipo primitivo.

Operações com cadeias, como casamento de padrões e concatenação simples, são fundamentais e devem ser incluídas para valores de tipo cadeia. Ainda que as cadeias de tamanho dinâmico evidentemente sejam as mais flexíveis, a sobretaxa de sua implementação deve ser pesada, em relação à flexibilidade adicional.

### 6.3.5 Implementação dos Tipos Cadeias de Caracteres

Os tipos cadeia de caracteres, às vezes, são suportados diretamente em hardware, mas, na maioria dos casos, usa-se software para implementar o armazenamento, a recuperação e a manipulação. Quando eles são representados como vetores de caracteres, freqüentemente a linguagem fornece poucas operações.

Um descritor para um tipo cadeia de caracteres estático, somente exigido durante a compilação, tem três campos. O primeiro campo de todo descritor é o nome do tipo. No caso das cadeias de caracteres estáticas, o segundo campo é o tamanho do tipo (em caracteres). O terceiro campo é o endereço do primeiro caractere. Esse descritor é mostrado na Figura 6.2. As cadeias dinâmicas limitadas exigem um descritor em tempo de execução para armazenar tanto o tamanho máximo fixo como o tamanho atual, como mostra a Figura 6.3. As cadeias de tamanho dinâmico exigem um descritor em tempo de execução mais simples, porque somente o tamanho atual precisa ser armazenado.

As cadeias dinâmicas limitadas do C e do C++ não exigem descritores em tempo de execução porque o final de uma cadeia é marcado com um caractere nulo. Elas não precisam do tamanho máximo porque os valores de índice nas referências de vetor não são verificados quanto à faixa nessas linguagens.

Cadeia estática
Tamanho
Endereço

**FIGURA 6.2** Descritor em tempo de compilação para cadeias estáticas.

As cadeias de tamanho estático e de tamanho dinâmico limitado não exigem nenhuma alocação especial de armazenamento. Nas de tamanho dinâmico limitado, suficiente armazenamento para o tamanho máximo é alocado quando a variável de cadeia é vinculada ao armazenamento, de modo que somente um único processo de alocação é envolvido. O tamanho máximo é fixado em tempo de compilação.

As cadeias de tamanho dinâmico exigem um gerenciamento de armazenagem mais complexo. O tamanho de uma cadeia, e, portanto, o armazenamento ao qual ela está vinculada, devem crescer e reduzir-se dinamicamente.

Há duas abordagens possíveis ao problema da alocação dinâmica. Primeiro, as cadeias podem ser armazenadas em uma lista encadeada, de modo que quando uma delas cresce, as células recém-adquiridas podem vir de qualquer lugar do monte. O inconveniente desse método é a grande quantidade de armazenamento ocupado pelos vínculos (*links*) na representação da lista. A alternativa é armazenar cadeias completas em células de armazenagem adjacentes. O problema ocorre quando uma cadeia cresce: como um armazenamento adjacente

Cadeia dinâmica limitada
Tamanho máximo
Tamanho atual
Endereço

**FIGURA 6.3** Descritor em tempo de execução para cadeias dinâmicas limitadas.

cente às células existentes pode continuar a ser alocado para a variável de cadeia? Frequentemente, este não está disponível. Ao contrário, uma nova área de memória é encontrada, a qual pode armazenar a nova cadeia completa, e a parte antiga é movida para esta área. Então, as células de memória usadas para a cadeia antiga são desalocadas.

Não obstante o método de lista encadeada requerer mais armazenamento, os processos de alocação e de desalocação associados são simples. Porém, algumas operações com cadeias tornam-se lentas em função da exigência de seguimento de ponteiros. Por outro lado, usar memória adjacente para cadeias de caracteres completas resulta em operações mais rápidas com elas e exige significativamente menos armazenamento. Porém, o processo de alocação é mais lento. O método da adjacência envolve o problema geral de alocação e da desalocação de segmentos de tamanho variável. Esse problema será discutido na Seção 6.10.10.3.

## 6.4 Tipos Ordinais Definidos pelo Usuário

Um **tipo ordinal** é aquele cuja faixa de valores possíveis pode ser facilmente associada ao conjunto dos números inteiros positivos. No Pascal e na Ada, por exemplo, os tipos ordinais primitivos são o inteiro, o caractere e o booleano. Em muitas linguagens, os usuários podem definir dois tipos de ordinais: enumerações e subfaixa.

### 6.4.1 Tipos Enumeração

Um **tipo enumeração** é aquele em que todos os valores possíveis, os quais se tornam constantes simbólicas (na Ada, eles também poderiam ser literais de caracteres), são enumerados na definição. Um tipo enumeração característico é mostrado no seguinte exemplo Ada:

```
type DIAS is (Seg, Ter, Quar, Qui, Sex, Sab, Dom);
```

A principal questão de projeto específica aos tipos enumeração é a seguinte: deve permitir-se que um literal constante apareça em mais de uma definição de tipo e, se assim for, como o tipo de uma ocorrência desse literal é verificado no programa?

#### 6.4.1.1 Projetos

No Pascal, não é permitido que um literal constante seja usado em mais de uma definição de tipo de enumeração em determinado ambiente de referenciamento. Variáveis enumeradas podem ser usadas como subscritos de matrizes, como variáveis de laço **for**, e como expressões seletoras **case**, mas não podem ser usados em instruções de entrada e saída. Duas variáveis de tipo enumeração e/ou literais do mesmo tipo podem ser comparadas com os operadores relacionais, com suas posições relativas na declaração que determina o resultado. Por exemplo, em

```
type tipocor = (vermelho, azul, verde, amarelo);
var cor : tipocor;
...
cor := azul;
if cor > vermelho ...
```

a expressão booleana do **if** será avaliada como verdadeira.

No ANSI C e no C++, bem como no Pascal, o mesmo literal constante não pode aparecer em mais de uma definição de tipo enumeração em determinado ambiente de referência. Os valores de enumeração no ANSI C e no C++ são implicitamente convertidos para números inteiros, de modo que estão sujeitos somente às regras de uso dos mesmos.

Os tipos enumeração da Ada são similares aos do Pascal, exceto a permissão de que os literais apareçam em mais de uma declaração no mesmo ambiente de referenciamento. Esses são chamados de **literais sobrecarregados**. A regra para resolver a sobrecarga — ou seja, decidir o tipo de ocorrência dessa literal — é que ele deve ser determinável a partir do contexto de seu aparecimento. Por exemplo, se um literal sobrecarregado e uma variável de enumeração forem comparados, o tipo da literal será resolvido para ser o da variável.

Em alguns casos, o programador deve indicar alguma especificação do tipo para uma ocorrência de um literal sobrecarregado. Suponhamos, por exemplo, que um programa tenha os dois tipos de enumeração seguintes:

```
type LETRAS is ('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
                 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
                 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
                 'Y', 'Z');
type VOGAIS is ('A', 'E', 'I', 'O', 'U');
```

Suponhamos, ainda, que o programa use um laço **for** cuja variável deve assumir os valores do VOGAIS, como em

```
for LETRA in 'A'.. 'U' loop
```

O problema é que o compilador não pode determinar o tipo correto para **LETRA**, de modo que a faixa discreta (nesse caso, '**A**' .. '**U**') é ambígua. (Em Ada, a variável **for** é implicitamente tipificada pelo compilador. Ela tem o tipo da faixa discreta especificada na instrução.) A solução é usar um qualificador de tipo nos literais da faixa discreta, como em

```
for LETRA IN vogais('A') .. VOGAIS('U') loop
```

Variáveis de enumeração podem constar em instruções de saída (**output**), e literais de enumeração podem ser introduzidas (**input**) usando-se o pacote **TEXT\_IO** da Ada. Essas operações exigem uma instanciação genérica de um pacote incorporado para o tipo de enumeração específico. As instanciações genéricas de pacotes serão discutidas no Capítulo 11.

Na Ada, tantos os tipos **BOOLEAN** como os **CHARACTER** são, de fato, tipos de enumeração predefinidos. As operações comuns para eles são com o antecessor, o sucessor, a posição na lista de valores e o valor para determinado número de posição. Em Pascal, essas operações são fornecidas por funções incorporadas. Por exemplo, **pred(azul)** é vermelho. Em Ada, elas são atributos. Por exemplo, **LETRA'PRED('B')** é '**A**'.

Em Java, os tipos enumeração são classes que implementam a interface **Enumeration**.

#### 6.4.1.2 Avaliação

Os tipos enumeração apresentam vantagens tanto em termos de legibilidade como de confiabilidade. A primeira é aumentada de uma maneira muito direta: valores nomeados são facilmente reconhecidos, enquanto que os valores codificados não o são. As codificações são inexpressivas para qualquer pessoa, exceto para o autor do programa. Por exemplo, suponhamos que um programa que esteja sendo escrito em FORTRAN exija que uma variável armazene 10 cores diferentes. Muito provavelmente, os nomes das cores seriam codifi-

cados como números inteiros, usando os valores 1, 2, ..., 10. Os valores inteiros usados para códigos raramente são conotativos. Se a constante 4, por exemplo, denotar azul e for atribuída a uma variável, esse fato não será aparente para o leitor do programa.

Na área da confiabilidade, os tipos enumeração apresentam duas vantagens. Primeiro, se uma variável de número inteiro for usada pelo programador para uma codificação que exija somente uma subfaixa muito pequena dos valores inteiros, erros de faixa poderiam ocorrer, mas não seriam detectados pelo sistema em tempo de execução, como, por exemplo, a cor 17. Em segundo lugar, qualquer uma das muitas operações aritméticas entre o dia codificado e qualquer número inteiro seria válida, porque seus tipos coincidiriam. Isso elimina a detecção de muitos erros lógicos e tipográficos por intermédio do compilador, envolvendo os dados codificados. (Uma vez que o ANSI C e o C++ tratam as variáveis de enumeração como variáveis inteiras, essas linguagens não oferecem esta vantagem.)

Assim, usar um tipo enumerado como o LETRAS da Ada definido acima, em vez de um número inteiro, restringe as variáveis atribuíveis a uma pequena faixa, é mais legível e oferece verificação de tipos.

## 6.4.2 Tipos Subfaixa

Um **tipo subfaixa** ( subrange) é uma subseqüência de um ordinal. Por exemplo, 12..14 é uma subfaixa do tipo inteiro. Os tipos subfaixa foram introduzidos pelo Pascal e também estão incluídos na Ada. Não existem questões de projeto específicas a eles.

### 6.4.2.1 Projetos

Em Pascal, as declarações do tipo subfaixa típicas são

```
type
  maiuscula = 'A'..'Z';
  indice = 1..100;
```

A conexão de um tipo subfaixa ao seu primitivo é estabelecida combinando-se os valores existentes na definição da subfaixa com os dos ordinais anteriormente declarados ou incorporados. No exemplo acima, o maiuscula é definido como uma subfaixa do tipo incorporado para caracteres únicos. O indice é definido como uma subfaixa dos números inteiros.

Na Ada, as subfaixas são incluídas na classe de tipos, chamada subtipos. Como foi declarado no Capítulo 5, os subtipos não são novos absolutamente, mas, ao contrário, são somente nomes novos para versões possivelmente restritas ou limitadas de tipos existentes. Por exemplo, supondo-se que DIAS seja definido como na Seção 6.4.1, poderíamos ter o seguinte:

```
subtype DIASEMANA is DIAS range Seg .. Sex;
subtype INDICE is INTEGER range 1..100;
```

Nesses exemplos, a restrição para os tipos existentes está na faixa de valores possíveis. Todas as operações definidas para o tipo-pai também são definidas para o subtipo, exceto a atribuição de valores fora da faixa especificada. Por exemplo, na seguinte,

```
DIA1 : DIAS;
```

```
DIA2 : DIASSEMANA;
...
DIA2 := DIA1;
```

a atribuição é válida, a menos que o valor de `DIA1` seja `Sab` ou `Dom`. Como na Ada, as subfaixa no Pascal herdam todas as operações do pai.

Os tipos ordinais definidos pelo usuário servem, mais comumente, para os índices de matrizes, como discutimos na Seção 6.5. Eles podem ser usados também para variáveis de laço. De fato, as subfaixas de tipos ordinais são a única maneira pela qual a faixa de variáveis de laço `for` Ada pode ser especificada.

Note que os tipos subfaixa são muito diferentes dos tipos derivados da Ada, discutidos no Capítulo 5. Por exemplo, considere as seguintes declarações de tipo:

```
type PEQUENO_INT_DERIVADO is new INTEGER range 1..100;
subtype PEQUENO_INT_SUBFAIXA is INTEGER range 1..100;
```

As variáveis de ambos os tipos, `PEQUENO_INT_DERIVADO` e `PEQUENO_INT_SUBFAIXA`, herdam a faixa de valores e as operações dos inteiros. Porém, as variáveis do tipo `PEQUENO_INT_DERIVADO` não são compatíveis com qualquer tipo `INTEGER`. Por outro lado, as variáveis do tipo `PEQUENO_INT_SUBFAIXA` são compatíveis com variáveis e com constantes do tipo `INTEGER` e com qualquer subtipo de `INTEGER`.

#### 6.4.2.2 Avaliação

Os tipos subfaixa aumentam a legibilidade ao esclarecer para os leitores que as variáveis de subtipos podem armazenar somente faixas de valores. A confiabilidade aumenta com eles porque a atribuição de um valor a uma variável de subfaixa fora da faixa especificada é detectada como um erro pelo compilador (no caso do valor atribuído ser um valor literal) ou pelo sistema em tempo de execução (no caso de uma variável ou de uma expressão).

#### 6.4.3 Implementação de Tipos Definidos pelo Usuário

Os tipos enumeração normalmente são implementados, associando um valor inteiro não-negativo a cada constante simbólica no tipo. Normalmente, o primeiro valor de enumeração é representado como 0, o segundo como 1 e assim por diante. Obviamente, as operações permitidas são drasticamente diferentes daquelas com números inteiros, exceto para os operadores relacionais, que são idênticas. Conforme afirmamos anteriormente, os tipos enumeração ANSI C e C++ freqüentemente são tratados exatamente como números inteiros.

Os tipos subfaixa são implementados exatamente da mesma maneira que seus tipos-pai, exceto que as verificações de faixas devem ser implicitamente incluídas pelo compilador em toda atribuição de uma variável ou de expressão a uma variável de subfaixa. Isso aumenta o tamanho do código e o tempo de execução, mas, usualmente, considera-se que vale o custo. Além disso, um bom compilador otimizador pode melhorar parte da verificação.

## 6.5 Tipos Matriz

Uma **matriz** é um agregado homogêneo de elementos de dados cujo elemento individual é identificado por sua posição no agregado em relação ao primeiro. Uma referência a um elemento de matriz em um programa freqüentemente inclui um ou mais subscritos não-constantes. Essas referências exigem um cálculo em tempo de execução para determinar a localização da memória referenciada. Os elementos de dados individuais de uma matriz são de algum tipo previamente definido, primitivo ou não. A maioria dos programas de computador precisa modelar coleções de valores do mesmo tipo e que devem ser processadas da mesma maneira. Assim, a necessidade universal das matrizes é evidente.

### 6.5.1 Questões de Projeto

As questões de projeto específicas às matrizes são as seguintes:

- Quais tipos são legais para os subscritos?
- As expressões de subscrito nas referências a elementos são verificadas quanto à faixa?
- Quando as faixas de subscrito são vinculadas?
- Quando a alocação da matriz se desenvolve?
- Quantos subscritos são permitidos?
- Matrizes podem ser inicializadas quando têm seu armazenamento alocado?
- Quais tipos de fatias são permitidos, se for o caso?

Nas seções seguintes, exemplos das opções de projeto tomadas para as matrizes da maioria das linguagens de programação mais comuns serão discutidos.

### 6.5.2 Matrizes e Índices

Elementos específicos de uma matriz são referenciados por meio de um mecanismo sintático de dois níveis, cuja primeira parte é o nome do agregado e a segunda é um seletor possivelmente dinâmico que consiste em um ou mais itens conhecidos como **subscritos** ou **índices**. Se todos os índices em uma referência forem constantes, o seletor será estático; caso contrário, será dinâmico. A operação de seleção pode ser imaginada como uma correspondência do nome da matriz e o conjunto de valores de índice com um elemento do agregado. De fato, as matrizes, às vezes, são chamadas de *mapeamento finito*. Simbolicamente, este mapeamento pode ser mostrado como

*nome\_da\_matriz* (*lista\_de\_valores\_indice*) → *elemento*

A sintaxe das referências a matrizes é mais ou menos universal: ao nome da matriz segue-se o da lista de índices, colocada entre parênteses ou entre colchetes. Um problema com os parênteses é que eles freqüentemente são usados para conter os parâmetros de chamadas a subprogramas; isso faz as referências a matrizes parecerem-se exatamente com essas chamadas. Por exemplo, considere a seguinte instrução de atribuição FORTRAN:

`SOMA = SOMA + B(I)`

Uma vez que parênteses são usados para parâmetros de subprograma e em subscritos de matriz no FORTRAN, tanto os leitores do programa como os compiladores são obrigados a

usar outras informações para determinar se  $B(1)$  dessa atribuição é uma chamada a função ou uma referência a um elemento de matriz. Isso pode ser frustrante para o leitor.

Os projetistas das linguagens FORTRAN anteriores à 90 e da PL/I escolheram os parênteses para subscritos de matriz porque nenhum outro caractere adequado estava disponível na época. Quando um identificador seguido de uma expressão ou de uma lista de expressões entre parênteses é encontrado em um programa FORTRAN, anterior à década de 90, ou em PL/I, o compilador determina se ele é uma referência a matriz ou uma chamada a função, comparando o nome com todas as matrizes declaradas no ambiente de referenciamento. Se nenhuma coincidência for encontrada, irá presumir-se que ele é uma chamada à função. Se não se concluir que é um subprograma localmente definido, será presumido que ele é definido externamente. Se a referência tiver sido a uma matriz cuja declaração está faltando, tal fato não poderá ser determinado pelo compilador, porque essas linguagens têm compilações separadas, e subprogramas usados em um programa, mas definidos em outro lugar, não precisam ser declarados como externos.

Na Ada, que usa parênteses para envolver parâmetros de subprograma e índices de matriz, o compilador sempre pode determinar se uma referência é à matriz ou a uma função, porque ele tem acesso às informações (de programas compilados anteriormente) sobre todos os nomes que podem ser referenciados em uma unidade de programa em compilação. Isso contrasta com os FORTRANs, anteriores à década de 90, e com a PL/I, nos quais o compilador não tem nenhum acesso às informações sobre programas anteriormente compilados.

Os projetistas da Ada, especificamente, escolheram os parênteses para envolver subscritos de modo que houvesse uniformidade entre referências a matrizes e chamadas a funções em expressões, apesar dos potenciais problemas de legibilidade. Eles fizeram tal opção baseando-se no fato de que tanto as referências a elementos de matriz como chamadas a funções são associações. As primeiras associam os subscritos a um elemento particular da matriz. As outras associam os parâmetros reais à definição de função e, por fim, a um valor funcional.

O Pascal, o C, o C++ e o Java usam colchetes para delimitar seus índices de matriz.

Dois tipos distintos estão envolvidos em um tipo de matriz: o do elemento e o dos subscritos. Este último freqüentemente é uma subfaixa de números inteiros, mas o Pascal e a Ada permitem que alguns outros tipos sejam usados como subscritos, como, por exemplo, booleano, caractere e enumeração.

As primeiras linguagens de programação não especificavam que as faixas de subscrito deveriam ser implicitamente verificadas; por isso, erros de faixas são comuns em programas. Assim, exigir sua verificação é um fator importante na confiabilidade das linguagens. Entre as contemporâneas, o C, o C++ e o FORTRAN não especificam a verificação de faixa de subscritos, mas o Pascal, a Ada e o Java especificam.

### 6.5.3 Vinculações de Subscrito e Categorias de Matrizes

A vinculação do tipo do subscrito a uma variável matriz normalmente é estática, mas as faixas de valor de subscrito, às vezes, são vinculadas dinamicamente.

Em algumas linguagens, o limite inferior da faixa de subscrito é implícito. Por exemplo, no C, no C++ e no Java, o limite inferior de todas as faixas de índice é fixado em zero; no FORTRAN I, II e IV, ele foi fixado em 1; no FORTRAN 77 e no FORTRAN 90, seu padrão é 1. Na maioria das outras linguagens, as faixas de subscrito devem ser especificadas completamente pelo programador.

Como acontece com as variáveis escalares, as matrizes ocorrem em quatro categorias. Nesse caso, as definições de categoria baseiam-se na vinculação às faixas de valor de subscrito e nas vinculações ao armazenamento. Novamente, os nomes de categoria indicam onde e quando este é alocado.

Uma **matriz estática** é aquela em que as faixas de subscrito estão estaticamente vinculadas e a alocação de armazenamento é estática (feita antes da execução). A vantagem das matrizes estáticas é a eficiência. Nenhuma alocação ou desalocação dinâmica é exigida.

Uma **matriz fixa dinâmica na pilha** (matriz fixa) é aquela em que as faixas de subscrito estão estaticamente vinculadas, mas a alocação é feita no momento de elaboração da declaração durante a execução. A sua vantagem sobre as matrizes estáticas é a eficiência de espaço. Uma matriz grande em um procedimento pode usar o mesmo espaço que uma grande em um procedimento diferente, contanto que ambos os procedimentos não estejam ativos ao mesmo tempo.

Uma **matriz dinâmica na pilha** é aquela em que as faixas de subscrito estão dinamicamente vinculadas e a alocação de armazenamento é dinâmica (feita durante a execução). Porém, assim que as faixas de subscrito são vinculadas e o armazenamento é alocado, eles permanecem fixos durante o tempo de vida da variável. A vantagem desta sobre as estáticas e sobre as fixas dinâmicas na pilha é a flexibilidade. O tamanho de uma matriz não precisa ser conhecido até que ela esteja prestes a ser usada.

Uma **matriz dinâmica no monte** é aquela em que a vinculação das faixas de subscrito e de alocação de armazenamento é dinâmica e pode mudar qualquer número de vezes durante o seu tempo de vida. A sua vantagem sobre as outras é a flexibilidade: as matrizes podem crescer e reduzir-se durante a execução do programa conforme mudar a necessidade de espaço. Exemplos das quatro categorias são dados nos parágrafos seguintes.

No FORTRAN 77, o tipo do subscrito é vinculado a uma matriz no tempo de projeto da linguagem; todos os subscritos são do tipo inteiro. As faixas de subscrito são estaticamente vinculadas e todo o armazenamento é alocado estaticamente; portanto, as matrizes do FORTRAN 77 são estáticas.

As matrizes que são declaradas em procedimentos Pascal e em funções C (sem o especificador **static**) são exemplos de matrizes fixas dinâmicas na pilha.

As matrizes Ada podem ser dinâmicas na pilha, como no exemplo seguinte:

```
GET(LIST_LEN);
declare
    LIST : array (1..LIST_LEN) of INTEGER;
begin
    ...
end;
```

Nesse exemplo, o usuário introduz o número de elementos desejados na matriz LIST, os quais são, então, dinamicamente alocados quando a execução atinge o bloco **declare**. Quando a execução atinge o final do bloco, a matriz LIST é desalocada.

O FORTRAN 90 oferece matrizes dinâmicas. Elas podem ser alocadas e desalocadas mediante solicitação. Suas faixas de subscrito podem ser mudadas por qualquer processo de salvar, de desalocar ou de reallocar.

Por exemplo, no FORTRAN 90, pode-se declarar uma matriz para que seja dinâmica, com

```
INTEGER, ALLOCATABLE, ARRAY (:,:) :: MAT
```

o qual declara que `MAT` é uma matriz de elementos do tipo `INTEIRO` que pode ser dinamicamente alocada e cuja especificação é feita com uma instrução `ALLOCATE`, como, por exemplo,

```
ALLOCATE (MAT(10, NUMERO_DE_COLS))
```

As faixas de subscrito podem ser especificadas por variáveis de programa, bem como por literais. Os limites inferiores das faixas de subscrito são assumidos como 1.

As matrizes dinâmicas podem ser destruídas pela instrução `DEALLOCATE`, como em

```
DEALLOCATE (MAT)
```

Para tornar uma matriz dinâmica maior ou menor, seus elementos devem ser salvos temporariamente em outra matriz, e ela deve ser desalocada e, depois, realocada no novo tamanho.

O C e o C++ também oferecem matrizes dinâmicas. As funções de biblioteca padrão, `malloc` e `free`, ambas operações de alocação e de desalocação gerais do monte, respectivamente, podem ser usadas em matrizes C. O C++ usa os operadores `new` e `delete` para gerenciar o armazenamento no monte. Uma vez que não há nenhuma verificação de faixa de índices no C e no C++, o tamanho de uma matriz não interessa ao sistema em tempo de execução; assim, estender ou encolher uma matriz é fácil. Elas são tratadas como ponteiros para algumas coleções de células de armazenagem em que o ponteiro pode ser indexado, conforme discutiremos na Seção 6.10.6.

A Perl e o JavaScript têm outro tipo de matriz dinâmica. Elas crescem implicitamente quando são feitas atribuições a elementos além do último atual. Pode-se fazer com que elas encolham ao atribuir-lhes um agregado vazio, especificado com `()`.

Na versão original do Pascal, a faixa ou as faixas de índice de uma matriz faziam parte de seu tipo. Isso, juntamente com o uso da equivalência de nomes para garantir a compatibilidade, desaprovava a existência de um subprograma que processava matrizes de tamanhos diferentes. Um procedimento que classificava uma matriz de números inteiros, por exemplo, somente podia ser escrito para aquelas com uma única faixa de subscrito fixa. O ISO Standard Pascal (ISO, 1982) oferece uma saída para o problema: as matrizes **conformantes**, parâmetros formais que incluem a definição de tipo da matriz. Considere o seguinte exemplo:

```
procedure somalista ( var soma : integer;
                      lista : array [inferior .. superior :
                                     integer] of integer);
var indice : integer;
begin
  soma := 0;
  for indice := inferior to superior do
    soma := soma + lista[indice]
  end;
```

Um exemplo de chamada para esse procedimento é

```
var escores : array [1..100] of integer;
...
somalista(soma, escores)
```

### 6.5.4 O Número de Subscritos em Matrizes

O FORTRAN I limitou a três o número de subscritos de uma matriz porque, na época da execução do projeto, a eficiência era uma preocupação fundamental. Os projetistas do FORTRAN I desenvolveram um método muito rápido para acessar elementos de matrizes de até três dimensões, mas não mais do que três. Do FORTRAN IV em diante, permitiu-se que o número de dimensões chegasse a sete, mas a maioria das outras linguagens contemporâneas não impõe esse limite. Não existe nenhuma justificativa para a limitação do FORTRAN. Um programador que deseja usar uma variável com 10 dimensões e esteja disposto a pagar pelo custo das referências aos elementos dessa matriz deve ter permissão para fazê-lo.

### 6.5.5 Inicialização de Matrizes

Algumas linguagens fornecem o meio de inicializar matrizes no momento em que o armazenamento é alocado. No FORTRAN 77, todo o armazenamento de dados é alocado estaticamente; assim, a inicialização no momento de execução, usando-se a instrução **DATA**, é permitida. Por exemplo, no FORTRAN 77 poderíamos ter

```
INTEGER LISTA(3)
DATA LISTA /0, 5, 5/
```

**LISTA** é inicializada para os valores da lista delimitados por barras diagonais.

O ANSI C, o C++ e o Java também permitem inicialização de suas matrizes, mas com uma nova mudança: na declaração

```
int lista [] = {4, 5, 7, 83};
```

o compilador define o tamanho do vetor. Isso pretende ser uma conveniência, mas tem seu custo. Ela efetivamente remove a possibilidade do sistema poder detectar alguns erros do programador, como, por exemplo, deixar erroneamente um valor fora da lista.

As cadeias de caracteres no C e no C++ são implementadas como vetores de **char**. Estes, por sua vez, podem ser inicializados para constantes de cadeia, como em

```
char nome [] = "freddie";
```

O vetor **nome** terá oito elementos, porque todas as cadeias são encerradas com um caractere nulo (zero), o qual é implicitamente suportado pelo sistema para constantes de cadeia.

Vetores de cadeias no C e no C++ também podem ser inicializados com literais de cadeia. Nesse caso, o vetor é de ponteiros para caracteres. Por exemplo,

```
char *nomes [] = {"Bob", "Jake", "Darcie"};
```

ilustra a natureza dos literais de caracteres em C e em C++. No exemplo anterior de literal de cadeia usada para inicializar o vetor **char nome**, a literal é considerada um vetor de **char**. Mas, no último exemplo (**nomes**), as literais são consideradas ponteiros para os caracteres, de modo que o vetor é de ponteiros para caracteres. Por exemplo, **nomes[0]** é um ponteiro para a letra 'B' no vetor de caracteres literais que contém 'B', 'o', 'b', e nulo.

O Pascal e o Modula-2 não permitem inicialização de matrizes nas seções de declaração do programa.

A Ada fornece dois mecanismos para inicializar matrizes na instrução de declaração: listando os valores na ordem em que são armazenados, ou atribuindo-os diretamente a uma

posição do índice usando o operador `=>`, que é chamado de seta na Ada. Por exemplo, considere o seguinte:

```
LISTA : array (1..5) of INTEGER := (1, 3, 5, 7, 9);
GRUPO : array (1..5) of INTEGER := (1 => 3, 3 => 4,
                                         others => 0);
```

Na primeira instrução, todos os elementos de `LISTA` têm valores de inicialização, os quais são atribuídos às localizações de cada elemento do vetor na ordem em que eles aparecem. Na segunda, o primeiro e o terceiro elementos deste são inicializados, usando-se a atribuição direta e a cláusula `others` é usada para inicializar os elementos restantes. Essas coleções de valores, definidas pelos parênteses, são chamadas de **valores agregados**.

### 6.5.6 Operações com Matrizes

Uma operação de matriz é aquela em que ela opera como uma unidade. Algumas linguagens, como o FORTRAN 77, não oferecem qualquer operação desse tipo.

A Ada permite atribuições com matrizes, inclusive aquelas em que o lado direito é um valor agregado em vez de um nome de matriz. Ela também oferece concatenação, especificada pelo E comercial (`&`), definida entre dois vetores e entre um vetor e um escalar. Quase todos os tipos da Ada têm os operadores relacionais incorporados para igualdade e desigualdade.

O FORTRAN 90 inclui um grande número de operações de matrizes chamados **elementares** porque são operações entre pares de elementos de matriz. Por exemplo, o operador de adição (+) entre dois vetores resulta em um vetor das somas dos pares de elementos dos dois vetores. Os operadores de atribuição, aritméticos, relacionais e lógicos são sobre-carregados para matrizes de qualquer tamanho ou forma. O FORTRAN 90 também inclui funções intrínsecas ou de biblioteca para multiplicação de matrizes, para transposição de matrizes e para produto escalar de vetores.

As matrizes e suas operações são o coração da APL; ela é a linguagem de processamento de matrizes mais poderosa já idealizada. Devido à sua relativa obscuridade e à falta de influência nas linguagens subsequentes, apresentamos aqui somente uma pequena parte de suas operações com matrizes.

Na APL, as quatro operações aritméticas básicas são definidas para vetores (matrizes unidimensionais) e matrizes, bem como para operandos escalares. Por exemplo,

`A + B`

é uma expressão válida, quer `A` e `B` sejam variáveis escalares, vetores ou matrizes.

A APL inclui um conjunto de operadores unários para vetores e para matrizes, alguns dos quais são os seguintes (em que `v` é um vetor e `M` é uma matriz):

- $\phi v$  inverte os elementos de `v`
- $\phi M$  inverte as colunas de `M`
- $\theta M$  inverte as linhas de `M`
- $\Omega M$  transpõe `M` (suas linhas tornam-se suas colunas e vice-versa)
- $\boxplus M$  inverte `M`

Elas também inclui diversos operadores especiais que tomam outros operadores como operandos. Um deles é o de produto interno, especificado com um ponto final (.). Ele assume dois operandos, operadores binários. Por exemplo,

$+ . \times$

é um novo operador que toma dois argumentos, vetores ou matrizes. Primeiro, ele multiplica os elementos correspondentes dos dois argumentos e depois soma os resultados. Por exemplo, se  $A$  e  $B$  forem vetores,

$A + . \times B$

será o produto matemático interno de  $A$  e  $B$  (um vetor de produtos dos elementos correspondentes de  $A$  e  $B$ ). A instrução

$A + . \times B$

é a soma dos produtos internos de  $A$  e  $B$ . Se  $A$  e  $B$  forem matrizes, essa expressão especificará a multiplicação matricial de  $A$  e  $B$ .

Os operadores especiais da APL são, de fato, formas funcionais, que serão descritas no Capítulo 15.

### 6.5.7 Fatiás

Uma **fatiá** (*slice*) de uma matriz é alguma subestrutura desta. Por exemplo, se  $A$  for uma matriz, a primeira linha de  $A$  será uma fatia possível, o mesmo acontecendo com a última linha e com a primeira coluna. É importante perceber que uma fatia não é um novo tipo de dados. Ao contrário, é um mecanismo para referenciar parte de uma matriz como uma unidade. Se as matrizes não puderem ser manipuladas como unidades, a linguagem não terá nenhum uso para as fatias.

Uma das questões de projeto relacionadas com as fatias é a sintaxe para especificar uma referência a uma fatia particular. Uma referência a um elemento particular de uma matriz completa é o nome desta e uma expressão para cada subscrito. Uma vez que uma fatia é uma subestrutura de uma matriz, uma referência a uma delas requer um número menor de expressões de subscrito do que uma referência à matriz inteira. De alguma maneira, entretanto, as expressões de subscrito ausentes devem ser denotadas, de tal forma que as expressões presentes sejam associadas com os subscritos corretos. O subscrito ou os subscritos de referência a fatias que faltam, às vezes, são especificados por meio de asteriscos. Por exemplo, considere as seguintes declarações FORTRAN 90:

```
INTEGER VET(1:10) , MAT(1:3, 1:3), CUBO(1:3, 1:3, 1:4)
```

`VET(3:6)` é um vetor de quatro elementos que contém do terceiro ao sexto elementos de `VET`; `MAT(1:3, 2)` refere-se à segunda coluna de `MAT`; `MAT(3, 1:3)` refere-se à terceira linha de `MAT`. Todas essas referências podem ser usadas como matrizes unidimensionais. As referências a todas as fatias destas são tratadas como se fossem matrizes da dimensionalidade restante. Assim, uma referência a fatia como `CUBO(1:3, 1:3, 2)` poderia ser legalmente atribuída à `MAT`. Elas também podem aparecer como destinos das

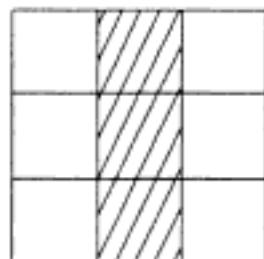
instruções de atribuição. Por exemplo, um vetor poderia ser atribuído a uma fatia de uma matriz. A Figura 6.4 mostra diversas fatias de MAT e CUBO.

Fatias mais complexas também podem ser especificadas no FORTRAN 90. Por exemplo, `VET(2:10:2)` é um vetor de cinco elementos que consiste no segundo, no quarto, no sexto, no oitavo e no décimo elementos de `VET`. Fatias também podem ter arranjos não-regulares de elementos de uma matriz existente. Por exemplo, `VET((/3, 2, 1, 8/))` é um vetor do terceiro, do segundo, do primeiro e do oitavo elementos de `VET`.

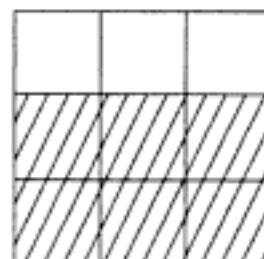
Na Ada, somente fatias altamente restritas são permitidas: aquelas que se compõem de elementos consecutivos de um único vetor. Por exemplo, se `LISTA` for um vetor com faixa de índice `(1..100)`, `LISTA(5..10)` será uma fatia de `LISTA` que consiste nos seis elementos indexados de 5 a 10. Conforme discutimos na Seção 6.3.2, uma fatia de um tipo `CADEIA` é chamada de referência a subcadeia.

### 6.5.8 Avaliação

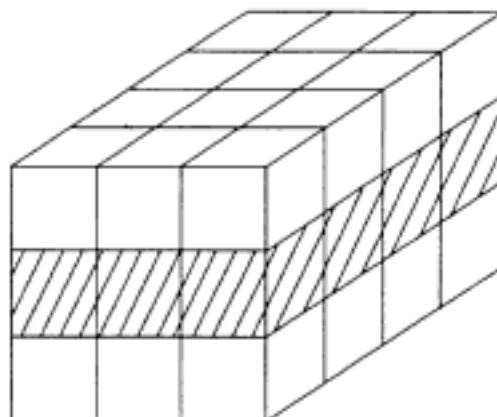
As matrizes têm sido incluídas em praticamente todas as linguagens. Elas são simples e foram bem desenvolvidas. O único avanço significativo desde sua introdução no FORTRAN I foi a inclusão de todos os tipos ordinais como tipos de subscrito possíveis e, é claro, as



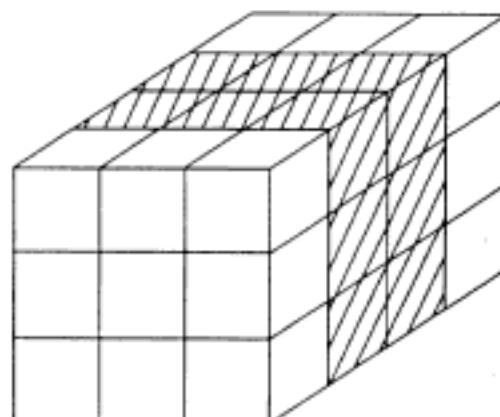
MAT (1:3, 2)



MAT (2:3, 1:3)



CUBO (2, 1:3, 1:4)



CUBO (1:3, 1:3, 2:3)

**FIGURA 6.4** Exemplos de fatias em FORTRAN 90.

matrizes dinâmicas. Existe pouca controvérsia no projeto das matrizes ainda que elas sejam essenciais e fundamentais.

Mesmo que, às vezes, seja conveniente permitir que o programador especifique uma subestrutura particular de uma matriz multidimensional, o aumento da dificuldade de implementação e de legibilidade não valem, necessariamente, a conveniência.

### 6.5.9 Implementação do Tipo Matriz

Implementar matrizes exige mais esforço em tempo de compilação do que implementar tipos simples, como, por exemplo, o inteiro. O código que permite acessar elementos de uma matriz deve ser gerado no tempo de compilação. Em tempo de execução, este código deve ser executado para produzir endereços de elementos. Acessos a elementos de matrizes, especialmente naquelas com diversos subscritos, são mais dispendiosos do que o necessário se o código de acesso não for cuidadosamente projetado. Isso é verdadeiro, independentemente da matriz estar estática ou dinamicamente vinculada à memória. Não existe qualquer maneira de computar previamente o endereço a ser acessado por uma referência, como em

```
lista[k]
```

Uma matriz unidimensional é uma lista de células de memória adjacentes. Suponhamos que `lista` seja definida para ter um limite inferior de faixa de subscrito igual a 1. A função de acesso para `lista` freqüentemente terá a forma

$$\text{endereço}(\text{lista}[k]) = \text{endereço}(\text{lista}[1]) + (k-1) * \text{tamanho\_do\_elemento}$$

Isso é simplificado para

$$\text{endereço}(\text{lista}[k]) = (\text{endereço}(\text{lista}[1]) - \text{tamanho\_do\_elemento}) + (k * \text{tamanho\_do\_elemento})$$

em que o primeiro operando da adição é a parte constante da função de acesso, enquanto o segundo é a parte variável.

Se o tipo do elemento estiver estaticamente vinculado, e o vetor estiver estaticamente vinculado ao armazenamento, o valor da parte constante poderá ser computado antes da execução. Somente as operações de adição e de multiplicação continuam a ser feitas em tempo de execução. Se o endereço-base ou inicial do vetor não for conhecido até a execução, a subtração deverá ser feita quando aquele for alocado.

A generalização dessa função de acesso para um limite inferior arbitrário é

$$\text{endereço}(\text{lista}[k]) = \text{endereço}(\text{lista}[\text{limite\_inferior}]) + ((k - \text{limite\_inferior}) * \text{tamanho\_do\_elemento})$$

O descritor em tempo de compilação para matrizes unidimensionais pode ter a forma mostrada na Figura 6.5 (ver p. 234). Ele inclui informações necessárias para construir a função de acesso. Se a verificação de faixas de índice em tempo de execução não for feita e os atributos forem todos estáticos, somente a função de acesso será exigida durante a execução; nenhum descritor será necessário. Se a verificação de faixas de índice em tempo de execução for feita, estas talvez precisem ser armazenadas em um descritor em tempo de execução. Se as faixas de subscrito de um tipo de matriz particular forem estáticas, elas poderão ser incorporadas ao código que faz a verificação, eliminando a necessidade daquele descritor. Se qualquer uma das entradas do descritor estiverem dinamicamente vinculadas, elas devem ser mantidas em tempo de execução.

Vetor
Tipo de elemento
Tipo de índice
Limite inferior do índice
Limite superior do índice
Endereço

**FIGURA 6.5** Descritor em tempo de compilação para matrizes unidimensionais.

As matrizes multidimensionais são mais complexas para implementar do que as unidimensionais, ainda que a extensão para um número maior de dimensões seja razoavelmente direta. A memória de hardware é linear — usualmente, é uma seqüência de bytes. Assim, valores de tipos de dados com duas ou mais dimensões devem estar associados à memória unidimensional. Há duas maneiras comuns pelas quais as matrizes multidimensionais podem ser associadas a uma dimensão: a ordem de linha maior e a ordem de coluna maior. Na **ordem de linha maior**, os elementos da matriz que têm como seu primeiro subscrito o valor do limite inferior deste último são armazenados na frente, seguidos dos elementos do segundo valor do primeiro subscrito e assim por diante. No caso de uma matriz, ela será armazenada por linhas. Por exemplo, se a matriz tivesse os valores

3	4	7
6	2	5
1	3	8

ela seria armazenada em ordem de linha maior como

3, 4, 7, 6, 2, 5, 1, 3, 8

Na **ordem de coluna maior**, os elementos de uma matriz cujo último subscrito é o valor do limite inferior deste são armazenados primeiro, seguidos dos elementos do segundo valor do último e assim por diante. No caso de uma matriz, ela será armazenada por colunas. Se a matriz do exemplo acima fosse armazenada em ordem de coluna maior, ela teria a seguinte ordem na memória:

3, 6, 1, 4, 2, 3, 7, 5, 8

A ordem de coluna maior é usada no FORTRAN, mas as outras linguagens usam a de linha maior.

Às vezes, é fundamental saber a ordem de armazenamento de matrizes multidimensionais; por exemplo, quando estas são processadas usando ponteiros em programas C, e quando uma matriz em uma instrução EQUIVALENCE corresponde a outra que possui uma forma diferente em um programa FORTRAN. Em todos os casos, o acesso sequencial a elementos de matriz será mais rápido se eles forem feitos na ordem em que estão armazenados, porque isso minimizará a paginação. (Paginação é o movimento de blocos de informação entre o disco e a memória principal.)

A função de acesso para uma matriz multidimensional é a correspondência de seu endereço-base e um conjunto de valores de índice com o endereço na memória do elemento especificado pelos valores de índice. A função de acesso para matrizes bidimensionais armazenadas em ordem de linha maior pode ser desenvolvida conforme expomos a seguir. Em geral, o endereço de um elemento é o endereço-base da estrutura somado ao tamanho do elemento, multiplicado pelo número de elementos que o precedem na estrutura. Para uma matriz em ordem de linha maior, o número de elementos que precedem um elemento é o número de linhas acima do elemento multiplicado pelo tamanho de uma linha, somado ao número de elementos à sua esquerda. Isso é ilustrado na Figura 6.6, na qual fazemos a suposição simplificadora de que os limites inferiores de subscrito são todos 1.

Para obter um valor de endereço real, o número de elementos que precedem aquele desejado deve ser multiplicado pelo tamanho dele. Agora, a função de acesso pode ser escrita como

$$\begin{aligned} \text{localização}(a[i, j]) = & \text{endereço de } a[1, 1] + \\ & (((\text{número de linhas acima da } i\text{-ésima linha}) * (\text{tamanho de uma linha})) + (\text{número de elementos à esquerda da } j\text{-ésima coluna}) * \text{tamanho do elemento}) \end{aligned}$$

Uma vez que o número de linhas acima da  $i$ -ésima linha é  $(i - 1)$  e o número de elementos à esquerda da  $j$ -ésima coluna é  $(j - 1)$ , temos

$$\begin{aligned} \text{localização}(a[i, j]) = & \text{endereço de } a[1, 1] + (((i - 1) * n) + (j - 1)) \\ & * \text{tamanho\_do\_elemento} \end{aligned}$$

em que  $n$  é o número de elementos por linha. Isso pode ser reorganizado para a forma

$$\begin{aligned} \text{localização}(a[i, j]) = & \text{endereço de } a[1, 1] - ((n + 1) * \text{tamanho\_do\_elemento}) \\ & + ((i * n + j) * \text{tamanho\_do\_elemento}) \end{aligned}$$

em que os dois primeiros termos são a parte constante, e o último é a variável.

A generalização para limites inferiores arbitrários resulta na seguinte função de acesso:

$$\begin{aligned} \text{localização}(a[i, j]) = & \text{endereço de } a[\text{linha\_lb}, \text{col\_lb}] + (((i - \text{linha\_lb}) \\ & * n) + (j - \text{col\_lb})) * \text{tamanho\_do\_elemento} \end{aligned}$$

	1	2	$\cdots$	$j-1$	$j$	$\cdots$	$n$
1							
2							
:							
$i-1$							
$i$					$\otimes$		
:							
$m$							

FIGURA 6.6 A localização do elemento  $[i, j]$  em uma matriz.

em que `linha_lb` é o limite inferior das linhas, e `col_lb` é o inferior das colunas. Isso pode ser reorganizado para a forma

```
localização(a[i, j]) = endereço de a[linha_lb, col_lb] +
    (((linha_lb * n) + col_lb) * tamanho_do_elemento)
    + (((i * n) + j) * tamanho_do_elemento)
```

em que os dois primeiros termos são a parte constante, e o último é a variável. Isso pode ser generalizado para um número arbitrário de dimensões com relativa facilidade.

Para cada dimensão de uma matriz, uma instrução de adição e uma de multiplicação são necessárias para a função de acesso. Portanto, os acessos a elementos de matrizes com diversos subscritos são custosos. O descritor em tempo de compilação para uma matriz multidimensional é mostrado na Figura 6.7.

As fatias adicionam outra camada de complexidade às funções de associação de armazenamento. Para ilustrar isso, considere um programa no qual haja uma matriz e um vetor, e uma coluna da matriz seja atribuída a este, como em

```
INTEGER MAT (1:10, 1:5), LISTA (1:10)
...
LISTA = MAT (1:3, 3)
```

A função de associação de armazenamento para a matriz `MAT`, supondo-se a ordem de linha maior e um tamanho de elemento igual a 1, é

```
localização(MAT[i, j]) = endereço de MAT[1, 1] + ((i - 1) * 5 +
    (j - 1)) * 1 = (endereço de MAT[1, 1] - 6) + ((5 * i) + j)
```

A função de associação de armazenamento para a referência à fatia `MAT[1 : 3, 3]` é

```
localização(MAT[i, 3]) = endereço de MAT[1, 1] + ((i - 1) * 5 +
    (3 - 1)) * 1 = endereço de MAT[1, 1] - 3 + (5 * i)
```

Note que essa associação tem exatamente a mesma forma que qualquer outra função de acesso a vetor, ainda que a forma da parte constante seja diferente porque a matriz base é bidimensional.

Matriz multidimensional
Tipo de elemento
Tipo de índice
Número de dimensões
Faixa de índice 1
:
Faixa de índice $n$
Endereço

**FIGURA 6.7** Um descritor em tempo de compilação para uma matriz multidimensional.

Os elementos de **MAT** a ser atribuídos a **LISTA** são encontrados admitindo-se que **i** assuma os valores na faixa de subscrito da primeira dimensão de **MAT**.

## 6.6 Matrizes Associativas

Uma matriz associativa é um conjunto não-ordenado de elementos indexados por um número igual de valores chamados chaves. No caso de matrizes não-associativas, os índices nunca precisam ser armazenados (devido a sua regularidade). Nas associativas, entretanto, as chaves definidas pelo usuário devem ser armazenadas na estrutura. Assim, cada elemento de uma matriz associativa é, de fato, um par de entidades: uma chave e um valor. Usamos o modelo das matrizes associativas da Perl para ilustrar essa estrutura de dados. Eles também são suportados pela biblioteca de classes padrão do Java.

As questões de projeto que são específicas para as matrizes associativas são

- Qual é a forma de referência aos elementos?
- O tamanho de uma matriz associativa é estático ou dinâmico?

### 6.6.1 Estruturas e Operações

Na Perl, as matrizes associativas, muitas vezes, são chamados *hashes*, porque, na implementação, seus elementos são armazenados e recuperados com funções de *hash*. O espaço de nome para elas é distinto na Perl; cada variável de *hash* deve iniciar-se com um símbolo de porcentagem (%). Elas também podem ser ajustadas com valores de literais com a instrução de atribuição, como em

```
%salarios = ("Cedric" => 75000, "Perry" => 57000,
              "Mary" => 55750, "Gary" => 47850);
```

Os valores de elementos individuais são referenciados usando-se uma notação singular à Perl. O valor-chave é colocado entre chaves, e o nome do *hash* é substituído por um nome de variável escalar que é o mesmo, exceto para o primeiro caractere. Os nomes de variáveis escalares iniciam-se com um cifrão (\$). Por exemplo,

```
$salarios{"Perry"} = 58850;
```

Um novo elemento é adicionado usando-se a mesma forma de instrução. Um elemento pode ser removido do *hash* com o operador *delete*, como em

```
delete $salarios{"Gary"};
```

O *hash* inteiro pode ser esvaziado atribuindo-se a literal vazia a ele, como em

```
%salarios = ();
```

O tamanho de um *hash* Perl é dinâmico: ele cresce quando um novo elemento é adicionado e reduz-se quando um é excluído, e também quando ele é esvaziado pela atribuição da literal vazia. O operador *exists* retorna verdadeiro ou falso, dependendo se o seu operando chave é um elemento do *hash*. Por exemplo,

```
if (exists $salarios{"Shelly"}) ...
```

O operador `keys`, quando aplicado a um `hash`, retorna um vetor das chaves do `hash`. O operador `values` faz a mesma coisa em relação aos valores deste. O operador `each` itera sobre os pares de elementos de um `hash`.

Um `hash` é muito melhor do que uma matriz se forem necessárias buscas dos elementos, porque a operação de *hashing* implícita, usada para acessar o elemento do `hash`, é muito eficiente. Por outro lado, se todo elemento de uma lista precisar ser processado, seria mais eficiente usar uma matriz.

### 6.6.2 Implementando Matrizes Associativas

A implementação das matrizes associativas Perl é otimizada para prover pesquisas rápidas, mas quando o crescimento da matriz demanda reorganização, esta pode ser feita rapidamente. Um valor `hash` de 32 bits é calculado para cada entrada e armazenado junto com ela, embora inicialmente a matriz associativa use uma pequena parte do valor. Quando a matriz precisa ser expandida além de seu tamanho inicial, a função `hash` não precisa ser mudada, apenas são usados mais bits do valor `hash`. Somente a metade das entradas precisa ser movida, quando isso acontece. Assim, embora a expansão de uma matriz associativa não seja gratuita, ela não é tão custosa quanto o que você pode ter pensado.

## 6.7 Tipos Registro

Um **registro** é um agregado possivelmente heterogêneo de elementos de dados. Cada elemento individual é identificado por seu nome.

Freqüentemente, há a necessidade dos programas modelarem conjuntos de dados que não são homogêneos. Por exemplo, informações a respeito de um estudante colegial poderiam incluir o nome, o número do estudante, a média das notas e assim por diante. Um tipo de dados para esse conjunto poderia usar uma cadeia de caracteres para o nome, um número inteiro para o número do estudante, um número real para a média das notas, etc. Os registros são projetados para atender esse tipo de necessidade.

Os registros fizeram parte de todas as linguagens de programação mais populares, exceto as versões do FORTRAN anteriores à 90, desde o início da década de 60, quando foram introduzidos pelo COBOL.

Nas linguagens orientadas a objeto, a construção de classes suporta registros. O C++ ainda inclui o `struct` do C para estruturas de registros, não obstante ser redundante. O Java não tem `struct`.

As seções seguintes descrevem como os registros são declarados ou definidos, como as referências a campos dentro de registros são feitas e operações comuns nos registros.

As questões de projeto específicas aos registros são:

- Qual é a forma sintática das referências a campos?
- São permitidas referências elípticas?

### 6.7.1 Definições de Registros

A diferença fundamental entre um registro e uma matriz é a homogeneidade dos elementos nas matrizes contra a possível heterogeneidade dos elementos nos registros. Um resultado dessa diferença é que os elementos do registro, ou os campos, normalmente não são referenciados por índices. Ao contrário, os campos são nomeados com identificadores, e as referências a eles são feitas usando esses identificadores. Uma diferença mais importante entre as matrizes e os registros é que estes, em algumas linguagens, têm permissão para incluir uniões, a serem discutidas no Seção 6.8.

A forma COBOL de uma declaração de registro, que faz parte da divisão de dados deste programa, é ilustrada no seguinte exemplo:

```

01 REGISTRO-EMPREGADO.
02 NOME-EMPREGADO.
    05 PRIMEIRO          PICTURE IS X(20).
    05 MEIO              PICTURE IS X(10).
    05 ULTIMO            PICTURE IS X(20).
02 TAXA-HORARIA        PICTURE IS 99V99.

```

O registro REGISTRO-EMPREGADO consiste no registro NOME-EMPREGADO e no campo TAXA-HORARIA. Os numerais 01, 02 e 05 que iniciam as linhas da declaração de registro são **números de nível**, e indicam, por meio de seus valores relativos, a estrutura hierárquica dele. Qualquer linha que seja seguida por outra com um número de nível mais alto é, ela própria, um registro. As cláusulas PICTURE mostram os formatos das localizações de armazenagem do campo, com X(20) especificando 20 caracteres alfanuméricos, e 99V99 especificando quatro dígitos decimais com a vírgula decimal no meio.

O Pascal e a Ada usam uma sintaxe diferente para os registros; em vez de usar os números de nível do COBOL, eles indicam as estruturas de registro em uma forma ortogonal, simplesmente aninhando as declarações de registro. Considere a seguinte declaração Ada:

```

REGISTRO_EMPREGADO :
  record
    NOME_EMPREGADO :
      record
        PRIMEIRO : STRING (1..20);
        MEIO : STRING (1..10);
        ULTIMO : STRING (1..20);
      end record;
    TAXA_HORARIA : FLOAT;
  end record;

```

O C e o C++ também oferecem registros, chamados de estruturas. Eles são muito semelhantes aos do Pascal, exceto que não incluem as variantes de registro ou de uniões do Pascal, que serão discutidas na Seção 6.8.

As declarações de registro FORTRAN 90 exigem que quaisquer registros aninhados sejam previamente definidos como tipos. Assim, para o registro de empregados acima, o nome do empregado precisaria ser definido primeiro, e, depois, o registro do empregado simplesmente iria nomeá-lo como um tipo de seu primeiro campo.

### 6.7.2 Referências a Campos do Registro

As referências aos campos individuais de registros são sintaticamente especificadas por meio de diversos métodos diferentes, dois dos quais nomeiam o campo desejado e seus registros envolventes. As referências a campos do COBOL têm a forma

```
nome_do_campo OF nome_do_registro_1 OF ... OF nome_do_registro_n
```

em que o primeiro registro nomeado é o menor e o mais interno que contém o campo. O nome do registro seguinte da sequência é o do registro que contém o registro anterior e assim por diante. Por exemplo, o campo MEIO no exemplo de registro COBOL dado, pode ser referenciado com

```
MEIO OF NOME-EMPREGADO OF REGISTRO-EMPREGADO
```

A maioria das outras linguagens usa uma notação de pontos para as referências a campos, cujos componentes da referência são conectados com pontos. Os nomes, na notação de pontos, têm a ordem oposta das referências COBOL: eles usam o nome do maior registro envolvente primeiro e o nome do campo por último. Por exemplo, apresentamos a seguir uma referência ao campo MEIO no exemplo de registro Ada acima:

```
REGISTRO_EMPREGADO.NOME_EMPREGADO.MEIO
```

As referências a campos do FORTRAN 90 têm essa forma, exceto que se usa os símbolos de porcentagem (%) em vez de pontos.

Uma **referência amplamente qualificada** a um campo de registro é aquela em que todos os nomes de registro intermediários, do maior registro envolvente ao campo específico, são nomeados na referência. Tanto esta referência a campos COBOL como a Ada apresentadas acima são amplamente qualificadas. Como uma alternativa a estas referências, o COBOL e a PL/I permitem **referências elípticas** a campos de registro. Nestas, o campo é nomeado, mas qualquer um ou todos os nomes de registro envolventes podem ser omitidos, contanto que a referência resultante não seja ambígua no ambiente de referência. Por exemplo, PRIMEIRO, PRIMEIRO OF NOME\_EMPREGADO e PRIMEIRO OF REGISTRO\_EMPREGADO são referências elípticas ao primeiro nome do empregado no registro COBOL declarado acima. Elas exigem que o compilador tenha estruturas de dados e procedimentos elaborados para identificar corretamente o campo referenciado, embora sejam uma conveniência para o programador. Também são bastante prejudiciais para a legibilidade.

O Pascal permite um tipo de referência elíptica dentro de estruturas específicas. Um segmento de código pode ser colocado em uma cláusula **with**, em que se especifica uma parte da qualificação como implícita. Por exemplo, considere os dois segmentos de código seguintes, nos quais o primeiro é escrito sem uma cláusula **with** e o segundo usa-a.

```
empregado.nome := 'Bob';
empregado.idade := 42;
empregado.sexo := 'M';
empregado.salario := 23750.0;

with empregado do
  begin
    nome := 'Bob';
    idade := 42;
    sexo := 'M';
```

```

    salario := 23750,0
    end; { fim de with }

```

Ainda que uma cláusula **with** relativamente pequena seja um auxílio para a legibilidade, uma que ocupe diversas páginas de código pode prejudicá-la.

### 6.7.3 Operações em Registros

A atribuição é uma operação de registro comum. Na maioria dos casos, os tipos dos dois lados devem ser idênticos. A Ada permite comparações de registro para igualdade e para desigualdade. Além disso, os registros Ada podem ser inicializados com literais agregados.

O COBOL oferece a instrução **MOVE CORRESPONDING** para mover registros. Ela copia um campo do registro-fonte especificado para o de destino somente se este tiver um campo com o mesmo nome. Essa, freqüentemente, é uma operação útil nas aplicações de processamento de dados, cujos registros de entrada são transferidos para arquivos de saída depois de algumas modificações. Uma vez que os registros de entrada freqüentemente têm muitos campos com os mesmos nomes e propósitos que os registros de saída, mas não necessariamente na mesma ordem, a operação **MOVE CORRESPONDING** pode economizar muitas instruções. Por exemplo, considere as seguintes estruturas COBOL:

```

01 REGISTRO-ENTRADA.
  02 NOME.
    05 ULTIMO      PICTURE IS X(20).
    05 MEIO        PICTURE IS X(15).
    05 PRIMEIRO    PICTURE IS X(20).
  02 NUMERO-EMPREGADO PICTURE IS 9(10).
  02 HORAS-TRABALHADAS PICTURE IS 99.

01 REGISTRO-SAIDA.
  02 NOME.
    05 PRIMEIRO    PICTURE IS X(20).
    05 MEIO        PICTURE IS X(15).
    05 ULTIMO      PICTURE IS X(20).
  02 NUMERO-EMPREGADO PICTURE IS 9(10).
  02 PGMTO-BRUTO   PICTURE IS 999V99.
  02 PGMTO-LIQUIDO PICTURE IS 999V99.

```

A instrução

```
MOVE CORRESPONDING REGISTRO-ENTRADA TO REGISTRO-SAIDA.
```

copia os campos PRIMEIRO, MEIO, ULTIMO e NUMERO-EMPREGADO do registro de entrada para o registro de saída.

### 6.7.4 Avaliação

Os registros, freqüentemente, são tipos de dados valiosos nas linguagens de programação. O projeto de tipos de registro é direto e, seu uso, seguro. O único aspecto dos registros que não é claramente legível são as referências elípticas permitidas pelo COBOL e pela PL/I.

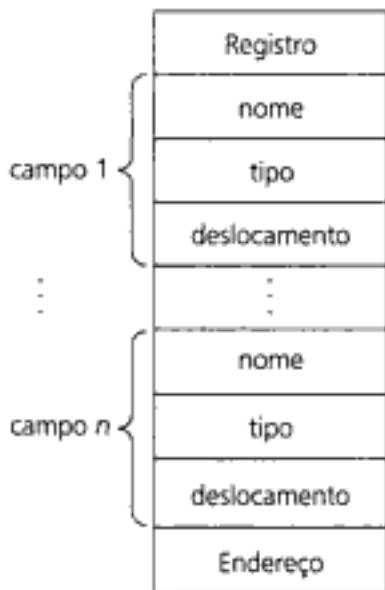
Os registros e as matrizes são formas estruturais estreitamente relacionadas e, por conseguinte, é interessante compará-los. As matrizes são usadas quando todos os valores de dados têm o mesmo tipo e são processados da mesma maneira. Este é feito facilmente quando há uma maneira sistemática de percorrer estrutura. Tal processamento é bem-sucedido, usando-se subscritos dinâmicos como o método de endereçamento.

Os registros são usados quando um conjunto de valores de dados é heterogêneo, e os diferentes campos não são processados da mesma maneira. Além disso, os campos de um registro, muitas vezes, não precisam ser processados em uma ordem seqüencial particular. Os nomes de campos são como subscritos literais ou constantes. Uma vez que são estáticos, eles oferecem um acesso muito eficiente aos campos. Subscritos dinâmicos poderiam ser usados para acessar campos de registro, mas isso rejeitaria a verificação de tipos e também seria mais lento.

Os registros e as matrizes representam métodos imaginosos e eficientes de preencher duas aplicações separadas, mas relacionadas, de estruturas de dados.

### 6.7.5 Implementação de Tipos Registro

Os campos de registros são armazenados em localizações de memória adjacentes. Entretanto, uma vez que os seus tamanhos não são necessariamente os mesmos, o método de acesso usado para matrizes não é o mesmo que para os registros. Em vez disso, o endereço do deslocamento, em relação ao seu início, é associado a cada campo. Os acessos a estes são manipulados usando-se esses deslocamentos. O descritor em tempo de compilação de um registro tem a forma geral mostrada na Figura 6.8. Descritores em tempo de execução para registros são desnecessários.



**FIGURA 6.8** Um descritor em tempo de compilação para um registro.

## 6.8 Tipos União

Uma **união** é um tipo que pode armazenar diferentes valores de tipo durante a execução do programa. Como um exemplo da necessidade de um tipo união, considere a tabela de constantes para um compilador, usada para armazenar as constantes encontradas em um programa sendo compilado. Um campo de cada entrada da tabela serve para o valor da constante. Suponhamos que, para uma linguagem particular em compilação, os tipos de constantes fossem inteiro, real e booleano. Em termos de gerenciamento da tabela, seria conveniente se a mesma localização, o campo da tabela, pudesse armazenar um valor de qualquer um destes três. Então, todos os valores de constantes poderiam ser endereçados da mesma maneira. O tipo dessa localização é, em certo sentido, a união dos três valores de tipos que ela pode armazenar.

### 6.8.1 Questões de Projeto

O problema da verificação de tipo dos tipos união, discutidos no Capítulo 5, leva a uma questão de projeto importante. A outra questão fundamental é como representar sintaticamente uma união. Em alguns casos, as uniões limitam-se a fazer parte das estruturas de registro, mas, em outros, não. Assim, as principais questões de projeto particulares aos tipos união são as seguintes:

- A verificação de tipos deve ser exigida? Note que qualquer verificação de tipos deve ser dinâmica.
- As uniões devem ser incorporadas aos registros?

### 6.8.2 Uniões Livres

O FORTRAN, o C e o C++ oferecem construções de união em que não há nenhum suporte de linguagem para verificação de tipos. No FORTRAN, a instrução EQUIVALENCE é usada para especificar uniões; no C e no C++, é a construção union. Nessas linguagens, elas são chamadas de **uniões livres**, porque é permitida completa liberdade aos programadores quanto a verificação de tipos em seu uso.

### 6.8.3 Tipos União Pascal

A verificação de tipos em uniões exige que cada construção de união inclua um indicador de tipo. Esse indicador é chamado **marca** ou **discriminante** e uma união com discriminante é chamada **união discriminada**. A primeira linguagem a oferecer a união discriminada foi o ALGOL 68.

O Pascal introduziu o conceito de integrar uniões discriminadas com uma estrutura de registro. Essa idéia foi transportada para a Ada. Nessas linguagens, a união discriminada é chamada **registro variante** ou parte variante de um registro. O discriminante é uma variável acessível ao usuário no registro que armazena o valor de tipo atual da variante. O exemplo seguinte ilustra um registro Pascal com uma parte variante:

```

type forma = (circulo, triangulo, retangulo);
cores = (vermelho, verde, azul);
figura =
record
    preenchido : boolean;
    cor : cores;
    case aspecto : forma of
        circulo: (diametro : real);
        triangulo: (ladoesquerdo : integer;
                    ladodireito : integer;
                    angulo : real);
        retangulo: (lado1 : integer;
                    lado2 : integer)
    end;
var minhafigura : figura;

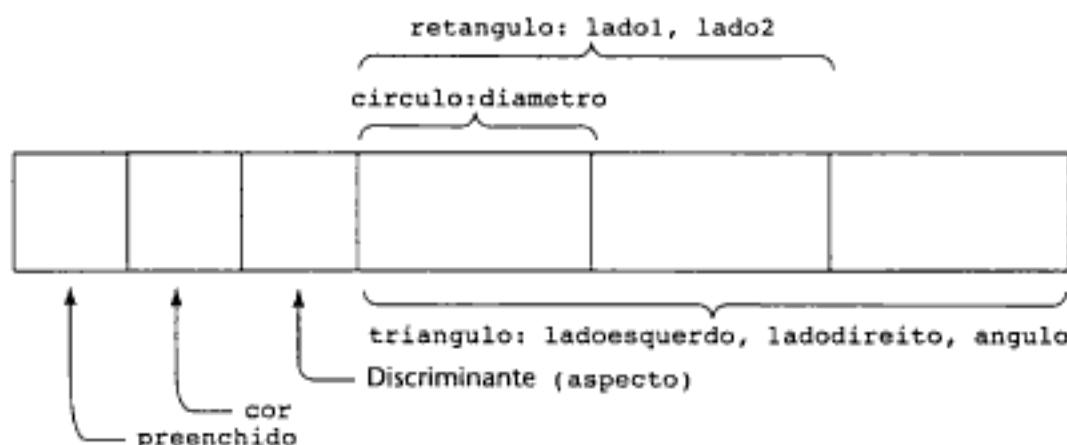
```

A estrutura desse registro variante é mostrada na Figura 6.9, na qual presumimos que os números inteiros exigem a mesma quantidade de armazenamento dos reais. A variável **figura** consiste na marca, que tem o nome **aspecto** e suficiente armazenagem para sua maior variante. Nesse caso, a maior variante é para **triangulo**, que consiste em dois inteiros e em um real. Em qualquer momento durante a execução, a marca deve indicar qual variante está atualmente armazenada. Se a variante precisasse ser impressa, poderíamos usar o seguinte:

```

case minhafigura.aspecto of
    circulo: writeln('É um círculo; seu diâmetro é:',
                      minhafigura.diametro);
    triangulo: begin
        writeln('É um triângulo');
        writeln(' seus lados são: ', minhafigura.ladoesquerdo,
                minhafigura.ladodireito);
    end;

```



**FIGURA 6.9** Uma união discriminada de três variáveis de forma (supondo que todas as variáveis são do mesmo tamanho).

```

        writeln(' o ângulo entre os lados é:',
               minhafigura.angulo);
      end;
retangulo: begin
    writeln (' É um retângulo');
    writeln (' seus lados são:', minhafigura.ladol,
              minhafigura.lado2)
end
end

```

Há dois problemas distintos no projeto que tornam a verificação de tipos impossível, mesmo tendo havido uma tentativa no Pascal de, pelo menos, considerar esta verificação em seus registros variantes. O primeiro é que o programa pode trocar a marca sem fazer a correspondente mudança na variante. Portanto, mesmo que o sistema verifique o tipo da variante em tempo de execução ao examinar a marca antes de usá-la, ele poderia não detectar todos os erros de tipo; o programa pode ter mudado a marca, de modo que seu valor, agora, é incoerente com o tipo da variável atual. Eis uma razão pela qual os implementadores tipicamente ignoram a verificação de tipos de registros variantes nessas linguagens.

O segundo problema é que o programador pode simplesmente omitir a marca da estrutura do registro variante, transformando-a em uma união livre. Considere o seguinte exemplo:

```

type
figura =
record
...
case forma of
  circulo : (diametro : real);
  triangulo : (ladoesquerdo : integer);
...
end

```

Com essa estrutura, nem o usuário, nem o sistema têm qualquer maneira de determinar o tipo variante atual. Suponhamos que o valor de `minhafigura.diametro` seja 2,73. Não existe maneira alguma de proteger-se de referências incorretas como, por exemplo,

```

lado := minhafigura.ladoesquerdo;

```

que raramente é útil porque, atualmente, há um valor fracionário na parte variante onde `ladoesquerdo` reside. Tanto `minhafigura.diametro` como `minhafigura.ladoesquerdo` podem ser atribuídas e referenciadas a qualquer momento.

Os registros variantes do Pascal, às vezes, são usados para contornar algumas das restrições da linguagem. Eles constituem uma brecha conveniente nas regras de verificação de tipos. Por exemplo, a aritmética de ponteiros não é permitida no Pascal, ainda que algumas aplicações precisem manipular valores deles. Por exemplo, o sistema de gerenciamento de armazenamento dinâmico usa a aritmética de ponteiros para computar o endereço de áreas da memória a serem alocadas. Para contornar as restrições a ela, um ponteiro pode ser colocado em uma variante com um número inteiro, e a forma deste pode ser manipulada quando necessário. Essa aplicação particular dos registros variantes não é necessária no C, no C++ ou na Ada, uma vez que todos eles oferecem outros métodos para fazer aritmética de ponteiros ou de endereços.

### 6.8.4 Tipos União Ada

A Ada estende a forma Pascal dos registros variantes para torná-los mais seguros. Ambos os problemas associados com os registros variantes Pascal são evitados. A marca não pode ser mudada sem que a variante também seja mudada, e ela é necessária em todos os registros variantes. Além disso, exige-se que os sistemas Ada verifiquem a marca de todas as referências a variantes.

O projeto da Ada permite que o usuário especifique variáveis de um tipo de registro variante que armazenem somente um dos valores de tipo possíveis nela. Assim, o usuário poderá dizer ao sistema quando a verificação de tipos pode ser estática. Uma variável assim tão restrita é chamada de **variável variante restringida**.

A marca desta variável é tratada como uma constante nomeada. Os registros variantes não-restringidos na Ada se assemelham mais aos seus contrapartes Pascal em razão de que os valores de suas variantes podem mudar de tipo durante a execução. Porém, o tipo da variante pode ser mudado somente atribuindo-se o registro inteiro, inclusive o discriminante. Isso rejeita registros incoerentes, pois, se o registro recém-atribuído for um agregado de dados constante, o valor da marca e o tipo da variante poderão ser estaticamente verificados quanto à coerência. Se o valor atribuído for uma variável, sua coerência foi assegurada quando ele foi atribuído, de modo que o novo valor dela, que agora é atribuído, certamente é coerente.

O exemplo seguinte mostra a versão Ada do registro variante Pascal definido anteriormente:

```

type FORMA is (CIRCULO, TRIANGULO, RETANGULO) ;
type CORES is (VERMELHO, VERDE, AZUL);
type FIGURA (ASPECTO : FORMA) is
    record
        PREENCHIDO : BOOLEAN;
        COR : CORES;
        case ASPECTO is
            when CIRCULO =>
                DIAMETRO : FLOAT;
            when TRIANGULO =>
                LADO_ESQUERDO : INTEGER;
                LADO_DIREITO : INTEGER;
                ANGULO : FLOAT;
            when RETANGULO =>
                LADO_1 : INTEGER;
                LADO_2 : INTEGER;
        end case;
    end record;

```

As duas instruções seguintes declaram variáveis do tipo FIGURA:

```

FIGURA_1 : FIGURA;
FIGURA_2 : FIGURA(ASPECTO => TRIANGULO);

```

`FIGURA_1` é declarada como um registro variante não-restringido que não tem valor inicial. Seu tipo pode mudar pela atribuição de um registro inteiro, incluindo o discriminante, como no seguinte:

```

FIGURA_1 := (PREENCHIDO => true,
              COR => AZUL,
              ASPECTO => RETANGULO,
              LADO_1 => 12,
              LADO_2 => 3);

```

O lado direito dessa atribuição é um agregado de dados.

A variável FIGURA\_2 declarada acima é restrinida para ser um triângulo e não pode ser mudada para outra variante.

Essa forma de união discriminada é perfeitamente segura, porque sempre permite verificação de tipos, não obstante as referências a campos em variantes não-restrinidas precisarem ser verificadas dinamicamente.

### 6.8.5 Avaliação

As uniões são construções potencialmente inseguras em muitas linguagens. Elas constituem um dos motivos pelos quais linguagens como o FORTRAN, o Pascal, o C e o C++ não são fortemente tipificadas, pois não permitem verificação de tipos das referências a suas uniões.

Por outro lado, as uniões proporcionam flexibilidade de programação; por exemplo, no Pascal, sua presença permite a aritmética de ponteiros. Além disso, podem ser projetadas para que possam ser usadas com segurança, como na Ada. Na maioria das outras linguagens, as uniões devem ser usadas com cuidado.

Um grande número de linguagens não inclui uniões. Entre elas estão a Oberon, o Modula-3 e o Java. Isso talvez reflita a grande preocupação com a segurança nas linguagens de programação.

### 6.8.6 Implementação de Tipos União

As uniões discriminadas são implementadas simplesmente usando-se o mesmo endereço para toda variante possível. É alocado um armazenamento suficiente para a maior variante. No caso das variantes restrinidas da Ada, a quantidade exata de armazenamento pode ser usada porque não há variações. A marca de uma união discriminada é armazenada com a variante em uma estrutura assemelhada a registros.

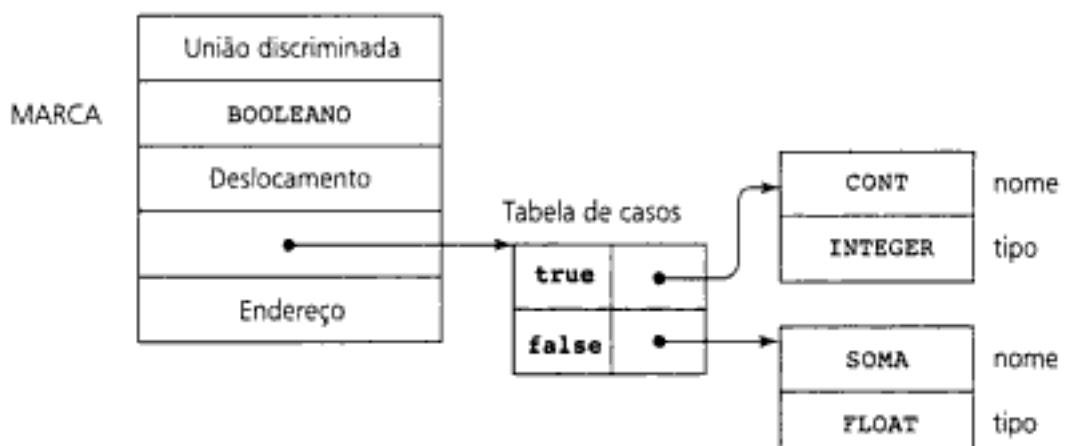
Durante a compilação, a descrição completa de cada variante deve ser armazenada. Isso pode ser feito associando-se uma tabela de casos com a entrada da marca no descritor. A primeira tem uma entrada para cada variante que aponta para um descritor dessa variante em particular. Para ilustrar esse arranjo, considere o seguinte exemplo Ada:

```

type VERTICE (MARCA : BOOLEAN) is
  record
    case MARCA is
      when true => CONT : INTEGER;
      when false => SOMA : FLOAT;
    end case;
  end record;

```

O descritor para esse tipo poderia ter a forma mostrada na Figura 6.10 (ver p. 248).



**FIGURA 6.10** Um descritor em tempo de compilação para uma união discriminada.

## 6.9 Tipos Conjunto

Um tipo **conjunto** (set) é aquele cujas variáveis podem armazenar coleções não-ordenadas de valores distintos de algum tipo ordinal chamado de seu **tipo básico**. Os tipos conjunto freqüentemente são usados para modelar conjuntos matemáticos. Por exemplo, a análise de texto, muitas vezes, exige que pequenos conjuntos de caracteres, como de pontuação ou de vogais, sejam armazenados e convenientemente pesquisados.

A única questão de projeto particular aos tipos conjunto é: qual deve ser o número máximo de elementos em um conjunto?

### 6.9.1 Conjuntos no Pascal

Entre as linguagens imperativas comuns, somente o Pascal inclui conjuntos como tipos de dados. Descreveremos agora, brevemente, o tipo de dados conjunto da linguagem Pascal.

O tamanho máximo dos conjuntos básicos Pascal depende da implementação. Muitas delas restringem-no severamente, muitas vezes, a menos de 100. Elas fazem isso porque os conjuntos e suas operações são mais eficientemente implementadas representando-se as variáveis de conjunto como cadeias de bits que cabem em uma única palavra de máquina.

Um problema de permitir que o tamanho da palavra de máquina determine a extensão máxima do conjunto básico é que os usuários limitam-se a modelar somente conjuntos muito pequenos. Essa é uma deficiência de capacidade de escrita. Outro problema é que diferentes máquinas têm diferentes tamanhos de palavra, de modo que programas desenvolvidos em máquinas com tamanhos de palavra maiores e que usam conjuntos maiores podem não ser portáveis a máquinas com tamanhos de palavra menores. Ambos os problemas resultam do fato de que o tamanho máximo do conjunto básico é escolhido pelos implementadores, em vez de torná-lo parte do projeto da linguagem.

O Pascal inclui uma coleção de operações com conjuntos, como união, intersecção e igualdade de conjuntos.

O exemplo seguinte mostra a definição de um conjunto e algumas de suas variáveis.

```
type cores = (vermelho, azul, verde, amarelo, laranja, branco,
               preto);
type conjcores = set of cores;
var conj1, conj2 : conjcores;
```

Valores constantes podem ser atribuídos às variáveis `conj1` e `conj2`, como em

```
conj1 := {vermelho, azul, verde, amarelo, branco};
conj2 := {preto, azul};
```

Conjuntos comumente são usados para simplificar e para abreviar expressões booleanas OR compostas. Por exemplo,

```
if (ch = 'a') or (ch = 'e') or (ch = 'i') or (ch = 'o')
  or (ch = 'u') ...
```

pode ser substituída por

```
if ch in ['a', 'e', 'i', 'o', 'u'] ...
```

### 6.9.2 Avaliação

A Ada não inclui o tipo conjunto, embora tenha se baseado no Pascal. Ao contrário, os projetistas da Ada adicionaram o operador pertinência dos conjuntos para seus tipos enumeração. Isso proporciona uma das operações mais comumente necessárias. Obviamente, os conjuntos que podem ser testados quanto à pertinência são apenas valores de enumeração constantes.

Em outras linguagens sem tipos conjunto, as operações com conjuntos devem ser feitas com vetores, e o usuário deve escrever o código para fornecê-las. Isso não é difícil, ainda que seja, de fato, mais incômodo e, muito provavelmente, bem menos eficiente. Por exemplo, se o conjunto de vogais fosse representado como um vetor `char` no Pascal, identificar se determinada variável armazenou uma vogal exigiria um laço para procurá-la no vetor de vogais. Se estas fossem representadas como um conjunto, a mesma determinação poderia ser feita com uma aplicação do operador `in`. Isso não somente é eficiente para o programador, mas, provavelmente, também será eficiente para o computador. Em ambos os casos, é melhor porque pode-se lidar com o conjunto inteiro como uma unidade, enquanto o vetor deve ser pesquisado um elemento por vez.

Os vetores são, é claro, bem mais flexíveis do que os conjuntos: eles permitem muito mais operações, formas mais complexas e mais opções para tipos de elemento. De fato, se os vetores fossem restringidos a um tamanho máximo de 32, como são os conjuntos em muitas implementações Pascal, os usuários não os considerariam aceitáveis. Os conjuntos oferecem uma alternativa que troca flexibilidade por eficiência para certa classe de aplicações.

### 6.9.3 Implementação de Tipos Conjunto

Normalmente, os conjuntos são armazenados como cadeias de bits na memória. Por exemplo, se um conjunto tiver o tipo básico ordinal

[ 'a' . . . 'p' ]

as variáveis deste conjunto poderão usar os primeiros 16 bits de uma palavra de máquina, com cada bit ligado (1) representando um elemento presente, e cada bit desligado (0) representando um elemento ausente. Usando tal esquema, o valor do conjunto

[ 'a' , 'c' , 'h' , 'o' ]

seria representado como

1010000100000010

A compensação dessa abordagem é que uma operação típica, como uma união de conjuntos, pode ser computada como uma única instrução de máquina, um OR lógico. A pertinência a um conjunto também pode ser feita em uma única instrução quando a cardinalidade básica do conjunto for menor ou igual ao tamanho da palavra de máquina. Por exemplo, se tivéssemos uma variável de conjunto chamada **setchars**, e o teste de pertinência fosse

'g' in setchars

o processo poderia ser feito com uma operação AND entre as representações da cadeia de bits dos dois operandos.

## 6.10 Tipos Ponteiro

Um tipo ponteiro é aquele em que as variáveis têm uma faixa de valores que consiste em endereços de memória e um valor especial, o **nil**. O **nil** não é um endereço válido e serve para indicar que um ponteiro não pode ser usado atualmente para referenciar qualquer célula de memória.

Os ponteiros foram projetados para dois tipos distintos de usos. Primeiro, eles apresentam parte do poder de endereçamento indireto, muito usado em programação em linguagem de montagem. Em segundo lugar, fornecem um método de gerenciamento de armazenamento dinâmico. Um ponteiro pode ser usado para acessar uma localização na área em que o armazenamento é alocado dinamicamente, chamada usualmente de **monte**.

Variáveis alocadas dinamicamente no monte são chamadas **dinâmicas no monte**. Muitas vezes, elas não têm identificadores associados a si mesmas e, assim, podem ser referenciadas somente pelo ponteiro ou por variáveis de tipo de referência. As variáveis sem nomes são chamadas **variáveis anônimas**. É nessa última área de aplicação dos ponteiros que surgem as questões de projeto mais importantes.

Os ponteiros, diferentemente das matrizes e dos registros, não são tipos estruturados, ainda que sejam definidos um operador de tipo (\* no C e no C++, **access** na Ada e ^ no Pascal). Além disso, eles também diferem das variáveis escalares porque são mais frequentemente usados para referenciar alguma outra variável, em vez de armazenarem dados de alguma espécie.

Ambos os tipos de usos adicionam capacidade de escrita a uma linguagem. Por exemplo, suponhamos que seja necessário implementar uma estrutura dinâmica, como uma árvore binária em uma linguagem como o FORTRAN 77, que não tem ponteiros. Isso exigiria que o programador oferecesse e mantivesse um conjunto de vértices disponíveis da árvore, que, provavelmente, seriam implementados em vetores paralelos. Além disso, devido à falta de armazenamento dinâmico no FORTRAN 77, seria necessário que o programador adi-

vinhasse o número máximo de vértices exigidos. Evidentemente, esta é uma maneira cansativa e complicada de lidar com árvores binárias.

### 6.10.1 Questões de Projeto

As principais questões de projeto particulares aos ponteiros são as seguintes:

- Quais são o escopo e o tempo de vida de uma variável de ponteiro?
- Qual é o tempo de vida de uma variável dinâmica no monte?
- Os ponteiros são restritos quanto ao tipo de valor para o qual eles apontam?
- Os ponteiros são usados para gerenciamento de armazenamento dinâmico, para endereçamento indireto, ou para ambos?
- A linguagem deve suportar tipos ponteiro, tipos referência ou ambos?

### 6.10.2 Operações com Ponteiros

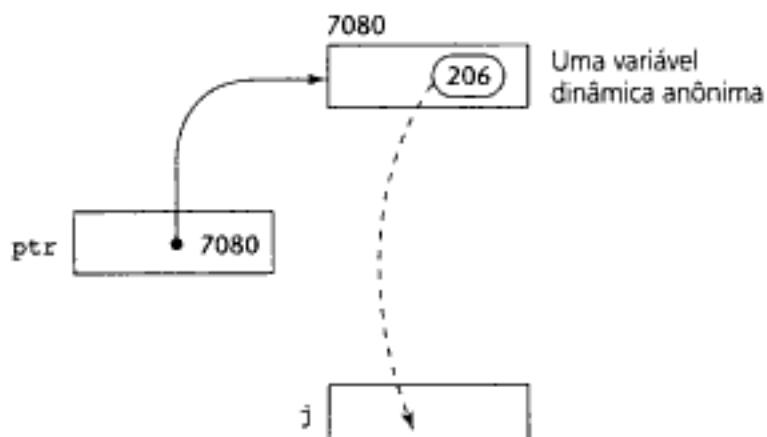
As linguagens que oferecem um tipo ponteiro normalmente incluem duas operações fundamentais com ponteiros: atribuição e desreferenciamento. A primeira operação fixa o valor de uma variável de ponteiro em um endereço útil. Se variáveis de ponteiro forem usadas somente para gerenciar o armazenamento dinâmico, o mecanismo de alocação, seja pelo operador ou pelo subprograma incorporado, serve para inicializá-la. Se os ponteiros forem usados para endereçamento indireto a variáveis que não são dinâmicas no monte, deve haver um operador explícito ou um subprograma incorporado para buscar o endereço de uma variável, que poderá, então, ser atribuído a uma variável de ponteiro.

Uma ocorrência de uma variável de ponteiro em uma expressão pode ser interpretada de duas maneiras distintas. Primeiro, como uma referência ao conteúdo da célula de memória à qual a variável está vinculada, que, por sua vez, no caso de um ponteiro, é um endereço. É exatamente assim que uma variável não-ponteiro em uma expressão seria interpretada, ainda que ela não seja um endereço nesse caso. Porém, o ponteiro também poderia ser interpretado como uma referência ao valor da célula de memória cujo endereço está na célula de memória em que a variável está vinculada. Neste caso, é interpretado como uma referência indireta. O caso anterior é uma referência normal a ponteiro; o último é o resultado de **desreferenciar** o ponteiro. O desreferenciamento, que pega uma referência por meio de um nível de indireção, é a segunda operação fundamental com ponteiros. Para clarificá-la, considere uma variável de ponteiro, `ptr`, vinculada a uma célula de memória com o valor 7080. Suponhamos que a célula de memória cujo endereço é 7080 tenha o valor 206. Uma referência normal a `ptr` produz 7080, mas uma referência realocada a `ptr` produz 206.

O desreferenciamento pode ser explícito ou implícito. No FORTRAN 90 ele é implícito, mas, na maioria das outras linguagens contemporâneas, ocorre somente quando explicitamente especificado. No C++, ele é explicitamente especificado com o asterisco (\*) como uma operação unária pré-fixada. Por exemplo, se `ptr` for um ponteiro com o valor 7080, como no exemplo acima, e a célula cujo endereço é 7080 tem o valor 206, a atribuição

```
j = *ptr
```

fixará `j` em 206. Esse processo é mostrado na Figura 6.11 (ver p. 252).



**FIGURA 6.11** A operação de atribuição `j = *ptr`.

Quando ponteiros apontam para registros, a sintaxe das referências aos campos desses registros varia entre as linguagens. No C e no C++, há duas maneiras pelas quais um ponteiro para um registro pode ser usado para referenciar um campo desse registro. Se uma variável de ponteiro `p` apontar para um registro dentro de um campo que tem o nome `age`, `(*p).age` poderá ser usado para referir-se a esse campo. O operador `->`, quando usado entre um ponteiro para um registro e um campo desse registro combina desreferenciamento e referência a campo. Por exemplo, a expressão `p->age` é equivalente a `(*p).age`. No Pascal, a mesma referência é feita com `p^.age`. Na Ada, `p.age` pode ser usada, porque esses usos de ponteiros são implicitamente desreferenciados.

As linguagens que oferecem ponteiros para o gerenciamento de monte devem incluir uma operação de alocação explícita. Algumas também oferecem uma operação de desalocação explícita. Essas duas operações freqüentemente estão na forma de subprogramas incorporados, embora Ada, C++ e Java usem um operador alocador.

### 6.10.3 Problemas com Ponteiros

A primeira linguagem de programação de alto nível a incluir variáveis de ponteiro foi a PL/I, cujos ponteiros podem ser usados para referir-se tanto a variáveis dinâmicas no monte como a outras de programa. Os ponteiros da PL/I são altamente flexíveis, mas o seu uso pode levar a diversos erros de programação. Alguns dos problemas dos ponteiros da PL/I também estão presentes nas linguagens subsequentes. As mais recentes, como o Java, substituíram completamente os ponteiros por tipos referência, que, juntamente com a desalocação implícita, eliminam os principais problemas com eles.

#### 6.10.3.1 Ponteiros Pendurados

Um **ponteiro ou referência pendurada**, é um ponteiro que contém o endereço de uma variável dinâmica no monte desalocada. Os ponteiros pendurados são perigosos por várias razões. Primeiro, a localização para a qual se aponta pode ter sido realocada para alguma nova variável dinâmica no monte. Se a nova variável não for do mesmo tipo que a antiga, o

uso do ponteiro pendurado será inválido. Mesmo que a nova seja do mesmo tipo, seu novo valor não terá relação com o conteúdo desreferenciado do ponteiro antigo. Além disso, se o ponteiro pendurado for usado para mudar a variável dinâmica no monte, o novo valor dela será destruído. Finalmente, é possível que a localização, agora, esteja sendo usada temporariamente pelo sistema de gerenciamento de armazenamento, possivelmente como um ponteiro em uma cadeia de blocos de armazenamento disponíveis, permitindo, assim, que uma mudança na localização faça com que o gerenciador de armazenagem falhe.

A seguinte seqüência de operações cria um ponteiro pendurado em muitas linguagens:

1. O ponteiro `p1` é ajustado para apontar para uma nova variável dinâmica no monte.
2. O valor de `p1` é atribuído ao ponteiro `p2`.
3. A variável dinâmica no monte apontada por `p1` é explicitamente desalocada (fixando `p1` em `nil`), mas `p2` não é mudado pela operação. `P2`, agora, será um ponteiro pendurado.

#### 6.10.3.2 Variáveis Dinâmicas no Monte Perdidas

Uma **variável dinâmica no monte perdida** é uma variável dinâmica no monte aloca-  
da não mais acessível ao programa usuário. Essas variáveis, muitas vezes, são chamadas de **lixo** porque não são úteis para seu propósito original e também não podem ser realocadas para algum novo uso. As variáveis dinâmicas no monte perdidas, mais freqüentemente, são criadas pela seguinte seqüência de operações:

1. O ponteiro `p1` é ajustado para apontar para uma variável dinâmica no monte recém-criada.
2. Mais tarde, `p1` é ajustado para apontar para outra variável dinâmica no monte recém-criada.

Agora, a primeira variável dinâmica no monte é inacessível ou perdida.

As linguagens que exigem desalocação explícita de variáveis dinâmicas compartilham do problema das variáveis dinâmicas no monte perdidas. Às vezes, ele é chamado de **vazamento de memória**. Nas seções seguintes, investigaremos como os projetistas têm lidado com os problemas dos ponteiros pendurados e com os das variáveis dinâmicas no monte perdidas.

#### 6.10.4 Ponteiros no Pascal

No Pascal, os ponteiros são usados somente para acessar variáveis anônimas alocadas dinamicamente. Ponteiros pendurados podem ser criados facilmente porque ele inclui uma operação de desalocação explícita. Um programador Pascal cria variáveis dinâmicas no monte com `new` e as destrói com `dispose`. Para destruí-las com segurança, seria necessário que a função `dispose` localizasse e fixasse em `nil` todos os ponteiros que apontam para aquela que está sendo destruída. Infelizmente, esse processo é complexo e custoso e, em consequência disso, raramente é implementado dessa maneira. Eis um problema para todos os processos de desalocação explícita nas linguagens de programação.

Os implementadores do Pascal escolhem entre as seguintes alternativas para desalocação explícita:

- Simplesmente ignorar o `dispose`, em cujo caso nenhuma desalocação é feita, e nenhum dos ponteiros que estavam apontando para o objeto dinâmico no monte em questão é mudado.
- Não incluir `dispose` na linguagem, tornando-o uma instrução ilegal.
- Desalocar, de fato, a variável dinâmica no monte em questão e fixar em `nil` o ponteiro de parâmetro para `dispose`, criando, assim, ponteiros pendurados a partir de quaisquer outros ponteiros que estejam apontando para o objeto.
- Implementar `dispose` completa e corretamente, inibindo os ponteiros pendurados. (O autor não conhece qualquer implementação Pascal que tenha escolhido tal alternativa).

### 6.10.5 Ponteiros na Ada

A Ada oferece ponteiros, chamados tipos-**acesso**, semelhantes aos do Pascal. Na Ada, entretanto, o problema dos ponteiros pendurados é parcialmente aliviado pelo projeto da linguagem. Uma variável dinâmica no monte pode ser (conforme a opção do implementador) desalocada implicitamente no final do escopo de seu tipo-ponteiro, diminuindo drasticamente a necessidade de desalocação explícita. Uma vez que variáveis dinâmicas no monte somente podem ser acessadas por variáveis de um tipo, quando o final do escopo dessa declaração de tipo é alcançada, não pode restar nenhum ponteiro apontando para o objeto. Isso diminui o problema porque uma desalocação explícita implementada inadequadamente é a maior fonte de ponteiros pendurados. Infelizmente, a Ada também tem uma desalocação explícita, `UNCHECKED_DESLOCATION`. Seu nome pretende desencorajar seu uso ou, pelo menos, avisar ao usuário sobre seus potenciais problemas.

O problema da variável dinâmica no monte perdida não é eliminado pelo projeto de ponteiros da Ada.

Uma outra pequena melhoria nos ponteiros Ada em relação aos do Pascal é a exigência que a linguagem faz de que todos os ponteiros sejam implicitamente inicializados em `null` (a versão da Ada para `nil`). Isso impede acessos inadvertidos a localizações aleatórias na memória porque o usuário esqueceu de inicializar um ponteiro antes de usá-lo.

### 6.10.6 Ponteiros em C e C++

No C e no C++, ponteiros podem ser usados de uma maneira muito semelhante àquela em que os endereços são usados em linguagens de montagem. Isso significa que eles são extremamente flexíveis, mas devem ser usados com grande cuidado. Essas linguagens não oferecem qualquer solução para os problemas dos ponteiros pendurados ou das variáveis dinâmicas no monte perdidas. Porém, o fato da aritmética de ponteiros ser possível no C e no C++ torna seus ponteiros mais interessantes do que os das outras linguagens de programação.

Diferentemente dos ponteiros do Pascal e da Ada 83, que somente podem apontar para dentro do monte, os ponteiros do C e do C++ podem apontar para, virtualmente, qualquer variável em qualquer lugar da memória.

No C e no C++, o asterisco (\*) denota a operação de desreferenciamento, e o E comercial (&) denota o operador para produzir o endereço de uma variável. Por exemplo, no código

```
int *ptr;
int cont, init;
```

```
...
ptr = &init;
cont = *ptr;
```

as duas instruções de atribuição são equivalentes à atribuição única:

```
cont = init;
```

A atribuição da variável `ptr` fixa `ptr` no endereço de `init`. A primeira atribuição a `cont` desreferencia `ptr` para produzir o valor em `init`, então atribuído a `cont`. Assim, o efeito das duas primeiras instruções de atribuição é atribuir o valor de `init` a `cont`. Note que a declaração de um ponteiro especifica seu tipo de domínio.

Ponteiros podem receber o valor de endereço de qualquer objeto do tipo de domínio correto e a constante zero, usada para `nil`.

A aritmética de ponteiros também é possível em algumas formas restritas. Por exemplo, se `ptr` for uma variável de ponteiro declarada para apontar para algum objeto de algum tipo de dados, então

```
ptr + indice
```

é uma expressão legal. A semântica é a seguinte: em vez de simplesmente adicionar o valor de `indice` a `ptr`, seu valor é dimensionado pelo tamanho da célula de memória (em unidades de memória) para a qual `ptr` está apontando. Por exemplo, se `ptr` apontar para uma célula de memória de um tipo que tenha quatro unidades de memória de tamanho, `indice` será multiplicado por 4, e o resultado será adicionado a `ptr`. A principal finalidade desse tipo de aritmética de endereços é a manipulação de matrizes. A discussão seguinte está relacionada somente a matrizes unidimensionais.

No C e no C++, todos os vetores usam zero como o limite inferior de suas faixas de subscrito, e os seus nomes sem subscritos sempre se referem ao endereço do primeiro elemento. De fato, um nome de vetor sem um subscrito é tratado exatamente como um ponteiro, exceto que ele é uma constante e, portanto, não pode ter valor atribuído. Considere as seguintes declarações:

```
int list [10];
int *ptr;
```

Considere a atribuição de inicialização

```
ptr = list;
```

que atribui o endereço de `list[0]` a `ptr`, porque um nome de vetor sem um subscrito é interpretado como endereço-base do vetor. Dada essa atribuição, podemos concluir que

```
*(ptr + 1) é equivalente a list[1],
*(ptr + indice) é equivalente a list[indice] e
ptr[indice] é equivalente a list[indice]
```

A partir dessas instruções, fica claro que as operações com ponteiros incluem o mesmo dimensionamento usado em operações de indexação. Além disso, ponteiros para vetores podem ser indexados como se fossem nomes de vetor.

Ponteiros podem apontar para funções. Esse recurso é usado para passar funções como parâmetros de outras funções. Ponteiros também são usados para passagem de parâmetros, conforme discutiremos no Capítulo 9.

O C e o C++ incluem ponteiros do tipo `void *`, o que significa que eles podem apontar para valores de qualquer tipo. Eles são, com efeito, ponteiros genéricos. Porém, a

verificação de tipos não é um problema com ponteiros `void *`, porque eles não podem ser desreferenciados. Eles são usados comumente como parâmetros de funções que operam na memória. Por exemplo, suponha que desejamos que uma função move uma seqüência de bytes de um lugar da memória para outro. Seria mais geral se pudesse ser passados dois ponteiros de qualquer tipo. Isso seria válido se os parâmetros formais correspondentes na função fossem do tipo `void *`. A função poderia, então, convertê-los para o tipo `char *` e fazer a operação, independentemente do tipo dos ponteiros enviados como parâmetros reais.

### 6.10.7 Ponteiros no FORTRAN 90

No FORTRAN 90, os ponteiros são usados para apontar tanto para variáveis dinâmicas no monte como para variáveis estáticas. Por exemplo, em

```
INTEGER, POINTER :: INT_PTR  
INTEGER, POINTER, DIMENSION () :: INT_LIST_PTR
```

`INT_PTR` pode apontar para qualquer valor do tipo `INTEGER` e `INT_LIST_PTR` pode apontar para qualquer vetor unidimensional de elementos `INTEGER`.

Os ponteiros FORTRAN 90 são implicitamente desreferenciados na maioria dos usos. Por exemplo, quando um ponteiro aparece em uma expressão normal, ele é sempre implicitamente desreferenciado. Uma forma da instrução de atribuição especial,

```
pointer => target
```

é usada quando o desreferenciamento não é desejado. Essa atribuição é usada para ajustar variáveis de ponteiro direcionados a variáveis particulares e, também, para defini-los de maneira que tenham os valores de endereço de outros ponteiros. Qualquer variável para a qual se deva direcionar um ponteiro deve ter o atributo `TARGET` definido em sua declaração. Por exemplo, na seguinte

```
INTEGER, TARGET :: PERA  
INTEGER LARANJA
```

pode-se apontar para `PERA` com `INT_PTR`, mas não para `LARANJA`.

Os ponteiros FORTRAN 90 podem facilmente tornar-se pendurados porque a instrução `DEALLOCATE`, que assume um ponteiro como um argumento, não faz nenhuma tentativa de determinar se outros ponteiros estão apontando para uma variável dinâmica no monte que está sendo desalocada.

### 6.10.8 Tipos Referência

O C++ inclui uma espécie única de ponteiro, o tipo referência, usado principalmente para os parâmetros formais em definições de funções. Uma variável de referência é um ponteiro constante que é sempre implicitamente desreferenciado. Uma vez que a variável de referência C++ é uma constante, deve ser inicializada com o endereço de alguma variável em sua definição. Depois da inicialização, uma variável do tipo referência jamais poderá ser fixada para referenciar qualquer outra. O desreferenciamento implícito, é claro, impede a atribuição ao valor de endereço de uma variável de referência.

As variáveis do tipo referência são especificadas nas definições colocando-se Es commerciais (&) antes de seus nomes. Por exemplo,

```
int result = 0;
int &ref_result = result;
...
ref_result = 100;
```

Nesse segmento de código, `result` e `ref_result` são apelidos.

Quando usados como parâmetros formais em definições de funções, os tipos referência proporcionam uma comunicação bidirecional entre a função chamadora e a chamada. Isso não é possível com tipos de parâmetro não-ponteiro, porque os parâmetros C++ são passados por valor. Passar um ponteiro como um parâmetro proporciona a mesma comunicação bidirecional, mas os ponteiros formais exigem desreferenciamento explícito, tornando o código menos legível e menos seguro. Os parâmetros de referência são tratados na função chamada exatamente como os outros parâmetros. A função que faz a chamada não precisa especificar que um parâmetro cujo parâmetro formal correspondente é um tipo referência seja algo incomum. O compilador passa endereços, em vez de valores, para os parâmetros de referência.

No Java, as variáveis de referência são estendidas de sua forma no C++ para uma que as permite substituir ponteiros integralmente. Em sua busca de maior segurança em relação ao C++, os projetistas do Java removeram totalmente os ponteiros do estilo do C e do C++. A diferença fundamental entre ponteiros C++ e referências Java é que aqueles referem-se a endereços da memória, ao passo que estas se referem a instâncias de classe. Isso impede imediatamente que a aritmética para referências seja algo sensato. Por outro lado, ela não desautoriza a atribuição. Assim, diferentemente das variáveis de referência do C++, as do Java podem ser atribuídas para referirem-se a diferentes instâncias de classe. Todas as instâncias de classe do Java podem ser referenciadas por meio de variáveis de referência. Esse, aliás, é o único uso destas últimas no Java. Tais questões serão discutidas adicionalmente no Capítulo 12. No exemplo a seguir, `String` é uma classe Java padrão:

```
String str1;
...
str1 = "Esta é uma cadeia literal Java";
```

Nesse código, `str1` é definido como uma referência à instância de classe ou ao objeto `String`, mas é inicialmente fixada em `null`. A atribuição subsequente define `str1` para referenciar o objeto `String`, "Esta é uma cadeia literal Java".

Uma vez que as instâncias de classe Java são implicitamente desalocadas (não há qualquer operador de desalocação explícito), você não pode ter uma referência pendurada.

### 6.10.9 Avaliação

Os problemas dos ponteiros pendurados e do lixo já foram discutidos extensamente. Os problemas de gerenciamento do monte serão discutidos na Seção 6.10.10.3.

Os ponteiros têm sido comparados com a instrução `goto` que amplia a faixa de instruções a serem executadas em seguida. As variáveis de ponteiro ampliam a faixa de células da memória que podem ser referenciadas por uma variável. Talvez a afirmação mais maldizente

a respeito dos ponteiros tenha sido feita por Hoare (1973): "A introdução deles nas linguagens de alto nível foi um passo para trás, do qual talvez jamais possamos nos recuperar."

As referências Java oferecem parte da flexibilidade e das facilidades dos ponteiros, sem seus riscos. Resta saber se os programadores estarão dispostos a trocar a força total dos ponteiros C e C++ pela maior segurança das referências Java.

### 6.10.10 Implementação de Ponteiros e de Referências

Na maioria das linguagens, os ponteiros são usados no gerenciamento do monte. O mesmo é verdadeiro em relação às referências Java. Assim, não podemos tratar esses dois recursos separadamente. Primeiro, descreveremos brevemente como os ponteiros e como as referências são representados. Depois, discutiremos duas soluções possíveis para o problema dos ponteiros pendurados. Por fim, descreveremos os principais problemas com as técnicas de gerenciamento do monte.

#### 6.10.10.1 Representação de Ponteiros e de Referências

Na maioria dos computadores de maior capacidade, os ponteiros e as referências são valores únicos armazenados em células de memória de dois ou de quatro bytes dependendo do tamanho do espaço de endereçamento da máquina. Porém, a maioria dos microcomputadores baseia-se em microprocessadores Intel, que usam endereços com duas partes: um segmento e um deslocamento (*offset*). Assim, os ponteiros e as referências são implementados, nesses sistemas, como pares de palavras de 16 bits, um para cada uma das duas partes de um endereço.

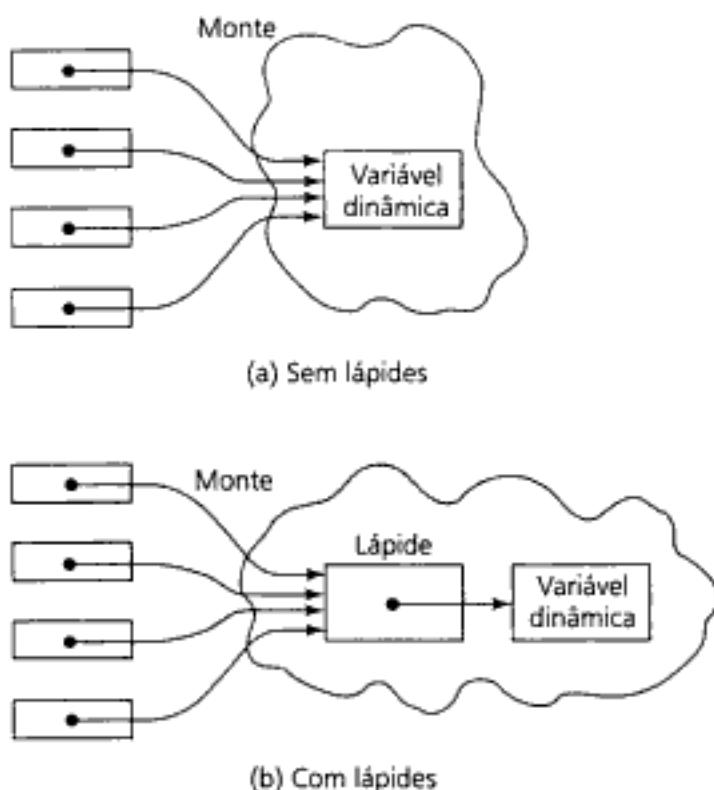
#### 6.10.10.2 Soluções para o Problema dos Ponteiros Pendurados

Os ponteiros pendurados apontam para o armazenamento desalocado. Freqüentemente, eles são criados pela desalocação explícita de variáveis dinâmicas no monte. A desalocação explícita cria ponteiros dinâmicos no monte porque a instrução de desalocação nomeia somente um ponteiro para a variável dinâmica no monte a ser desalocada. Quaisquer outros ponteiros para a variável descartada são transformados em pendurados pelo processo porque o sistema não determinou que existem outros em tempo de execução.

Duas soluções distintas, mas relacionadas, para o problema dos ponteiros pendurados têm sido propostas ou implementadas de fato. Primeiro, há a proposta que usa células extras do monte, chamadas **lápides** (*tombstones*).

As lápides foram propostas por Lomet (1975). A idéia é fazer com que todas as variáveis dinâmicas no monte incluam uma célula especial chamada lápide, que também é um ponteiro para a variável dinâmica no monte. A variável de ponteiro aponta somente para as lápides e jamais para variáveis dinâmicas no monte. Quando a variável dinâmica no monte é desalocada, a lápide permanece mas é ajustada com o valor *nil*, indicando que a variável dinâmica no monte não mais existe. Isso evita que um ponteiro aponte para uma variável desalocada. Qualquer referência a qualquer ponteiro que aponte para uma lápide nula pode ser detectada como um erro. A diferença entre as lápides e os métodos que não as utilizam é mostrada na Figura 6.12 (ver p. 259).

As lápides são custosas tanto em termos de tempo como de espaço. Uma vez que elas nunca são desalocadas, seu armazenamento jamais é reivindicado. Todo acesso a uma variável dinâmica no monte por meio de uma lápide requer mais um nível de indireção, que exige um ciclo de máquina adicional na maioria dos computadores. Aparentemente, ne-



**FIGURA 6.12** Implementação de variáveis dinâmicas com e sem lápides.

nhum dos projetistas das linguagens mais populares achou que a segurança adicional valia o custo, porque nenhuma delas empregou lápides.

Entretanto, elas têm sido consideradas valiosas fora das linguagens de programação. São usadas extensivamente pelo software básico do Macintosh para detectar ponteiros pendurados e para facilitar a realocação dinâmica de células de memória dinamicamente alocadas.

Uma alternativa para as lápides é a abordagem fechaduras-e-chaves (*locks-and-keys*) usada na implementação da UW-Pascal (Fischer e LeBlanc, 1977, 1980). Nesse compilador, os valores de ponteiro são representados como pares ordenados (chave, endereço), em que a chave é um valor inteiro. As variáveis dinâmicas no monte são representadas pelo seu armazenamento, mais uma célula de cabeçalho que armazena um valor de fechadura (*lock*) inteiro. Quando uma variável dinâmica no monte é alocada, um valor de fechadura é criado e colocado na sua célula de fechadura e na célula-chave do ponteiro especificado na chamada a `new`. Todo acesso ao ponteiro desreferenciado compara o valor de sua chave com o de fechadura da variável dinâmica no monte. Se eles coincidirem, o acesso será legal; caso contrário, ele será tratado como um erro de execução. Quaisquer cópias do valor de ponteiro para outros ponteiros devem copiar o valor da chave. Portanto, qualquer número de ponteiros pode referenciar determinada variável dinâmica no monte. Quando esta é desalocada com `dispose`, seu valor de fechadura é mudado para um ilegal. Então, se um outro ponteiro que não aquele especificado na função `dispose` for desreferenciado, seu valor de endereço ainda permanecerá intacto, mas o de sua chave não mais coincidirá com a fechadura; assim, o acesso será negado.

Conforme afirmamos anteriormente, uma vez que o Java não tem uma operação de desalocação explícita para a memória do monte, suas referências jamais são penduradas.

### 6.10.10.3 Gerenciamento do Monte

O gerenciamento do monte pode ser um processo em tempo de execução muito complexo. Examinaremos o processo em duas situações distintas: uma em que todo o armazenamento do monte é alocado e desalocado em unidades de um único tamanho e outra em que segmentos de tamanho variável são alocados e desalocados. Nossa discussão será breve e longe de ser abrangente, uma vez que uma análise cuidadosa desses processos e de seus problemas associados não é tanto uma questão de projeto da linguagem, mas de implementação.

**Células de Tamanho Único.** A situação mais simples ocorre quando toda a alocação e desalocação provier de uma célula de tamanho único. Ela é mais simplificada ainda, quando contém um ponteiro. Esse é o cenário de muitas implementações do LISP, em que problemas de alocação de armazenamento dinâmico foram encontrados, pela primeira vez, em grande escala. Todos os programas LISP e a maioria dos seus dados consistem em células ligadas a listas encadeadas. Alguns processos de gerenciamento de cadeias também estão envolvidos, mas nós os ignoramos aqui.

Em um monte de alocação de tamanho único, todas as células disponíveis são ligadas, usando os seus ponteiros, formando uma lista de espaço disponível. A alocação é uma simples questão de pegar o número necessário de células dessa lista quando elas forem necessárias. A desalocação é um processo muito mais complexo. Discutiu-se o problema básico com a desalocação na Seção 6.10.4, juntamente com o procedimento *dispose* do Pascal. Pode-se apontar para uma variável dinâmica no monte por meio de mais de um ponteiro, tornando difícil determinar quando a variável não é mais útil para o programa. Simplesmente porque um ponteiro está desconectado de uma célula ela não se torna lixo; poderia haver diversos outros ponteiros que ainda estão apontando para a célula.

No LISP, diversas das operações mais freqüentes nos programas criam coleções de células não mais acessíveis e, portanto, devem ser desalocadas (colocadas novamente na lista de espaço disponível). Uma das metas de projeto fundamentais do LISP foi assegurar que a reivindicação de células não-usadas não seja uma tarefa do programador, mas, ao contrário, do sistema em tempo de execução. Isso deixou os implementadores LISP com a questão de projeto fundamental: quando a desalocação deve ser executada?

Há dois processos distintos e, de certo modo, opostos de se reivindicar o lixo: contadores de referência, cuja reivindicação incremental é feita quando células inacessíveis são criadas, e coleta de lixo, cuja reclamação somente ocorre quando a lista de espaço disponível fica vazia. Esses dois métodos, às vezes, são chamados **abordagem ansiosa** e **preguiçosa**, respectivamente.

O método **contador de referência** da reivindicação de armazenamento realiza sua meta mantendo um contador em cada célula, o que armazena o número de ponteiros que estão apontando atualmente para a célula. É incorporada uma verificação de valor zero à operação de decremento dos contadores de referência, que ocorre quando um ponteiro é desconectado da célula. Se o contador de referência atingir zero, significa que nenhum ponteiro de programa está apontando para a célula e, assim, virou lixo e pode ser retornado à lista de espaço disponível.

Existem três problemas distintos com tal método. Primeiro, se as células de armazenamento forem relativamente pequenas, o espaço necessário para o contador será significativo. Em segundo lugar, parte do tempo de execução é evidentemente exigido para manter os valores dele. Todas as vezes que um valor de ponteiro é mudado, a célula para a qual ele estava apontando precisa ter seu contador decrementado, e a célula para a qual ele está apontando agora deve ter seu valor incrementado. Em uma linguagem como o LISP, em que

quase todas as ações envolvem mudança de ponteiros, isso pode ser uma parte significativa do tempo total de execução de um programa. Obviamente, se as mudanças de ponteiro não forem muito freqüentes, isso evidentemente não é um problema. Em terceiro lugar, surgem complicações quando uma coleção de células são conectadas circularmente. O problema, aqui, é que cada célula da lista circular tem um valor de contador de referência igual a pelo menos 1, o que o impede de ser coletado e colocado de volta na lista de espaço disponível. Uma solução pode ser encontrada em Friedman e Wise (1979).

A principal alternativa para os contadores de referência é a chamada **coleta de lixo**. Com esse método, o sistema, em tempo de execução, aloca células de armazenamento conforme o solicitado e desconecta ponteiros daquelas quando necessário, sem levar em consideração a reivindicação de armazenamento (permitindo que o lixo se acumule), até haver alocado todas as células disponíveis. Nesse ponto, o processo de coleta é iniciado para retirar todo o lixo flutuante em torno do monte. Para facilitar o processo de coleta de lixo, toda célula do monte tem um bit ou campo indicador extra usado pelo algoritmo de coleta.

O processo de coleta consiste nessas três fases distintas. Primeiro, todas as células do monte têm seus indicadores definidos para indicar que são lixo. Essa é, obviamente, uma suposição correta somente para algumas das células. A segunda parte é a mais difícil. Todo ponteiro do programa é localizado no monte, e todas as células às quais se pode chegar são marcadas como não sendo lixo. Depois disso, a terceira fase é executada. Todas as células no monte que não foram especificamente marcadas como sendo usadas são retornadas à lista de espaço disponível.

Para ilustrar o sabor dos algoritmos usados para marcar as células atualmente em uso, oferecemos a seguinte versão simples de um algoritmo de marcação. Supomos que todas as variáveis dinâmicas no monte ou células do monte compõem-se de uma parte de informação, de uma parte para a marca, chamada `tag`, e de dois ponteiros chamados `llink` e `rlink`. Essas células são usadas para construir grafos dirigidos com um máximo de dois arcos que partem de qualquer vértice. O algoritmo de marcação atravessa todas as árvores que os grafos abrangem, marcando todas as células encontradas. À semelhança de outras travessias de grafos, o algoritmo de marcação usa a recursão.

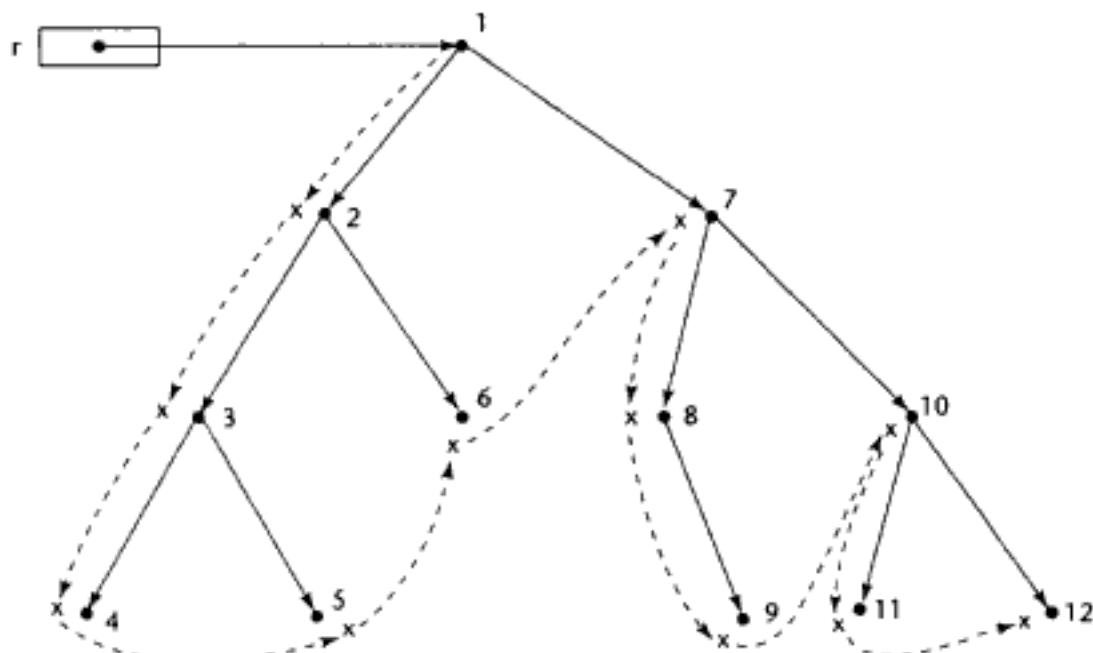
```

for every pointer r do
    mark(r)

procedure mark(ptr)
    if ptr <> null then
        if ptr^.tag is not marked then
            set ptr^.tag
            mark(ptr^.llink)
            mark(ptr^.rlink)
        end if
    end if

```

Um exemplo das ações desse procedimento em determinado grafo é mostrado na Figura 6.13 (ver p. 262). Tal algoritmo de marcação simples padece do problema de usar muito armazenamento (de espaço da pilha para suportar recursão). Um processo de marcação que não exige espaço adicional na pilha foi desenvolvido por Schorr e Waite (1967). O método reverte ponteiros enquanto localiza estruturas ligadas. Então, quando o final de uma lista é alcançado, o processo pode seguir os ponteiros retroativamente para fora da estrutura.



As linhas tracejadas mostram a ordem de marcação dos vértices.

**FIGURA 6.13** Um exemplo das ações do algoritmo de marcação.

O problema mais sério com a coleta de lixo pode ser resumido da seguinte maneira: quando você mais precisa, pior é seu funcionamento. Ela é mais necessária quando o programa realmente necessita da maioria das células do monte. Nessa situação, a coleta consome muito tempo, porque a maioria das células precisa ser rastreada e marcada como sendo útil. Entretanto, nesse caso, o processo rende somente um pequeno número de células que podem ser colocadas na lista de espaço disponível. Além desse problema, há o custo do espaço adicional das células marcadas, que precisa ser somente um bit, e o tempo de execução exigido para executar o processo de coleta. Porém, esses problemas não são tão sérios como parecem, por duas razões relacionadas. Primeiro, a memória é abundante na maioria dos computadores contemporâneos. Em segundo lugar, todos os computadores de maior capacidade usam memória virtual, o que faz suas grandes memórias parecerem muito maiores do que realmente são.

Tanto os algoritmos de marcação para a coleta de lixo como os processos exigidos pelo contador de referência podem ser transformados em mais eficientes por meio do uso das operações de rotação e de deslizamento de ponteiros descritas por Suzuki (1982).

**Células de Tamanho Variável.** O gerenciamento de um monte a partir do qual células de tamanho variável são alocadas tem todas as dificuldades daqueles de tamanho fixo mais alguns problemas adicionais. Infelizmente, células de tamanho variável são necessárias na maioria das linguagens de programação. Os problemas adicionais dependem do método utilizado. Se for o da coleta de lixo, os seguintes problemas adicionais ocorrem:

- A configuração inicial dos indicadores de todas as células do monte para indicar que elas são lixo é difícil. Uma vez que as células têm tamanhos diferentes, vascülhá-las torna-se um problema. Uma solução para isso é exigir que cada célula contenha o seu tamanho em seu primeiro campo. Então, o esquadriamento

poderá ser feito, ainda que consuma um pouco mais de espaço e bem mais tempo do que sua contraparte para aquelas de tamanho fixo.

- O processo de marcação não é banal. Como uma célula pode ser seguida a partir de um ponteiro se não houver nenhuma localização predefinida para o ponteiro na célula para a qual se aponta? As que não contêm ponteiros também são um problema. Adicionar um ponteiro de sistema a cada célula funcionará, mas ele deve ser mantido em paralelo com os definidos pelo usuário. Isso adiciona sobrecarga tanto de espaço como de tempo de execução ao custo para rodar o programa.
- A manutenção da lista de espaço disponível é outra fonte de sobrecarga. A lista pode iniciar-se com uma única célula consistindo em todo o espaço disponível. Pedidos de segmentos simplesmente reduzem o tamanho desse bloco. As células reivindicadas são adicionadas à lista. O problema é que não demora muito, e a lista torna-se longa com segmentos de vários tamanhos. Isso desacelera a alocação porque os pedidos fazem com que a lista seja pesquisada para ver se há blocos suficientemente grandes. Por fim, esta pode consistir em um número muito grande de blocos muito pequenos, que não são grandes o bastante para a maioria dos pedidos. Nesse ponto, blocos adjacentes podem aglutarinhar-se para formar blocos maiores. As alternativas para o uso do primeiro bloco suficientemente grande da lista podem abreviar a busca, mas exigem que a lista seja organizada por tamanho de blocos. Em qualquer um dos casos, manter a lista é uma sobrecarga adicional.

Se forem usados contadores de referência, os dois primeiros problemas serão evitados, mas o problema de manutenção da lista de espaço disponível persistirá.

## RESUMO

Os tipos de dados de uma linguagem são uma parte grande daquilo que determina o seu estilo e o seu uso. Juntamente 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 numérico, o caractere e o booleano. Os tipos numéricos freqüentemente são diretamente suportados por hardware.

Os tipos enumeração e a subfaixa definidos pelo usuário são convenientes e aumentam a legibilidade e a confiabilidade dos programas.

As matrizes fazem parte da maioria das linguagens de programação. A relação entre uma referência a um elemento de matriz e o endereço deste é dada em uma função de acesso que é uma implementação de um mapeamento biunívoco. As matrizes podem ser estáticas, como no FORTRAN 77; fixas dinâmicas na pilha, como nos procedimentos Pascal; dinâmicas na pilha, como nos blocos Ada; ou dinâmicas no monte, como nas matrizes ALLOCATABLE do FORTRAN 90. A maioria das linguagens permite somente algumas operações com matrizes completas.

Atualmente, os registros estão incluídos na maioria das linguagens. Os seus campos são especificados de várias maneiras. Nos casos do COBOL e da PL/I, eles podem ser referenciados sem nomear todos os registros envolventes, ainda que isso seja confuso de implementar e prejudicial à legibilidade. Em uma linguagem orientada a objeto, como o Java, os registros são suportados na construção de classe.

As uniões são localizações que podem armazenar diferentes valores de tipo em diferentes tempos. As uniões discriminadas incluem uma marca para registrar o valor de tipo atual. Uma união livre é aquela que não tem uma marca. A maioria das linguagens que tem uniões não possuem um projeto seguro para si mesmas, com exceção da Ada.

Os conjuntos, às vezes, são convenientes e relativamente fáceis de implementar. As aplicações que normalmente usam conjuntos, entretanto, podem ser feitas sem grandes dificuldades, por meio de outros tipos de dados.

Os ponteiros são usados para dar flexibilidade de endereçamento e para controlar o gerenciamento de armazenamento dinâmico. Eles representam alguns perigos inerentes: os ponteiros pendurados são difíceis de ser evitados e o lixo é difícil de ser coletado.

Os tipos referência, como por exemplo os do Java, oferecem gerenciamento do monte sem os perigos dos ponteiros.

O nível de dificuldade para implementar um tipo de dado tem uma forte influência com o modo como o tipo será incluído em uma linguagem. Os tipos enumeração, os tipos subfaixa e os tipos registro são todos relativamente fáceis de implementar. As matrizes também são diretas, ainda que o acesso a um elemento delas seja um processo dispendioso quando possui diversos subscritos. A função de acesso exige uma adição e uma multiplicação para cada subscrito.

Os ponteiros são custosos de implementar se forem usados para gerenciamento do armazenamento dinâmico e se forem tomadas providências para evitar que fiquem pendurados. O gerenciamento do monte é relativamente fácil se todas as células tiverem o mesmo tamanho, mas torna-se mais complicado com a alocação e desalocação de células de tamanho variável.

## *NOTAS BIBLIOGRÁFICAS*

Existe uma enorme quantidade de literatura na ciência da computação que se preocupa com o projeto, com o uso e com a implementação de tipos de dados. Hoare apresenta uma das primeiras definições sistemáticas dos tipos estruturados em Dahl et al. (1972). Tanenbaum compara o projeto de tipos do ALGOL 68 com o Pascal (Tanenbaum, 1978). Feuer e Gehani (1982) compararam o C e o Pascal, inclusive suas estruturas de tipos. Uma discussão a respeito das inseguranças do projeto de tipos de dados Pascal foi incluída em Welsh et al. (1977). Uma discussão geral sobre uma ampla variedade de tipos de dados é apresentada em Cleaveland (1986).

A implementação de verificações em tempo de execução sobre as possíveis inseguranças dos tipos de dados Pascal é discutida em Fischer e LeBlanc (1980). A maioria dos livros de projeto de compiladores, como por exemplo, Fischer e LeBlanc (1988) e Aho et al. (1986), descreve métodos de implementação para tipos de dados, a exemplo de outros textos sobre linguagens de programação, como Pratt (1984) e Ghezzi e Jazayeri (1987). Uma discussão detalhada dos problemas do gerenciamento do monte pode ser encontrada em Tanenbaum et al. (1990). Métodos de coleta de lixo são desenvolvidos por Schorr e Waite (1967) e Deutsch e Bobrow (1976). Uma discussão abrangente dos algoritmos de coleta de lixo pode ser encontrada em Cohen (1981).

## *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 relativas aos tipos cadeia de caracteres?
4. Defina as três opções de tamanho de cadeia.
5. Defina o que são tipos ordinal, enumeração e subfaixa.
6. Quais são as vantagens dos tipos enumeração, definidos pelo usuário?

7. Quais são as questões de projeto relativas às matrizes?
8. Defina o que são matrizes estáticas, fixas dinâmicas na pilha, dinâmicas na pilha e dinâmicas. Quais as vantagens de cada uma?
9. Qual recurso de inicialização de matrizes está disponível na Ada, mas não está disponível em outras linguagens imperativas comuns?
10. O que é uma constante agregada?
11. Quais operações de matrizes são oferecidas especificamente para matrizes unidimensionais na Ada?
12. Quais são as diferenças entre as fatias (*slices*) do FORTRAN 90 e as da Ada?
13. Defina ordem de linha maior e ordem de coluna maior.
14. O que é uma função de acesso para uma matriz?
15. Quais são as entradas exigidas em um descritor de matriz Pascal, e quando elas devem ser armazenadas (no tempo de compilação ou no de execução)?
16. Qual é o propósito dos números de nível nos registros COBOL?
17. Defina referências amplamente qualificadas e referências elípticas a campos em registros.
18. Defina união, união livre e união discriminada.
19. Quais são as questões de projeto relativas às uniões?
20. Quais são os dois problemas com as uniões do Pascal?
21. De que maneira as uniões da Ada são mais seguras do que as do Pascal?
22. Por que normalmente há severas restrições quanto ao tamanho dos conjuntos nas implementações Pascal?
23. Quais são as questões de projeto relativas aos tipos ponteiro?
24. Quais são os dois problemas comuns com os ponteiros?
25. Quais são as duas maneiras segundo as quais os ponteiros da Ada são mais seguros do que os do Pascal?
26. Por que os ponteiros da maioria das linguagens restringem-se a apontar para um objeto de tipo único?
27. O que é um tipo referência C++ e qual é seu uso comum?
28. Por que as variáveis de referência no C++ são melhores do que os ponteiros para parâmetros formais?
29. Quais vantagens as variáveis de tipo referência do Java têm sobre os ponteiros de outras linguagens?
30. Descreva as abordagens preguiçosa e ansiosa de reivindicar o lixo.
31. Quais são as diferenças entre as variáveis de referência do C++ e do Java?
32. Por que a aritmética em referências Java não faria sentido?

## PROBLEMAS

1. Quais são os argumentos favoráveis e contrários à representação dos valores booleanos como bits únicos na memória?
2. Por que um valor decimal gasta tanto espaço de memória?
3. O COBOL usa diversos métodos diferentes para armazenar números decimais. Explique o formato e o propósito de cada um.
4. Os microcomputadores VAX usam um formato para números de vírgula-flutuante que não é o mesmo que o padrão IEEE. Qual é esse formato e por que ele foi escolhido pelos projetistas dos computadores VAX? Uma referência para as representações do vírgula-flutuante VAX encontra-se em Sebesta (1991).
5. Compare os métodos das lápides e das chaves-e-fechaduras (*locks-and-keys*) para evitar ponteiros pendurados, do ponto de vista da segurança e do custo de implementação.
6. Projete um conjunto de programas de teste simples para determinar as regras de compatibilidade de tipos de um compilador Pascal ou C ao qual você tem acesso. Escreva um relatório de suas descobertas.
7. Quais desvantagens há no desreferenciamento implícito de ponteiros, mas somente em certos contextos? Por exemplo, considere o desreferenciamento implícito de um ponteiro para um registro em Ada quando ele é usado para referenciar um campo de registro.

8. Qual justificativa importante há para o operador  $\rightarrow$  no C e no C++?
  9. Determine qual opção de implementação descrita na Seção 6.9.4 é usada por algum compilador Pascal ao qual você tem acesso?
  10. As uniões no C e no C++ são separadas dos registros dessas linguagens, em vez de serem combinadas como são no Pascal e na Ada. Quais são as vantagens e as desvantagens dessas duas opções?
  11. Determine se algum compilador Pascal que você tem acesso implementa o procedimento `dispose`.
  12. Determine se algum compilador C ao qual você tem acesso implementa a função `free`.
  13. Suponhamos que uma linguagem inclua tipos-enumeração definidos pelo usuário e que valores de enumeração possam ser sobreescritos; ou seja, o mesmo valor literal poderia aparecer em dois tipos-enumeração diferentes, como em:  
`type`  
`cores = (vermelho, blue, verde);`  
`humor = (feliz, brabo, blue);`
- O uso da constante `blue`\* não pode ser verificado quanto ao tipo. Proponha um método para permitir essa verificação de tipos sem desativar completamente essa sobreescrita.
14. Matrizes multidimensionais podem ser armazenados em ordem de linha maior, como no Pascal, ou em ordem de coluna maior, como no FORTRAN. Desenvolva as funções de acesso para ambos os arranjos para matrizes tridimensionais.
  15. Na linguagem Burroughs Extended ALGOL, as matrizes são armazenadas como um vetor unidimensional de ponteiros para as linhas da matriz, que são tratados como vetores unidimensionais de valores. Quais são as vantagens e as desvantagens desse esquema?
  16. Escreva um programa que faz a multiplicação de matrizes em alguma linguagem que faça verificação da faixa de subscrito e para a qual você possa obter uma versão de linguagem de montagem ou de linguagem de máquina do compilador. Determine o número de instruções necessárias para a verificação da faixa de subscrito e compare-o com o número total de instruções para o processo de multiplicação de matrizes.
  17. Escreva um programa Pascal que inclua as seguintes declarações:  
`var`  
`A, B : array [1..10] of integer;`  
`C : array [1..10] of integer;`  
`D : array [1..10] of integer;`

Inclua no programa o código que determina, para cada vetor, qual dos outros três vetores são compatíveis com ele.

18. Se você tiver acesso a um compilador em que o usuário pode especificar se a verificação de faixa de subscrito é desejada, escreva um programa que faça um grande número de acessos a matrizes e cronometre a execução deles. Rode o programa com verificação de faixa de subscrito e sem ela e compare os tempos.
19. Analise e escreva uma comparação das funções `malloc` e `free` do C com os operadores `new` e `delete` do C++. Use a segurança como a principal consideração na comparação.
20. Analise e escreva uma comparação do uso de ponteiros C++ e variáveis de referência Java para referir-se a variáveis dinâmicas no monte. Use a segurança e a conveniência como as principais considerações na comparação.
21. Escreva uma breve discussão daquilo que foi ganho e daquilo que foi perdido na decisão dos projetistas do Java de não incluir os ponteiros do C++.
22. Quais são os argumentos favoráveis e os contrários à recuperação de armazenamento do monte implícito do Java, em comparação com a recuperação de armazenamento do monte explícito, exigido no C++?

\*N. de R. Blue: em inglês, pode significar tanto a cor azul como triste.

## Capítulo 7

# Expressões e Instruções de Atribuição



### Friedrich (Fritz) L. Bauer

Fritz Bauer reside em Munique e, juntamente com Klaus Samelson, do inicio até meados da década de 50, projetou uma linguagem algébrica que podia ser implementada diretamente em hardware. As mais significativas contribuições de Bauer para o projeto de linguagens, entretanto, aconteceram pelo fato de ele ser um dos principais membros da equipe de projeto do ALGOL.

- 7.1** Introdução
- 7.2** Expressões Aritméticas
- 7.3** Operadores Sobrecarregados
- 7.4** Conversões de Tipo
- 7.5** Expressões Relacionais e Booleanas
- 7.6** Avaliação Curto-Círcuito
- 7.7** Instruções de Atribuição
- 7.8** Atribuição de Modo Misto

Como o título indica, o foco deste capítulo são as expressões e as instruções de atribuição. As regras semânticas que determinam a ordem de avaliação dos operadores nas expressões serão discutidas primeiro. Depois, discutiremos os problemas potenciais da ordem de avaliação de operandos definida na implementação, quando as expressões podem ter efeitos colaterais. Os operadores sobrecarregados, tanto predefinidos como definidos pelo usuário, serão, então, abordados, juntamente com seus efeitos sobre as expressões em programas. Em seguida, as expressões de modo misto serão discutidas e avaliadas. Isso nos leva à definição e à avaliação das conversões de tipo de alargamento e de estreitamento, tanto implícitas como explícitas. As expressões relacionais e booleanas serão então discutidas, incluindo a idéia de avaliação curto-circuito.

Finalmente, cobriremos a instrução de atribuição, desde a forma mais simples a todas as suas variações, incluindo as atribuições como expressões e de modo misto.

O material deste capítulo restringe-se às linguagens de programação não-funcionais e não-lógicas convencionais, que usam a notação infixada para expressões aritméticas e lógicas. As questões referentes à especificação e à avaliação de expressões em linguagens funcionais e lógicas serão discutidas nos Capítulos 15 e 16, respectivamente.

As expressões de casamento de padrões com cadeias de caracteres foram discutidas como uma parte do material sobre cadeias no Capítulo 6, de forma que não serão mencionadas neste capítulo.

## 7.1 Introdução

As expressões são o meio fundamental de especificar computações em uma linguagem de programação. É crucial que o programador entenda tanto a sintaxe como a semântica das expressões. Métodos para descrever a sintaxe das expressões foram descritos no Capítulo 3. Neste capítulo, focalizaremos a semântica das expressões — ou seja, o que significam e o que é determinado pela maneira com que são avaliadas.

Para entender a avaliação de expressões é necessário estar familiarizado com as ordens de avaliação de operadores e de operandos. A ordem de avaliação de operadores de expressões é regida pelas regras de associatividade e de precedência da linguagem. Ainda que o valor de uma expressão, às vezes, dependa dela, a ordem de avaliação de operandos em expressões muitas vezes não é declarada pelos projetistas da linguagem, o que acarreta uma situação que permite aos programas produzirem diferentes resultados em diferentes implementações. Outras questões relativas à semântica das expressões são a não-coincidência de tipos, as coerções e a avaliação curto-circuito.

A essência das linguagens de programação imperativas é o papel predominante das instruções de atribuição cuja finalidade é mudar o valor de uma variável. Assim, uma parte integrante de todas as linguagens imperativas é o conceito de variáveis cujos valores mudam durante a execução do programa (as linguagens não-imperativas, às vezes, incluem variáveis de um tipo diferente, como, por exemplo, os parâmetros de funções em linguagens funcionais).

Uma instrução de atribuição pode simplesmente fazer com que um valor seja copiado de uma célula de memória para outra. Mas, em muitos casos, as instruções de atribuição incluem expressões com operadores, o que faz com que os valores sejam copiados para o processador e processados e o resultado copiado novamente para a memória.

As instruções de atribuição simples especificam que uma expressão deve ser avaliada e que haja uma localização alvo para colocar o resultado da avaliação da expressão. Conforme veremos neste capítulo, há um grande número de variações dessa forma básica.

## 7.2 Expressões Aritméticas

A avaliação automática de expressões aritméticas semelhantes àquelas encontradas na Matemática foi uma das principais metas das primeiras linguagens de programação de alto nível. A maioria das características das expressões aritméticas nas linguagens de programação foi herdada de convenções que se desenvolveram na Matemática. Nas linguagens de programação, as expressões aritméticas consistem em operadores, operandos, parênteses e chamadas a função. Os operadores podem ser **unários**, significando que têm um único operando, ou **binários**, significando que têm dois operandos. O C, o C++ e o Java incluem um operador **ternário**, o qual tem três operandos, conforme discutiremos na Seção 7.2.1.4.

Na maioria das linguagens de programação imperativas, os operadores binários são infixados, o que significa que eles aparecem entre seus operandos. Uma exceção é a Perl, que tem alguns operadores pré-fixados, o que significa que eles precedem seus operandos.

A finalidade de uma expressão aritmética é especificar uma computação aritmética. Uma implementação desta deve causar duas ações: buscar os operandos e executar as operações aritméticas sobre eles. Nas seções seguintes, investigaremos os detalhes de projeto comuns das expressões aritméticas nas linguagens imperativas.

A seguir, apresentamos as principais questões de projeto referentes às expressões aritméticas, todas as quais serão 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 de operandos?
- Há restrições quanto aos efeitos colaterais da avaliação de operandos?
- A linguagem permite sobrecarga de operadores definida pelo usuário?
- Qual mesclagem de modos é permitida nas expressões?

### 7.2.1 Ordem de Avaliação de Operadores

Investigaremos, primeiramente, as regras de linguagem que especificam a ordem de avaliação de operadores.

#### 7.2.1.1 Precedência

O valor de uma expressão depende, pelo menos em parte, da ordem de avaliação dos operadores na expressão. Consideremos a seguinte expressão:

$$a + b * c$$

Suponhamos que as variáveis A, B e C tenham os valores 3, 4 e 5, respectivamente. Se for avaliada da esquerda para a direita (a adição primeiro e depois a multiplicação), o resultado será 35. Se for avaliada da direita para a esquerda, o resultado será 23.

Em vez de simplesmente avaliar a ordem da esquerda para a direita ou da direita para a esquerda, os matemáticos desenvolveram o conceito de colocar operadores em uma hierarquia de prioridades de avaliação e basear a sua ordem parcialmente nessa hierarquia. Por exemplo, em Matemática, a multiplicação é considerada de maior prioridade do que a adição. Se seguissemos essa convenção em nosso exemplo de expressão, a multiplicação seria avaliada primeiro.

As regras de **precedência de operadores** para avaliação de expressões definem a ordem em que os operadores de diferentes níveis de precedência são avaliados. As regras de precedência de operadores referentes a expressões baseiam-se na hierarquia de prioridades do operador, conforme a visão do projetista da linguagem. As regras de precedência de operadores comuns são quase todas iguais, uma vez que todas se baseiam nas regras da Matemática. Nessas linguagens, a exponenciação tem a precedência mais elevada (quando é oferecida pela linguagem), seguida da multiplicação e da divisão no mesmo nível, e da adição e da subtração binárias no mesmo nível.

Muitas linguagens também incluem versões unárias da adição e da subtração. As primeiras são chamadas de **operador de identidade** porque normalmente elas não têm nenhuma operação associada e, dessa forma, não oferecem efeito sobre seu operando. Ellis e Stroustrup, falando sobre o C++, chamam-na de acidente histórico e rotulam-na corretamente de inútil (Ellis e Stroustrup, 1990, p. 56). No Java, o mais (+) unário tem realmente um efeito quando seu operando é **char**, **short** ou **byte** — ele provoca uma conversão implícita desse operando para o tipo **int**. O menos (-) unário, é claro, sempre muda o sinal de seu operando.

Em todas as linguagens imperativas comuns, o operador unário menos (-) pode aparecer em uma expressão no início, ou em qualquer lugar dentro dela, contanto que esteja entre parênteses para impedir que ele fique adjacente a outro operador. Por exemplo,

**A + (- B) \* C**

é legal, mas

**A + - B \* C**

normalmente não é.

Conforme veremos na Seção 7.2.1.2, a precedência dos operadores unários raramente é relevante.

As precedências dos operadores aritméticos de algumas linguagens de programação comuns são as seguintes:

	FORTRAN	Pascal	C	Ada
Mais alta	<b>**</b>	<b>*, /, div, mod</b>	<b>++, --</b> pós-fixo	<b>**, abs</b>
	<b>*, /</b>	todos <b>+, -</b>	<b>++, --</b> prefixo	<b>*, /, mod</b>
	todos <b>+, -</b>		<b>+,-</b> unário	<b>+,-</b> unário
			<b>*, /, %</b>	<b>*, -</b> binário
Mais baixa			<b>+, -</b> binário	

O operador **\*\*** é a exponenciação. Os operadores **/** e **div** do Pascal serão descritos na Seção 7.3. O operador **%** do C é exatamente como o operador **mod** do Pascal e da Ada: ele toma dois operandos inteiros e produz o resto da divisão do primeiro pelo segundo. Os operadores **++** e **--** do C serão descritos na Seção 7.7.5. As regras de precedência do C++ e do C são as mesmas, exceto que no C++, todos os operadores **++** e **--** têm precedência igual. As regras de precedência do Java são as mesmas do C++. O operador **abs** da Ada é um operador unário que produz o valor absoluto de seu operando.

A APL é ímpar entre as linguagens porque tem um único nível de precedência, como ilustra a próxima seção.

A precedência é responsável somente por algumas das regras referentes à ordem de avaliação de operadores; as regras de associatividade também a afeta.

### 7.2.1.2 Associatividade

Considere a seguinte expressão:

$$a - b + c - d$$

Se os operadores de adição e de subtração tiverem o mesmo nível de precedência, as regras de precedência nada dizem sobre a ordem de avaliação dos operadores nessa expressão.

Quando uma expressão contém duas ocorrências adjacentes de operadores com o mesmo nível de precedência, a questão sobre qual deles é avaliado primeiro responde-se pelas regras de **associatividade** da linguagem. Um operador pode ter associatividade à esquerda ou à direita, significando que a primeira ocorrência será avaliada antes ou a ocorrência mais à direita será avaliada primeiro, respectivamente.

A associatividade nas linguagens imperativas comuns ocorre da esquerda para a direita, exceto o operador de exponenciação (quando fornecido), que associa da direita para a esquerda. Na expressão FORTRAN

$$A - B + C$$

o operador esquerdo é avaliado primeiro. Mas a exponenciação, no FORTRAN, é associativa à direita; sendo assim, na expressão

$$A ** B ** C$$

o operador direito é avaliado primeiro.

Na Ada, a exponenciação é não-associativa, o que significa que a expressão

$$A ** B ** C$$

é ilegal. Uma expressão desse tipo deve ser colocada entre parênteses para mostrar a ordem desejada, como em

$$(A ** B) ** C$$

ou

$$A ** (B ** C)$$

Agora podemos explicar porque a precedência de operadores unários, muito freqüentemente, não é importante. Os operadores menos (-) unários e binários FORTRAN têm a mesma precedência, mas, na Ada (e na maioria das outras linguagens comuns), o menos unário tem precedência sobre o menos binário. Porém, considere a expressão

$$- A - B$$

Uma vez que o FORTRAN usa a associatividade tanto para o menos unário como para o menos binário, e em razão da Ada dar precedência ao primeiro sobre o segundo, essa expressão equivale a

$$(- A) - B$$

em ambas as linguagens. A seguir, considere as seguintes expressões:

- $A / B$
- $A * B$
- $A ** B$

Nos dois primeiros casos, a precedência relativa do operador unário menos e do operador binário é irrelevante — qual deles é executado primeiro não tem nenhum efeito sobre o valor da expressão. No último caso, entretanto, isso importa. Das linguagens de programação comuns, somente o FORTRAN e a Ada têm o operador de exponenciação. Em ambos os casos, a exponenciação tem maior precedência do que o menos unário; assim,

- $A ** B$

é equivalente a

- $(A ** B)$

Quando os operadores unários aparecem em posições que não a extremidade esquerda das expressões, eles devem ser colocados entre parênteses em ambas as linguagens, de modo que, nessas situações, eles sejam forçados a ter a maior precedência (os parênteses serão discutidos na Seção 7.2.1.3).

As regras de associatividade de algumas das linguagens imperativas mais comuns são apresentadas abaixo:

Linguagem	Regra de Associatividade
FORTRAN	Esquerda: *, /, +, - Direita: **
Pascal	Esquerda: Todos
C	Esquerda: ++ pós-fixo, -- pós-fixo, *, /, %, + binário, - binário Direita: ++ prefixo, -- prefixo, + unário, - unário
C++	Esquerda: *, /, %, + binário, - binário Direita: ++, --, - unário, + unário
Ada	Esquerda: todos, exceto ** não-associativo: **

Conforme afirmamos na Seção 7.2.1.1, na APL, todos os operadores têm o mesmo nível de precedência. Assim, a ordem de avaliação de operadores em suas expressões é determinada inteiramente pela regra de associatividade, da direita para a esquerda para todos os operadores. Por exemplo, na expressão

$$A * B + C$$

o operador de adição é avaliado primeiro, seguido pelo operador de multiplicação ( $*$  é o operador de multiplicação APL). Se  $A$  fosse 3,  $B$  fosse 4 e  $C$  fosse 5, o valor dessa expressão APL seria 27.

Muitos compiladores usam o fato de que alguns operadores aritméticos são matematicamente associativos, significando que as regras de associatividade não exercem nenhum impacto sobre o valor de uma expressão que contenha somente esses operadores. Por exemplo, a adição é matematicamente associativa, de modo que, em Matemática, o valor da expressão

$$A + B + C$$

não depende da ordem de avaliação do operador. Se as operações de números reais para operações matematicamente associativas também fossem associativas, o compilador poderia usar esse fato para executar algumas otimizações simples. Especificamente, se for permitido ao compilador reorganizar a avaliação de operadores, talvez ele seja capaz de produzir um código ligeiramente mais rápido para avaliação de expressões. Os compiladores fazem, de fato, esses tipos de otimizações.

Infelizmente, tanto as representações de números reais como as operações aritméticas com eles são somente aproximações da Matemática (devido a limitações de tamanho). O fato de o operador matemático ser associativo não implica necessariamente que a operação com números reais correspondente seja associativa. De fato, somente se todos os operandos e os resultados intermediários puderem ser representados exatamente em notação de vírgula-flutuante é que o processo será precisamente associativo. Por exemplo, há situações patológicas em que a adição de números inteiros em um computador não é associativa. Por exemplo, suponhamos que um programa deva avaliar a expressão

$$A + B + C + D$$

e que  $A$  e  $C$  sejam números positivos muito grandes,  $B$  e  $D$  números negativos com valores absolutos muito grandes. Nessa situação, adicionar  $B$  a  $A$  não causará um estouro (*overflow*), mas adicionar  $C$  a  $A$ , sim. Igualmente, adicionar  $C$  a  $B$  não causará um estouro, mas adicionar  $B$  a  $D$ , sim. Devido a essas limitações da aritmética computadorizada, a adição é catastroficamente não-associativa nesse caso. Portanto, se o compilador reorganizar as operações de adição, ele afetará o valor da expressão. Evidentemente, este é um problema que pode ser evitado pelo programador, supondo que os valores aproximados das variáveis sejam conhecidos. O programador pode simplesmente colocar entre parênteses a expressão (veja a Seção 7.2.1.3) para garantir que somente a ordem de avaliação segura será possível. Porém, tal situação pode surgir de uma maneira bem mais sutil, em que o programador tem menos probabilidade de notar a dependência da ordem.

### 7.2.1.3 Parênteses

Os programadores podem alterar as regras de precedência e de associatividade colocando parênteses nas expressões. Uma parte de uma expressão assim configurada tem precedência sobre suas partes adjacentes não colocadas entre parênteses. Por exemplo, mesmo a multiplicação tendo precedência sobre a adição, na expressão

$$(A + B) * C$$

a adição será avaliada primeiro. Matematicamente, isso é perfeitamente natural. Nessa expressão, o primeiro operando do operador de multiplicação não estará disponível até que a subexpressão entre parênteses seja avaliada.

As linguagens que permitem parênteses em expressões aritméticas poderiam dispensar todas as regras de precedência e simplesmente associar todos os operadores da esquerda para a direita ou vice-versa. O programador especificaria a ordem de avaliação desejada com parênteses. Isso simplificaria, pois nem o autor, nem os leitores dos programas precisariam lembrar-se de quaisquer regras de precedência ou de associatividade. A desvantagem desse esquema é que ele torna a escrita de expressões mais tediosa e também compromete seriamente a legibilidade do código. Contudo, essa foi a opção feita por Ken Iverson, o projetista da APL.

### 7.2.1.4 Expressões Condicionais

Enfocaremos o operador ternário, `? :`, que faz parte do C, do C++ e do Java. Este operador é usado para formar expressões condicionais.

Às vezes, instruções `if-then-else` são usadas para executar uma atribuição de expressão condicional. Por exemplo, considere:

```
if (cont == 0)
    then media = 0;
    else media = soma / cont;
```

No C, no C++ e no Java, isso pode ser especificado mais convenientemente nas instruções de atribuição usando-se uma expressão condicional, que tem a forma

```
expressão_1 ? expressão_2 : expressão_3
```

em que `expressão_1` é interpretada como uma expressão booleana. Se `expressão_1` for avaliada como verdadeira, o valor da expressão inteira será o valor da `expressão_2`; caso contrário, será o valor da `expressão_3`. Por exemplo, o efeito da `if-then-else` acima pode ser obtido com a seguinte instrução de atribuição, usando-se uma expressão condicional:

```
media = (cont == 0) ? 0 : soma / cont;
```

Com efeito, o ponto de interrogação denota o início da cláusula `then`, e o sinal de dois-pontos marca o inicio da cláusula `else`. Ambas são obrigatórias. Note que `?` é usado em expressões condicionais como um operador ternário.

Expressões condicionais podem ser usadas em qualquer lugar em um programa C, C++ ou Java em que qualquer outra expressão possa ser usada.

## 7.2.2 Ordem de Avaliação de Operandos

Uma característica de projeto de expressões menos comumente discutida é a ordem de avaliação de operandos. As variáveis, nas expressões, são avaliadas buscando seus valores na memória. As constantes, às vezes, são avaliadas da mesma maneira. Em outros casos, uma constante pode fazer parte da instrução em linguagem de máquina e não exigir uma busca na memória. Se um operando for uma expressão colocada entre parênteses, todos os operadores que ela contém devem ser avaliados antes que seu valor possa ser usado como um operando.

Se nenhum dos operandos de um operador tiver efeitos colaterais, a sua ordem de avaliação será irrelevante. Portanto, 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 **efeito colateral funcional**, ocorre quando ela modifica um de seus parâmetros ou uma variável global (ela é declarada fora da função, mas acessível na função).

Considere a expressão

```
a + fun(a)
```

Se `fun` não tiver o efeito colateral de modificar `a`, a ordem de avaliação dos dois operandos, `a` e `fun(a)`, não terá nenhum efeito sobre o valor da expressão. Porém, se `fun` modi-

ficar a, haverá um efeito. Considere a seguinte situação: fun retorna o valor de seu argumento dividido por 2 e modifica seu parâmetro para que tenha o valor 20. Suponhamos o seguinte:

```
a = 10;  
b = a + fun(a)
```

Então, se o valor de a for buscado primeiro (no processo de avaliação da expressão), seu valor será 10 e o valor da expressão será 15. Mas, se o segundo operando for avaliado primeiro, o valor do primeiro será 20, enquanto que o da expressão será 25.

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;  
} /* fun1 */  
void fun2 () {  
    a = a + fun1();  
} /* fun2 */  
void main () {  
    fun2 ();  
} /* main */
```

O valor computado para a em fun2 depende da ordem de avaliação dos operandos na expressão a + fun1(). O valor de a será 8 ou 20.

Há duas soluções para o problema da ordem de avaliação de operandos. Primeiro, o projetista da linguagem poderia impedir que a avaliação da função afetasse o valor das expressões simplesmente rejeitando os efeitos colaterais funcionais. O segundo método para evitar o problema é declarar, na definição da linguagem, que os operandos das expressões devem ser avaliados em uma ordem particular e exigir que os implementadores garantam esta ordem.

Rejeitar os efeitos colaterais funcionais é difícil e elimina um pouco da flexibilidade para o programador. Considere o caso do C e do C++, que têm somente funções. Para eliminar os efeitos colaterais de dois parâmetros bidirecionais e, ainda assim, oferecer subprogramas que retornam mais de um valor, seria necessário um novo tipo de subprograma semelhante aos procedimentos das outras linguagens imperativas. O acesso a globais em funções também teria de ser rejeitado. Porém, quando a eficiência é importante, usar o acesso a variáveis globais para evitar a passagem de parâmetros é um método importante de aumento da velocidade de execução. Nos compiladores, por exemplo, o acesso global a dados, como a tabela de símbolos, é comum.

O problema de ter-se somente uma ordem de avaliação estrita é que algumas técnicas de otimização de código usadas pelos compiladores envolvem reorganizar as avaliações de operandos. Uma ordem garantida rejeita esses métodos quando chamadas a funções estão envolvidas. Não há, portanto, nenhuma solução perfeita, conforme é sustentado pelos projetos de linguagem reais.

Os projetistas do FORTRAN 77 encontraram uma terceira solução. A definição do FORTRAN 77 afirma que expressões com chamadas a funções são legais somente se estas não modificarem os valores dos outros operandos na expressão. Infelizmente, não é fácil para o compilador determinar o efeito exato que uma função pode ter sobre variáveis fora

dela, especialmente na presença de variáveis globais fornecidas por COMMON e pelos apelidos fornecidos por EQUIVALENCE. Este é o caso cuja definição da linguagem especifica as condições sob as quais uma construção é legal, mas deixa para o programador a tarefa de assegurar que essas construções sejam legalmente especificadas nos programas.

O Pascal e a Ada permitem que os operandos de operadores binários sejam avaliados em qualquer ordem escolhida pelo implementador. Além disso, funções nessas linguagens podem ter efeitos colaterais, de modo que podem ocorrer os problemas discutidos acima. Os efeitos colaterais funcionais serão discutidos adicionalmente no Capítulo 9.

A definição da linguagem Java garante que os operandos sejam avaliados na ordem da esquerda para a direita, eliminando os problemas discutidos nesta seção.

### 7.3 Operadores Sobrecarregados

Os operadores aritméticos, muitas vezes, são usados para mais de uma finalidade. Por exemplo, + é usado freqüentemente para a adição de quaisquer operandos de tipo numérico. Algumas linguagens, o Java por exemplo, também o usa para concatenação de cadeias. Seu emprego múltiplo é chamado de **sobrecarga de operador** e geralmente é considerado aceitável, contanto que a legibilidade e/ou a confiabilidade não sofram. Alguns acham que há demasiada sobrecarga de operador na APL e na SNOBOL, nas quais se usa a maioria dos operadores tanto para operações unárias como para binárias.

Como um exemplo dos possíveis perigos da sobrecarga, considere o uso do E comercial (&) no C. Como operador binário, ele especifica uma operação AND lógica bit a bit\*. Como operador unário, entretanto, seu significado é totalmente diferente. Como operador unário com uma variável como o seu operando, o valor da expressão é o endereço da variável. Nesse caso, o E comercial é chamado de operador de endereço. Por exemplo, a execução de

```
x = &y;
```

faz com que o endereço de y seja colocado em x. Há dois problemas com esse uso múltiplo do E comercial. Primeiro, usar o mesmo símbolo para duas operações completamente sem relação uma com a outra é prejudicial para a legibilidade. Segundo, o simples erro de digitação de deixar fora o primeiro operando para uma operação AND bit a bit pode não ser detectado pelo compilador, porque ele será interpretado como um operador de endereço. Esse tipo de erro pode ser de difícil diagnóstico.

Virtualmente, todas as linguagens de programação têm um problema menos sério, mas similar, que freqüentemente se deve à sobrecarga do operador menos. O problema consiste somente no fato do compilador não saber dizer se o operador pretende ser binário ou unário. Sendo assim, mais uma vez, deixar de incluir o primeiro operando quando o operador pretende ser binário não pode ser detectado como um erro pelo compilador. Pôrém, os significados das duas operações, unária e binária, estão estreitamente relacionados, de modo que a legibilidade não é afetada.

Os símbolos de operador distintos não somente aumentam a legibilidade, mas, às vezes, são convenientes também para serem usados por operações comuns. O operador de

\*N. de T. Bit a bit: lidar com bits em vez de lidar com estruturas maiores, como um byte. Operadores bit a bit são comandos ou instruções de programação que funcionam com bits individuais.

divisão é um exemplo. Considere o problema de encontrar a média, um número real de uma lista de números inteiros. Normalmente, a soma deles é computada como um número inteiro. Suponhamos que isso tenha sido feito na variável `soma`, e que o número de valores esteja em `cont`. Agora, se a média, um número real, precisar ser computada e colocada na sua variável `media`, essa computação poderia ser especificada em C++ como

```
media = soma / cont;
```

Mas essa atribuição produz um resultado incorreto na maioria dos casos. Uma vez que ambos os operandos do operador de divisão sejam do tipo inteiro, uma operação de divisão de inteiros irá se desenvolver, deixando o resultado truncado para um número inteiro. Então, apesar do fato do destino (`media`) ser do tipo real, seu valor, a partir dessa atribuição, não pode ter uma parte fracionária. O resultado inteiro da divisão é convertido para real para a atribuição depois do truncamento da divisão de inteiros.

Quando um símbolo de operador distinto para divisão com números reais está disponível, a situação é simplificada. Por exemplo, no Pascal, em que `/` significa divisão com números reais, a seguinte atribuição pode ser usada:

```
media := soma / cont
```

em que `media` é um tipo real, enquanto `soma` e `cont` são do tipo inteiro. Ambos os operandos serão implicitamente convertidos para real, e uma operação com ele será usada. Tal tipo de operação de conversão implícita será discutido adicionalmente na seção seguinte. A divisão de inteiros no Pascal é especificada pelo operador `div`, que pega operandos inteiros e produz um resultado inteiro. Quando nenhum operador distinto para divisão com números reais é fornecido, conversões explícitas devem ser usadas. Elas serão discutidas na Seção 7.4.2.

Algumas linguagens que suportam tipos de dados abstratos (veja o Capítulo 11), por exemplo, a Ada, o C++ e o FORTRAN 90, permitem que o programador sobrecarregue ainda mais os símbolos de operador. Por exemplo, suponhamos que um usuário queira definir o operador `*` entre um inteiro escalar e uma matriz de inteiros para dar a entender que cada elemento da matriz deve ser multiplicado pelo escalar. Isso poderia ser feito escrevendo-se um subprograma de função chamado `*` para executar essa nova operação. O compilador escolherá o significado correto quando um operador sobrecarregado for especificado, baseando-se nos tipos dos operandos, como acontece com operadores sobrecarregados definidos pela linguagem. Por exemplo, se essa nova definição para `*` for definida em um programa Ada, seu compilador usará a nova definição para `*` sempre que o operador `*` apareça com um número inteiro simples como o operando esquerdo e com uma matriz de números inteiros como o direito.

Quando criteriosamente usada, a sobrecarga de operador definida pelo usuário pode auxiliar a legibilidade. Por exemplo, se `+` e `*` forem sobrecarregados para um tipo de dados matriz abstrata e `A`, `B` e `D` forem variáveis desse tipo,

`A * B + C * D`

pode ser usado em vez de

```
MatrizSoma(MatrizMult(A, B), MatrizMult(C, D))
```

Por outro lado, a sobrecarga definida pelo usuário pode ser danosa à legibilidade. Em primeiro lugar, nada impede que o usuário defina `+` para significar multiplicação. Além disso, ao ver um operador `*` em um programa, o leitor precisa descobrir tanto os tipos dos operandos como a definição do operador para determinar seus significados. Qualquer uma ou todas essas definições poderiam estar em outros arquivos.

O C++ tem alguns operadores que não podem ser sobre carregados. Entre eles, estão o operador de classe ou de membro de estrutura (.) e o operador de resolução de escopo (::). Curiosamente, a sobre carga de operador foi um dos recursos do C++ não copiado para o Java.

## 7.4 Conversões de Tipo

As conversões de tipo são de estreitamento ou de alargamento. Uma **conversão de estreitamento** transforma um valor para um tipo que não pode armazenar nem mesmo aproximações de todos os valores do original. Um exemplo é a conversão de uma **double** para uma **float** em C (a faixa de **double** é muito maior do que a de **float**). Uma **conversão de alargamento** muda um valor para um tipo que pode incluir, pelo menos, aproximações de todos os valores do original; por exemplo, convertendo uma **int** para uma **float** em C. As conversões de alargamento são quase sempre seguras, ao passo que as de estreitamento não.

Como exemplo de um problema potencial com uma conversão de alargamento, considere que, em muitas implementações de linguagem, ainda que as conversões de inteiros para reais sejam conversões de alargamento, alguma precisão pode ser perdida. Por exemplo, em algumas implementações, números inteiros são armazenados em 32 bits, o que permite, pelo menos, nove dígitos decimais de precisão. Mas, em muitos casos, valores de vírgula-flutuante também são armazenados em 32 bits, com somente sete dígitos decimais de precisão. Assim, o alargamento de números inteiros para reais podem resultar na perda de dois dígitos de precisão.

As conversões de tipo podem ser explícitas ou implícitas, que serão discutidas nas duas subseções seguintes.

### 7.4.1 Coerção em Expressões

Uma das decisões de projeto referentes às expressões aritméticas baseia-se na possibilidade de um operador ter operandos de tipos diferentes. As linguagens que permitem essas expressões, chamadas de **expressões de modo misto**, devem definir convenções para as conversões de tipo de operando implícitas, porque os computadores usualmente não têm operações binárias que levam operandos de tipos diferentes. Lembre-se de que, no Capítulo 5, definimos a coerção como uma conversão de tipo implícita iniciada pelo compilador. Referimo-nos às conversões de tipo explicitamente solicitadas pelo programador como conversões explícitas, ou casts, não coerções.

Não obstante alguns símbolos de operadores poderem ser sobre carregados, presumimos que um sistema de computador, em hardware ou em algum nível de simulação de software, tenha uma operação para cada tipo de operando e de operador definidos na linguagem. Para operadores sobre carregados em uma linguagem que usa vinculação de tipos estática, o compilador escolhe o tipo correto de operação baseando-se nos tipos dos operandos. Quando os dois operandos de um operador não são do mesmo tipo e isso é legal na linguagem, o compilador deve escolher um deles para ser coagido e produzir o código

para essa coerção. Na discussão seguinte, examinaremos as opções de projeto de coerção de diversas linguagens comuns.

Os projetistas de linguagens não chegaram a um acordo sobre a questão da coerção em expressões aritméticas. Os contrários a uma faixa ampla de coerções estão preocupados com os problemas de legibilidade porque elas eliminam os benefícios da verificação de tipos. Os que preferiram incluir todas as coerções estão mais preocupados com a perda de flexibilidade resultante das restrições. A questão é se os programadores devem preocupar-se com essa categoria de erros ou se o compilador é que deve detectá-los.

Como uma ilustração simples do problema, considere o seguinte método esquemático Java:

```
void meuMetodo() {
    int a, b, c;
    float d;
    ...
    a = b * d;
    ...
}
```

Suponhamos que o segundo operando do operador de multiplicação tivesse que ser **c**, mas, devido a um erro de digitação, tenha ficado como **d**. Uma vez que as expressões de modo misto são legais em Java, o compilador não detectaria isso como um erro. Ele simplesmente inseriria o código para converter o valor do outro operando **b** para **float**. Se expressões de modo misto não fossem legais em Java, esse erro de digitação teria sido detectado pelo compilador como um erro de tipo.

Como um exemplo mais extremo dos perigos e dos custos da demasiada coerção, considere os esforços da PL/I para conseguir flexibilidade nas expressões. Nela, uma variável do tipo cadeia de caracteres pode ser combinada com um número inteiro em uma expressão. Durante a execução, busca-se um valor numérico, vasculhando a cadeia. Se acontecer de o valor conter um ponto decimal, ele será presumido como do tipo real; o outro operando será coagido também para real e a operação resultante será do tipo real. A política de coerção é muito dispendiosa porque tanto a verificação como a conversão de tipos devem ser feitas em tempo de execução. Ela também elimina a possibilidade de detectar erros do programador nas expressões, porque um operador binário pode combinar um operando de qualquer tipo com outro de virtualmente qualquer tipo diferente.

A Ada permite muito poucos operandos de modo misto em expressões, uma vez que a detecção de erros é reduzida quando são permitidas expressões de modo misto. Ela não permite mesclar operandos inteiros e reais em uma expressão, com uma exceção: o operador de exponenciação, **\*\***, pode tomar um tipo real ou um inteiro como primeiro operando e um inteiro para o seu segundo. A Ada permite alguns outros tipos de mesclagem de tipos, usualmente relacionados aos tipos subfaixa.

Na maioria das outras linguagens comuns, não há nenhuma restrição para expressões aritméticas de modo misto.

O C++ e o Java têm tipos inteiros menores do que o **int**. No C++, eles são **char** e **short int**; no Java, eles são **byte**, **short** e **char**. Tais operandos são coagidos para **int** quando virtualmente qualquer operador seja aplicado a eles. Assim, embora dados possam ser armazenados em variáveis desses tipos, eles não podem ser manipulados antes da conversão para outro maior. Por exemplo, considere o seguinte código Java:

```
byte a, b, c;
...
a = b + c;
```

Os valores de `b` e `c` são coagidos para `int`, e uma adição inteira é executada. Depois, a soma é convertida para `byte` e colocada em `a`.

### 7.4.2 Conversão de Tipo Explícita

A maioria das linguagens oferece alguma capacidade para fazer conversões explícitas, tanto de alargamento como 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 convertido.

A Ada oferece conversão explícita com a sintaxe de chamada a função. Por exemplo, em uma conversão explícita, podemos ter

```
MEDIA := FLOAT(SOMA) / FLOAT(CONT)
```

em que `MEDIA` é do tipo real, `SOMA` e `CONT` podem ser qualquer tipo numérico.

Nas linguagens baseadas no C, as conversões de tipo explícitas são chamadas **`casts`**. A sintaxe de um `cast` não é a de uma chamada a função; ao contrário, o tipo desejado é colocado entre parênteses imediatamente antes da expressão a ser convertida, como é mostrado em

```
(int) angulo
```

Uma das razões para os parênteses nas conversões do C é que o C tem diversos nomes de tipo de duas palavras, como, por exemplo, `long int`.

### 7.4.3 Erros em Expressões

Um grande número de erros pode ocorrer na avaliação de expressões. Se a linguagem exigir verificação de tipos, não poderão ocorrer erros de tipo de operando. Já discutimos a respeito dos erros que podem ocorrer devido a coerções de operandos em expressões. Os outros devem-se a limitações da aritmética do computador e às inerentes à aritmética. O erro mais comum é criado quando o resultado de uma operação não pode ser representado na célula de memória onde ele deve estar armazenado. Isso é chamado *overflow*<sup>7</sup> ou *underflow*, dependendo se o resultado foi muito grande ou muito pequeno. Uma limitação da aritmética é que a divisão por zero é rejeitada. Obviamente, o fato dela não ser matematicamente permitida não impede que um programa tente fazê-la.

O *overflow* e o *underflow* de números reais e a divisão por zero são exemplos de erros de execução, às vezes, chamados de exceções. As maneiras pelas quais os projetistas de linguagens podem lidar com exceções serão discutidas no Capítulo 14.

<sup>7</sup>N. de T. *Overflow*: estouro do registrador. Uma condição de erro que ocorre quando o resultado de um cálculo é maior do que a maior quantidade que o computador pode armazenar.

## 7.5 Expressões Relacionais e Booleanas

Além das expressões aritméticas, as linguagens de programação têm expressões relacionais e booleanas ou lógicas.

### 7.5.1 Expressões Relacionais

Um **operador relacional** compara os valores de seus dois operandos. Uma **expressão relacional** tem dois operandos e um operador relacional. O valor da expressão relacional é booleano, exceto quando este último não é um tipo na linguagem. Os operadores relacionais normalmente são sobreescritos para uma variedade de tipos. A operação que determina a verdade ou a falsidade de uma expressão relacional depende dos tipos de operando. Ela pode ser simples, para inteiros, ou complexa, para cadeias de caracteres. Normalmente, os tipos dos operandos que podem ser usados para operadores relacionais são numéricos, cadeias e ordinais.

A sintaxe dos operadores relacionais disponíveis em algumas linguagens comuns é a seguinte:

Operação	Ada	Java	FORTRAN 90
Igual	=	==	.EQ. ou ==
Diferente	/=	!=	.NE. ou <>
Maior que	>	>	.GT. ou >
Menor que	<	<	.LT. ou <
Maior que ou igual	>=	>=	.GE. ou >=
Menor que ou igual	<=	<=	.LE. ou <=

Os projetistas do FORTRAN I usaram abreviações em inglês porque os símbolos > e < não estavam presentes nos cartões perfurados na época do seu projeto.

Os operadores relacionais sempre têm menor precedência do que os aritméticos, de modo que em expressões como

a + 1 > 2 \* b

as expressões aritméticas são avaliadas primeiro.

### 7.5.2 Expressões Booleanas

As expressões booleanas consistem em variáveis, em constantes, em expressões relacionais e em operadores booleanos, que normalmente incluem aqueles para operações AND, OR e NOT, e, às vezes, para OR exclusivo e para equivalência. Os operadores normalmente tomam somente operandos booleanos (variáveis, literais ou expressões relacionais) e produzem valores booleanos.

Na maioria das linguagens, os operadores booleanos, à semelhança dos operadores aritméticos, são avaliados em uma ordem de precedência hierárquica. Na maioria das linguagens imperativas comuns, o NOT unário tem a precedência mais alta, seguido do AND em um nível separado e do OR no nível mais baixo.

Uma vez que as operações aritméticas podem ser operandos de expressões relacionais, e estas podem ser operandos de expressões booleanas, as três categorias de operadores devem ser colocadas em níveis de precedência relativos entre si.

A precedência de todos os operadores Ada é

Mais alta	<b>**</b> , <b>abs</b> , <b>not</b>
	<b>*</b> , <b>/</b> , <b>mod</b> , <b>rem</b>
	<b>+</b> , <b>-</b> (unários)
	<b>+, -</b> , <b>&amp;</b> (binários)
	<b>=</b> , <b>/=</b> , <b>&lt;</b> , <b>&gt;</b> , <b>&lt;=</b> , <b>&gt;=</b> , <b>in</b> , <b>not in</b>
Mais baixa	<b>and</b> , <b>or</b> , <b>xor</b> , <b>and then</b> , <b>or else</b>

Note que os operadores booleanos da Ada, com exceção de **not**, compartilham do mesmo nível de precedência. Todos esses operadores booleanos de igual precedência são não-associativos. Se dois ou mais operadores booleanos diferentes aparecerem em uma expressão, parênteses devem ser usados para mostrar a ordem de avaliação. Por exemplo,

**A > B and A < C or K = 0**

é ilegal na Ada. Essa expressão pode ser escrita legalmente como

**(A > B and A < C) or K = 0**

ou

**A > B and (A < C or K = 0)**

Os operadores booleanos **and then** e **or else** da Ada serão discutidos na próxima seção.

O C é ímpar entre as linguagens imperativas populares em termos de que não tem nenhum tipo booleano e, dessa forma, nenhum valor booleano. Em vez disso, valores numéricos são usados para representá-los. Em lugar de valores booleanos, variáveis numéricas e constantes são usadas, sendo o zero considerado falso e todos os diferentes de zero considerados verdadeiros. O resultado da avaliação deste tipo de expressão é um número inteiro, com o valor 0 se for falsa e com 1 se for verdadeira.

Um resultado estranho do projeto do C é que a expressão

**a > b > c**

é válida. O operador relacional mais à esquerda é avaliado primeiro porque operadores relacionais do C são associativos à esquerda, produzindo 0 ou 1. Então, esse resultado é comparado com a variável **c**. Jamais há qualquer comparação entre **b** e **c**.

Quando os operadores não-aritméticos do C, do C++ e do Java são incluídos, há mais de 40 operadores e 15 diferentes níveis de precedência. Isso é uma clara evidência da riqueza dos conjuntos de operadores e da complexidade das expressões possíveis nessas linguagens.

A legibilidade determina que uma linguagem deve incluir um tipo booleano, conforme foi declarado no Capítulo 6, ao invés de usar tipos numéricos em expressões booleanas, como no C. Certa detecção de erros é perdida no uso que o C faz de tipos numéricos porque qualquer expressão desta natureza, seja pretendida ou não, é um operando legal para um operador booleano. Nas outras linguagens imperativas, qualquer expressão não-booleana usada como operando de um operador booleano é detectada como um erro.

No Pascal, os operadores booleanos têm maior precedência do que os operadores relacionais, de modo que a expressão

**a > 5 or a < 0**

é ilegal (porque 5 não é um operando booleano legal). A versão correta é

```
(a > 5) or (a < 0)
```

## 7.6 Avaliação Curto-Círcuito

Uma **avaliação curto-círcuito** de uma expressão tem seu resultado 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 0, porque  $0 * x = 0$  para qualquer  $x$ . Assim, quando  $a$  é 0, não há nenhuma necessidade de avaliar  $(b / 13 - 1)$  ou de realizar uma segunda multiplicação. Entretanto, em expressões aritméticas, esse atalho não é facilmente detectado durante a execução, de modo que ele jamais é tomado.

O valor da expressão booleana

```
(a >= 0) and (b < 10)
```

é independente da segunda expressão relacional se  $a < 0$ , porque  $(\text{FALSE} \text{ and } x)$  é **FALSE** para todos os valores de  $x$ . Assim, quando  $a < 0$ , não há nenhuma necessidade de avaliar  $b$ , a constante 10, a segunda expressão relacional, ou a operação **and**. Diferentemente do que acontece com as expressões aritméticas, esse atalho pode ser facilmente descoberto e tomado durante a execução.

Para ilustrar o problema potencial em avaliação de expressões booleanas sem usar curto-círcuito, suponha que o Java não a implemente. Suponha agora que escrevemos uma rotina de pesquisa em tabela usando a instrução **while**. Uma versão simples de código Java, assumindo que a variável **lista**, que possui **complis** elementos, é um vetor a ser pesquisado, e **chave** é o valor procurado, é:

```
indice = 0;
while ((indice < complis) && (lista[indice] != chave))
    indice++;
```

Se a avaliação não é feita em curto-círcuito, ambas as expressões relacionais na expressão booleana da instrução **while** serão avaliadas, independentemente do valor da primeira. Assim, se **chave** não estiver em **lista**, a rotina terminará com erro de subscripto fora da faixa. A mesma iteração que tem **indice == complis** fará referência à **lista[complis]**, que causa o erro de indexação porque **lista** foi declarada para ter **complis-1** como limite superior do valor de seu subscripto.

Se uma linguagem oferecer avaliação curto-círcuito de expressões booleanas e ela for usada, não será um problema. No exemplo precedente, um esquema de avaliação curto-círcuito avaliaria o primeiro operando do operador AND, mas pularia o segundo operando se o primeiro operando fosse falso.

A avaliação curto-círcuito de expressões expõe o problema de permitir efeitos colaterais em expressões. Suponhamos que a avaliação curto-círcuito seja usada em uma expressão e parte da expressão que contém um efeito colateral não seja avaliada. Então, o efeito colateral somente ocorrerá em avaliações completas da expressão inteira. Se a exatidão do

programa depender do efeito colateral, a avaliação curto-circuito poderá resultar em um sério erro. Por exemplo, considere a expressão C

```
(a > b) || (b++ / 3)
```

Nessa expressão b é modificado (na segunda expressão aritmética) somente quando a <= b. Se o programador presumir que b será modificado todas as vezes que essa expressão for avaliada durante a execução, o programa falhará.

A definição do FORTRAN 77 reconhece esse problema e simplesmente estabelece que o implementador pode optar por não avaliar qualquer outra coisa de uma expressão, a não ser o necessário para determinar o resultado. O aviso relevante reside na condição de que se a parte não-avaliada da expressão for uma chamada à função, esta deverá ser fixada em "não-definida" pela avaliação curto-circuito. Entretanto, há problema na implementação real da regra. O principal é a dificuldade de detectar qualquer um desses efeitos colaterais relevantes das funções.

A Ada permite que o programador especifique a avaliação curto-circuito dos operadores booleanos AND e OR usando os de duas palavras **and then** e **or else**. Por exemplo, novamente supondo que LISTA seja declarada como tendo uma faixa de subscrito de 1 ... COMPLIS, o código Ada

```
INDICE := 1;
while (INDICE <= COMPLIS) and then (LISTA (INDICE) /= CHAVE)
  loop
    INDICE := INDICE + 1;
  end loop;
```

não causará um erro quando CHAVE não constar em LISTA e INDICE tornar-se maior do que COMPLIS.

No C, no C++ e no Java, os operadores AND e OR comuns, **&&** e **||**, respectivamente, são curto-circuitos. Porém, essas linguagens também têm operadores AND e OR bit a bit, **&** e **|**, respectivamente, que podem ser usados em operandos com valor booleano e não são curto-circuitos.

A inclusão tanto dos operadores curto-círcito como dos comuns é, evidentemente, o melhor projeto porque oferece ao programador a flexibilidade de escolher a avaliação de curto-círcito para qualquer uma ou para todas as expressões booleanas.

## 7.7 Instruções de Atribuição

Conforme declaramos anteriormente, a instrução de atribuição é uma das instruções fundamentais das linguagens imperativas. Ela oferece o mecanismo por meio do qual o usuário pode modificar dinamicamente as vinculações de valores a variáveis. Na seção seguinte, a forma mais simples de atribuição será discutida. As seções subsequentes descrevem uma variedade de alternativas.

### 7.7.1 Atribuições Simples

A sintaxe geral da instrução de atribuição simples é

```
<variavel_alvo> <operador_de_atribuicao> <expressao>
```

O FORTRAN, o BASIC, a PL/I, o C, o C++ e o Java usam um símbolo de igualdade para o operador de atribuição. Isso pode gerar confusão se o símbolo de igualdade também for usado como um operador relacional, como na PL/I e no BASIC. Por exemplo, a atribuição PL/I

```
A = B = C
```

define A para o valor booleano da expressão relacional  $B = C$ , ainda que pareça que essas variáveis estejam sendo definidas como iguais entre si. Nos casos das outras linguagens que usam  $=$  para atribuição, um símbolo diferente é usado para o operador relacional de igualdade, o que evita o problema de sobrecarregar o operador de atribuição.

O ALGOL foi o pioneiro no uso de  $:=$  como operador de atribuição, e muitas linguagens posteriores seguiram essa opção.

O operador de atribuição no C, no C++ e no Java é tratado de maneira muito semelhante a um operador binário e, como tal, ele pode aparecer incorporado nas expressões. Tal operador será discutido na Seção 7.7.6.

As opções de projeto sobre como as atribuições são usadas em uma linguagem têm variado amplamente. Em algumas linguagens, como no FORTRAN, no Pascal e na Ada, ela somente pode aparecer como uma instrução independente, e o destino restringe-se a uma única variável. Há, entretanto, muitas alternativas.

### 7.7.2 Alvos Múltiplos

Uma alternativa para a instrução de atribuição simples é permitir a atribuição do valor da expressão a mais de uma localização. Por exemplo, na PL/I, a instrução

```
SOMA, TOTAL = 0
```

atribui o valor zero tanto a SOMA como a TOTAL. Instruções de atribuição de alvos múltiplos são uma conveniência para os programadores, mas ela não é significativa.

Os efeitos das atribuições de alvos múltiplos também podem ser conseguidos usando-se o operador de atribuição no C, no C++ e no Java, conforme discutiremos na Seção 7.7.6.

### 7.7.3 Alvos Condicionais

O C++ e o Java permitem alvos condicionais em instruções de atribuição. Por exemplo, considere

```
bandeira ? cont1 : cont2 = 0;
```

Isso é equivalente a

```
if (bandeira) cont1 = 0; else cont2 = 0;
```

### 7.7.4 Operadores de Atribuição Compostos

Um **operador de atribuição composto** é um método abreviado de 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 primeiro operando na expressão no lado direito, como em

```
a = a + b
```

Os operadores de atribuição compostos foram introduzidos pelo ALGOL 68 e posteriormente adotados sob uma forma ligeiramente diferente pelo C, e fazem parte do C++ e do Java. A sintaxe dos operadores de atribuição compostos é a concatenação do operador binário desejado com o operador =. Por exemplo,

```
soma += valor;
```

equivale a

```
soma = soma + valor;
```

O C, o C++ e o Java têm versões dos operadores de atribuição compostos para a maioria de seus operadores binários.

### 7.7.5 Operadores de Atribuição Unários

O C, o C++ e o Java incluem dois operadores aritméticos unários especiais, de fato, atribuições abreviadas. Elas combinam operações de incremento e de decremento com atribuição. Os operadores, ++ para incremento e -- para decremento, podem ser usados em expressões ou para formar instruções de atribuições independentes de um único operador. Eles podem aparecer como operadores prefixados, significando que precedem os operandos, ou como operadores pós-fixados, significando que vêm depois dos operandos. Na instrução de atribuição

```
soma = ++ cont;
```

o valor de `cont` é incrementado em 1 e depois é atribuído a `soma`. Isso também poderia ser declarado como

```
cont = cont + 1;
soma = cont;
```

Se o mesmo operador for usado como um operador pós-fixo, como em

```
soma = cont ++;
```

a atribuição do valor de `cont` a `soma` ocorre primeiro; depois, `cont` é incrementado. O efeito é o mesmo que o das duas instruções

```
soma = cont;
cont = cont + 1;
```

Um exemplo do uso do operador de incremento unário para formar uma instrução de atribuição completa é

```
cont ++;
```

que simplesmente incrementa `cont`. Ela não parece uma atribuição, mas certamente é. Ela é equivalente à instrução

```
cont = cont + 1;
```

Quando dois operadores unários aplicam-se ao mesmo operando, a associação ocorre da direita para a esquerda. Por exemplo, em

```
- cont ++
```

`cont` é incrementado primeiro e depois é transformado em negativo. Assim, ele é

```
- (cont ++)
```

não

```
(- cont) ++
```

Os operadores de incremento e decremento do C, do C++ e do Java são usados freqüentemente para formar expressões de subscrito de matrizes.

O computador PDP-11, em que o C foi implementado pela primeira vez, tem modos de endereçamento de auto-incremento e de auto-decremento, que são versões de hardware dos operadores de incremento e de decremento do C quando eles são usados como índices de matrizes. Poderíamos imaginar, a partir disso, que o projeto desses operadores C basearam-se no desenho da arquitetura do PDP-11. Essa conclusão estaria errada, entretanto, porque os operadores C foram herdados da linguagem B, projetada antes do PDP-11.

### 7.7.6 A Atribuição como uma Expressão

No C, no C++ e no Java, a instrução de atribuição produz um resultado, que é o mesmo valor atribuído ao alvo. Portanto, ele pode ser usado como uma expressão e como um operando em outras expressões. Esse conceito trata o operador de atribuição de uma maneira muito semelhante a qualquer outro binário, exceto que tem o efeito colateral de modificar seu operando esquerdo. Por exemplo, no C, é comum escrever instruções como

```
while ( (ch = getchar()) != EOF) { ... }
```

Nessa instrução, o caractere seguinte a partir do arquivo de entrada padrão, usualmente o teclado, é obtido com `getchar` e atribuído à variável `ch`. O resultado ou o valor atribuído, é, então, comparado com a constante `EOF`. Se `ch` não for igual a `EOF`, a instrução composta `{ ... }` será executada. Note que a atribuição deve ser colocada entre parênteses — nessas linguagens, a precedência do operador de atribuição é menor do que a dos operadores relacionais. Sem os parênteses, o novo caractere seria comparado com `EOF` primeiro. Depois, o resultado dessa comparação, 0 ou 1, seria atribuído a `ch`.

A desvantagem de permitir que instruções de atribuição sejam operandos em expressões acarreta mais um tipo de efeito colateral de expressão que pode levar a expressões difíceis de ler e de entender. Uma expressão com qualquer tipo de efeito colateral tem essa desvantagem: não pode ser lida como uma expressão, o que, em Matemática, é uma denotação de um valor, mas somente como uma lista de instruções com uma ordem ímpar de execução. Por exemplo, a expressão

```
a = b + (c = d / b++) - 1
```

denota as instruções

```
Atribuir b a temp
Atribuir b + 1 a b
Atribuir d / temp a c
Atribuir b + c a temp
Atribuir temp - 1 a a
```

Note que o uso que o C faz do operador de atribuição permite o efeito de atribuições a múltiplos alvos, como por exemplo,

```
soma = cont = 0;
```

em que `cont` é atribuído primeiramente a zero, e depois o valor de `cont` é atribuído a `soma`.

Há uma perda da capacidade de detecção de erros na operação de atribuição C que freqüentemente leva a erros de programa. Em especial, se digitarmos

```
if (x = y) ...
```

em vez de

```
if (x == y) ...
```

que é um erro fácil de ser cometido, ele não será detectado como um erro pelo compilador. Em vez de testar uma expressão relacional, o valor atribuído a `x` é testado (nesse caso, é o valor de `y` que alcança essa instrução). Esse é, de fato, um resultado de três decisões de projeto: permitir que as atribuições comportem como um operador binário comum, usar expressões aritméticas como operadores booleanos e usar dois operadores muito similares, `=` e `==`, com significados completamente diferentes. Eis um outro exemplo das deficiências de segurança de programas C e C++. Note que o Java permite somente expressões **booleanas** em suas instruções `if`, rejeitando o problema.

## 7.8 Atribuição de Modo Misto

Discutimos as expressões de modo misto na Seção 7.4.1. Com freqüência, as instruções de atribuição também são de modo misto. A questão de projeto é: o tipo da expressão tem de ser o mesmo do da variável em atribuição, ou pode-se usar coerção em alguns casos de não-coincidência de tipos?

O FORTRAN, o C e o C++ usam regras de coerção para atribuição de modo misto semelhantes àquelas que usam para suas expressões; ou seja, muitas das mesclagens de tipo possíveis são legais, com a coerção sendo aplicada livremente.

O Pascal inclui alguma coerção de atribuição; por exemplo, valores `integer` podem ser atribuídos a variáveis `real`, mas não o contrário. A Ada e o Modula-2 não permitem atribuição de modo misto.

Em um claro afastamento do C e do C++, o Java permite atribuições de modo misto somente se a coerção exigida for de alargamento. Assim, um valor `int` pode ser atribuído a uma variável `float`, mas não o contrário. Rejeitar a metade das possíveis atribuições de modo misto foi uma maneira simples mas efetiva de aumentar a confiabilidade do Java em relação ao C e C++.

Em todas as linguagens que permitem atribuição de modo misto, a coerção desenvolve-se somente depois que a expressão do lado direito tiver sido avaliada. Uma alternativa seria coagir todos os operandos no lado direito para o tipo do alvo antes da avaliação. Por exemplo, considere o seguinte código:

```
int a, b;  
float c;  
...  
c = a / b;
```

Uma vez que `c` é real, os valores de `a` e `b` poderiam ser coagidos para `float` antes da divisão, que poderia produzir um valor diferente para `c` do que se a coerção fosse retardada (por exemplo, se `a` fosse 2 e `b` fosse 3).

## RESUMO

As expressões consistem em constantes, em variáveis, em parênteses, em chamadas a função e em operadores. As instruções de atribuição incluem variáveis alvo, operadores de atribuição e expressões.

A semântica de uma expressão é determinada, em grande parte, pela ordem de avaliação de operadores. As regras de associatividade e de precedência para operadores nas expressões de uma linguagem determinam a ordem e a avaliação do operador naquelas expressões. A ordem de avaliação de operandos é importante se efeitos colaterais forem possíveis. As conversões de tipo podem ser de alargamento ou de estreitamento. Algumas conversões de estreitamento produzem valores errôneos. As conversões de tipo implícitas ou coerções em expressões são comuns, mesmo que eliminem o benefício da detecção de erros da verificação de tipos, o que, por sua vez, diminui a confiabilidade. As conversões de tipo explícitas, muitas vezes, são chamadas *casts*.

As instruções de atribuição têm surgido em uma ampla variedade de formas, inclusive alvos condicionais, alvos múltiplos e operadores de atribuição.

## QUESTÕES DE REVISÃO

1. Defina precedência de operadores e associatividade de operadores.
2. Defina efeito colateral funcional.
3. O que é uma coerção?
4. O que é uma expressão condicional?
5. O que é um operador sobrecarregado?
6. Defina conversões de estreitamento e de alargamento.
7. O que é uma expressão de modo-misto?
8. Como a ordem de avaliação de operandos interage com efeitos colaterais funcionais?
9. O que é avaliação curto-círcuito?
10. Cite uma linguagem que sempre faça avaliação curto-círcuito de expressões booleanas. Cite uma que nunca a faça. Cite uma em que seja permitido ao programador escolher.
11. Como o C suporta expressões relacionais e booleanas?
12. Qual é o propósito de um operador de atribuição composto?
13. Qual é a associatividade dos operadores aritméticos unários do C?
14. Qual é uma das possíveis desvantagens de tratar o operador de atribuição como se ele fosse um operador aritmético?
15. Quais atribuições de modo misto são permitidas na Ada?
16. Quais atribuições de modo misto são permitidas no Java?

**PROBLEMAS**

- Quando você poderia querer que o compilador ignorasse diferenças de tipos em uma expressão?
  - Apresente seus próprios argumentos contrários e favoráveis a permitir expressões aritméticas de modo misto.
  - Você acha que a eliminação de operadores sobrecarregados de sua linguagem favorita seria benéfica? Por que sim e por que não?
  - Seria uma boa idéia eliminar todas as regras de precedência de operadores e exigir parênteses para mostrar a precedência desejada nas expressões? Por que sim e por que não?
  - As operações de atribuição do C (por exemplo, `+=`) devem ser incluídas em outras linguagens? Por que sim e por que não?
  - As formas de atribuição de único operando do C (por exemplo, `++cont`) devem ser incluídas em outras linguagens? Por que sim e por que não?
  - Descreva uma situação em que o operador de adição em uma linguagem de programação não seria comutativo.
  - Descreva uma situação em que o operador de adição em uma linguagem de programação não seria associativo.
  - Escreva um segmento de programa Pascal, usando uma construção `while` para procurar em um vetor de números inteiros um número inteiro particular, que funcionaria mesmo que uma avaliação curto-circuito de expressões booleanas não fosse feita.
- Suponha as seguintes regras de associatividade e precedência para expressões:

<i>Precedência:</i>	<i>Mais alta</i>	<i>* , / , not</i>
		<i>+ , - , &amp; , mod</i>
		<i>- (únario)</i>
		<i>= , /= , &lt; , &lt;= , &gt;= , &gt;</i>
		<i>and</i>
		<i>or , xor</i>
<i>Associatividade</i>	<i>Mais baixa</i>	
	<i>Da esquerda para a direita</i>	

- Mostre a ordem de avaliação das seguintes expressões colocando entre parênteses todas as subexpressões e colocando um sobreescrito 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.  $a * (b - 1) / c \text{ mod } d$   
c.  $(a - b) / c \& (d * e / a - 3)$   
d.  $-a \text{ or } c = d \text{ and } e$   
e.  $a > b \text{ xor } c \text{ or } d \leq 17$   
f.  $-a + b$
- Mostre a ordem de avaliação das expressões do Problema 10, supondo que não haja quaisquer regras de precedência e que todos os operadores associem-se da direita para a esquerda.
- Escreva uma descrição BNF das regras de precedência e de associatividade definidas para as expressões do Problema 10. Suponha que os únicos operandos sejam os nomes `a`, `b`, `c`, `d` e `e`.
- Usando a gramática do Problema 12, desenhe árvores de análise das expressões do Problema 10.
- Admitindo que a função `FUN` seja definida como

```
function FUN(var K : integer) : integer;
begin
  K := K + 4;
```

```

    FUN := 3 * K - 1
    end;

```

Supondo que FUN seja usada em um programa da seguinte maneira:

```

...
I := 10;
SOMA1 = (I / 2) + FUN(I);
J := 10;
SOMA2 := FUN(J) + (J / 2);

```

Quais são os valores de SOMA1 e SOMA2

- se os operandos nas expressões forem avaliados da esquerda para a direita?
- se os operandos nas expressões forem avaliados da direita para a esquerda?

15. Admitindo que a função C **fun** seja definida como

```

int fun(int *k) {
    *k += 4;
    return 3 * (*k) - 1;
}

```

Supondo que fun seja usada em um programa da seguinte maneira:

```

void main() {
    int i = 10, j = 10, somal, soma2;
    somal = (i / 2) + fun(&i);
    soma2 = fun(&j) + (j / 2);
}

```

Determine os valores de somal e soma2 rodando o programa em um computador. Explique os resultados.

- Qual é seu principal argumento contrário (ou favorável) às regras de precedência de operadores da APL?
- Para alguma linguagem de sua escolha, componha uma lista de símbolos de operadores que possa ser usada para eliminar toda sobrecarga de operador.
- Determine se as conversões de tipo explícitas de estreitamento em duas linguagens que você conhece apresentam mensagens de erro quando um valor convertido perde sua utilidade.
- Deve-se permitir que um compilador de otimização para C ou para C++ mude a ordem de subexpressões em uma expressão booleana? Por que sim e por que não?
- Responda a pergunta do Problema 19 em relação à Ada.
- Considere o seguinte programa C:

```

int fun(int *i) {
    *i += 5;
    return 4;
}
void main() {
    int x = 3;
    x = x + fun(&x);
}

```

Qual é o valor de x depois da instrução de atribuição em main, supondo que:

- os operandos são avaliados da esquerda para a direita?
- os operandos são avaliados da direita para a esquerda?

- Escreva um programa de teste em sua linguagem favorita que determine e apresente como resultado a precedência e a associatividade de seus operadores aritméticos e booleanos.

Hidden page

## Capítulo 8

# Estruturas de Controle no Nível da Instrução



### Peter Naur

Peter Naur reside em Copenhague e envolveu-se fortemente no projeto de linguagens depois que o primeiro relatório do ALGOL foi publicado em 1958. Ele tornou-se editor do *ALGOL Bulletin*, um meio de discussão europeu para pessoas envolvidas no processo de desenvolvimento desta linguagem. Modificou a notação que Backus usou em 1959 e usou-a para apresentar a última versão do ALGOL no encontro sobre este em Paris, em 1960.

- 8.1** Introdução
- 8.2** Instruções Compostas
- 8.3** Instruções de Seleção
- 8.4** Instruções Iterativas
- 8.5** Desvio Incondicional
- 8.6** Comandos Protegidos
- 8.7** Conclusões

O fluxo de controle ou seqüência de execução em um programa pode ser examinado em diversos níveis. No Capítulo 7, discutimos o fluxo de controle dentro de expressões, governado pelas regras de associatividade e de precedência de operadores. No nível mais alto está o fluxo de controle entre as unidades de programa, o que será discutido nos Capítulos 9 e 13. Entre esses dois extremos, está a importante questão do fluxo de controle entre as instruções, o assunto deste capítulo.

Iniciamos apresentando uma visão geral da evolução das instruções de controle nas linguagens de programação imperativas, seguida de um cuidadoso exame das construções de seleção, incluindo aquelas para seleção unidirecional, bidirecional e múltipla. Discutiremos, então, a variedade de construções de laço que têm sido desenvolvidas e usadas nas linguagens de programação. Depois, daremos uma olhada bem de perto na controversa instrução de desvio incondicional. Finalmente, descreveremos as construções de controle de comando protegido.

## 8.1 Introdução

As computações em programas de linguagens imperativas são realizadas avaliando-se expressões e atribuindo-se os valores resultantes a variáveis. Entretanto, há somente alguns programas úteis que consistem inteiramente em instruções de atribuição. Pelo menos dois mecanismos lingüísticos adicionais são necessários para tornar flexíveis e poderosas as computações em programas: alguns meios de selecionar entre caminhos (de execução da instrução) alternativos do fluxo de controle e alguns de provocar a execução repetida de certos conjuntos de instruções. Estas que oferecem tais capacidades são chamadas de **instruções de controle**.

As instruções de controle da primeira linguagem de programação bem-sucedida, o FORTRAN, foram, com efeito, projetadas pelos arquitetos do IBM 704. Todas estavam diretamente relacionadas com instruções de linguagem de máquina, de modo que suas capacidades tinham mais a ver com o projeto da instrução do que com o projeto da linguagem. Na época, pouco se sabia sobre a dificuldade de programar e, como resultado, as instruções de controle do FORTRAN, no final da década de 50, eram consideradas totalmente aceitáveis. As seções subsequentes deste capítulo discutem as instruções de controle do FORTRAN e as razões de serem elas consideradas inadequadas para o desenvolvimento de software.

Muita pesquisa e discussão foram destinadas às instruções de controle em 10 anos, entre meados da década de 60 e meados da década de 70. Uma das principais conclusões de tais esforços foi que, mesmo uma única instrução de controle (uma goto selecionável), é evidentemente suficiente; uma linguagem projetada para não incluí-la precisa somente de um pequeno número de instruções de controle diferentes. De fato, demonstrou-se que todos os algoritmos possíveis de serem expressos por meio de fluxogramas podem ser codificados em uma linguagem de programação com somente duas instruções de controle: uma para escolher entre dois caminhos de fluxo de controle e uma para iterações controladas logicamente (Böhm e Jacopini, 1966). Um resultado importante é que as instruções de desvio incondicional são supérfluas — possivelmente convenientes, mas não-essenciais. Este fato, combinado com os problemas de usar desvios incondicionais, provocou muito debate sobre a goto, conforme discutiremos na Seção 8.5.1.

Os programadores preocupam-se menos com os resultados da pesquisa teórica sobre as instruções de controle do que com a capacidade de escrita e de legibilidade. Todas as linguagens popularizadas contêm mais instruções de controle do que as duas minimamente

exigidas, porque a capacidade de escrita é aumentada pelo seu maior número. Por exemplo, em vez de exigir o uso de um **while** para todos os laços, é mais fácil escrever programas quando um **for** pode ser usado para construir laços naturalmente controlados por um contador. O fator principal que restringe o número de instruções de controle em uma linguagem é a legibilidade, porque a presença de um grande número de suas formas exige que os leitores do programa aprendam uma linguagem maior. Lembre-se de que poucas pessoas aprendem tudo de uma linguagem muito grande. Pelo contrário, elas aprendem o subconjunto daquilo que escolhem usar, que, muitas vezes, é diferente daquele usado pelo programador que escreveu o programa que estão tentando ler. Por outro lado, um número demasiado pequeno de instruções pode exigir seu uso em nível mais baixo, como, por exemplo, a **goto**, que torna os programas menos legíveis.

A questão referente à melhor coleção de instruções de controle para oferecer as capacidades necessárias de escrita desejada foi amplamente discutida no último quarto de século. É essencialmente uma questão de quanto uma linguagem deve ser expandida para aumentar sua capacidade de escrita, à custa de sua simplicidade, de seu tamanho e de sua legibilidade.

Uma **estrutura de controle** é uma instrução de controle e sua coleção de comandos cuja execução ela controla.

## 8.2 Instruções Compostas

Um dos recursos de linguagem auxiliares que ajuda a tornar o projeto de instruções de controle mais fácil é um método para formar coleções de instruções. A principal razão para as insuficiências das instruções de controle das primeiras versões do FORTRAN era a falta dessa construção.

As instruções compostas, que foram introduzidas pelo ALGOL 60, permitem que uma coleção de instruções seja abstraída para uma única. Esse é um conceito poderoso que pode ser usado com grande proveito no projeto de instruções de controle. As declarações de dados podem ser adicionadas ao início de uma instrução composta em diversas linguagens, tornando-a um bloco, conforme discutimos no Capítulo 5.

O Pascal seguiu o projeto do ALGOL 60 quanto a instruções compostas, mas não permite blocos. As linguagens baseadas no C (C, C++ e Java) usam chaves para delimitar tanto instruções compostas como blocos. Algumas linguagens eliminaram a necessidade de instruções especialmente delimitadas, integrando-as em suas estruturas de controle. Essas serão discutidas na seção seguinte.

Há uma questão de projeto pertinente a todas as instruções de controle de seleção e de iteração: se ela pode ter múltiplas entradas. Todas as instruções de seleção e de iteração controlam a execução de segmentos de código, e a questão é se a execução destes últimos sempre se inicia com a primeira instrução do segmento. Agora, acredita-se que múltiplas entradas acrescentam pouco à flexibilidade de uma construção de controle, em relação à redução da legibilidade causada pelo aumento da complexidade. Note que múltiplas entradas são possíveis somente em linguagens que incluem **gotos** e rótulos de instrução.

Neste ponto, o leitor pode perguntar-se por que as múltiplas saídas de estruturas de controle não estão aqui listadas como uma questão de projeto. Uma vez que não há nenhum perigo nas saídas múltiplas, elas estão incluídas em todas as linguagens. Desta forma, elas não são um grande problema.

## 8.3 Instruções de Seleção

Uma **instrução de seleção** oferece os meios de escolher entre dois ou mais caminhos de execução em um programa. Essas instruções são partes essenciais e fundamentais de todas as linguagens de programação, como foi provado por Böhm e Jacopini.

As instruções de seleção situam-se em duas categorias gerais: seleção bidirecional e seleção  $n$ -direcional ou múltipla. Dentro da primeira, há uma categoria degenerada chamada seletores unidirecionais. Há, também, um seletor múltiplo degenerado, o **IF** aritmético do FORTRAN, um seletor tridirecional.

### 8.3.1 Instruções de Seleção Bidirecional

As variações na evolução das instruções de seleção bidirecional das linguagens imperativas contemporâneas basearam-se em um conjunto de considerações de projeto, ainda que elas sejam bastante semelhantes.

#### 8.3.1.1 Questões de Projeto

Talvez a questão de projeto mais simples referente aos seletores bidirecionais seja o tipo de expressão que controla o seletor. Uma questão de projeto mais interessante refere-se ao fato de instruções únicas, instruções compostas ou sequências de instruções poderem ser selecionadas. Um seletor capaz de selecionar somente instruções únicas é gravemente limitado e usualmente leva a uma forte dependência de **gotos**. Permitir que instruções compostas sejam selecionadas foi um grande passo na evolução das instruções de controle. Permitir que sequências de instruções sejam selecionadas exige que o seletor inclua uma entidade semântica para finalizá-las. Outra questão interessante e relacionada é como o significado de seletores aninhados é especificado — pela sintaxe ou por uma regra de semântica estática.

Tais questões de projeto são resumidas da seguinte maneira:

- Qual é a forma e o tipo da expressão que controla a seleção?
- Uma instrução única, uma sequência de instruções ou uma instrução composta pode ser selecionada?
- Como o significado de seletores aninhados pode ser especificado?

#### 8.3.1.2 Exemplos de Seletores Bidirecionais

Todas as linguagens imperativas incluem um seletor unidirecional, na maioria dos casos, como uma forma secundária de um seletor bidirecional. Uma exceção é o FORTRAN IV.

O seletor unidirecional do FORTRAN IV, chamado instrução lógica **IF**, tem a forma **IF** (**expressão booleana**) **instrução**

Sua semântica é que a instrução selecionável seja executada somente se a expressão booleana for avaliada como verdadeira. As opções de projeto para a instrução lógica **IF** do FORTRAN IV são as seguintes: a expressão de controle do seletor é do tipo booleano, e somente uma única instrução é selecionável. O aninhamento de instruções lógicas **IF** não é permitido.

A instrução lógica **IF** unidirecional é muito simples, ainda que altamente inflexível. O fato de uma única instrução poder ser selecionada promove o uso de **goto**, porque, muitas

vezes, mais de uma deve ser condicionalmente executada. A única maneira razoável de executar condicionalmente um grupo de instruções é desviar condicionalmente para fora do grupo. Por exemplo, suponhamos que queremos inicializar as duas variáveis *I* e *J* para os valores 1 e 2, mas somente se FLAG for 1. A maneira típica de fazer isso no FORTRAN IV é

```
IF(FLAG .NE. 1) GO TO 20
I = 1
J = 2
20 CONTINUE
```

A lógica negativa exigida por essa forma pode ser prejudicial para a legibilidade.

Essa estrutura pode ter múltiplas entradas, porque qualquer uma das instruções de atribuição pode ser rotulada e, dessa forma, ser o alvo de uma GO TO em qualquer lugar no programa.

A instrução composta, introduzida pelo ALGOL 60, oferece a construção de seleção com um mecanismo simples para executar condicionalmente grupos de instruções.

A maioria das linguagens que vieram depois do ALGOL 60, inclusive o C, C++, Java e FORTRAN 90, oferece seletores unidirecionais que podem selecionar uma instrução composta ou uma seqüência de instruções.

O ALGOL 60 introduziu o primeiro seletor bidirecional, cuja forma geral é:

```
if (expressão_booleana) then
    instrução
else
    instrução
```

em que qualquer uma ou ambas as instruções selecionáveis poderiam ser compostas. A instrução que se segue à palavra reservada **then** é chamada **cláusula then**, e a instrução que se segue à palavra reservada **else** é chamada **cláusula else**.

A semântica de um seletor bidirecional é que a cláusula **then** é executada se a expressão booleana for avaliada como verdadeira; caso contrário, a cláusula **else** será executada. Em nenhuma circunstância ambas as cláusulas são executadas.

Todas as linguagens imperativas projetadas desde meados da década de 60 têm incorporado instruções de seleção bidirecional, ainda que a sintaxe tenha variado. Note que mesmo nas linguagens que não usam a palavra reservada **then**, chamamos o segmento executado, quando a expressão booleana é verdadeira, de cláusula **then**.

Os seletores bidirecionais do ALGOL 60, do C e do C++ podem ter mais de uma entrada, mas os da maioria das linguagens contemporâneas pode ter somente um.

### 8.3.1.3 Aninhando Seletores

Um problema interessante surge quando construções de seleção bidirecionais podem ser aninhadas. Considere o seguinte código assemelhado ao Java:

```
if (soma == 0)
    if (cont == 0)
        resultado = 0;
    else
        resultado = 1;
```

Essa construção pode ser interpretada de duas maneiras diferentes, dependendo se a cláusula **else** seja coincidente com a primeira cláusula **then** ou com a segunda. Note que o

recurso parece indicar que a cláusula **else** pertence à primeira cláusula **then**. Porém, o recurso não tem nenhum efeito sobre a semântica na maioria das linguagens contemporâneas e, portanto, é ignorado por seus compiladores.

O ponto crucial do problema, nesse exemplo, é que a cláusula **else** segue duas **then** sem nenhuma **else** interveniente, e não há nenhum indicador sintático para especificar uma identificação desta última com uma das cláusulas **then**. No Java, como em muitas outras imperativas, a semântica estática da linguagem especifica que a **else** sempre faz par com a **then** não-emparelhada mais recentemente. Em vez de uma entidade sintática, é usada uma regra para superar a ambigüidade. Desse modo, no exemplo acima, a cláusula **else** seria a alternativa para a segunda cláusula **then**. A desvantagem de usar uma regra no lugar de alguma entidade sintática está no fato de que mesmo que o programador tenha pretendido que a **else** fosse a alternativa para a primeira **then** e que o compilador considerasse a estrutura sintaticamente correta, sua semântica é o oposto. Para forçar a semântica alternativa em Java, uma forma sintática diferente é necessária, em que o **if-then** interno é colocado em uma instrução composta.

Os projetistas do ALGOL 60 optaram por usar a sintaxe, no lugar de uma regra, para ligar **else** a **then**. Especificamente, não é permitido que uma instrução **if** seja aninhada diretamente a uma cláusula **then**. Se um **if** precisar ser aninhado em uma **then**, ele deverá ser colocado em uma instrução composta. Por exemplo, se a construção de seleção acima fosse para que a **else** fizesse par com a segunda **then**, no ALGOL 60, ela seria escrita como

```
if soma = 0 then
begin
  if cont = 0 then
    resultado := 0
  else
    resultado := 1
end
```

Se a cláusula **else** fosse destinada a fazer par com a primeira cláusula **then**, ela seria escrita como

```
if soma = 0 then
begin
  if cont = 0 then
    resultado := 0
  end
else
  resultado := 1
```

Isto é exatamente o necessário para obter este significado em Java. A diferença entre os dois projetos está no fato de que a versão em Java permite que uma pessoa escreva o seletor aninhado com aparência de emparelhar a **else** com a primeira **then**, mas isso não acontece, considerando que essa mesma forma é sintaticamente ilegal no ALGOL 60, evitando o problema sutil do Java.

O C e o C++ têm o mesmo problema que o Java com o aninhamento de instruções de seleção. A Perl exige que todas as cláusulas **then** e **else** sejam compostas, evitando completamente o problema.

Uma alternativa para o projeto do ALGOL 60 é exigir palavras de fechamento especiais para cláusulas **then** e **else**, conforme discutiremos na seção seguinte.

#### 8.3.1.4 Palavras Especiais e Fechamento de Seleção

Considere a estrutura sintática da instrução Java **if**. As cláusulas **then** seguem a expressão central e as cláusulas **else** são introduzidas pela palavra reservada **else**. Quando a cláusula **then** é uma única instrução, e a cláusula **else** está presente, ainda que não haja nenhuma necessidade de marcar o final, a palavra reservada **else** marca, de fato, o final da cláusula **then**. Quando a cláusula **then** é um composto, ela é finalizada por uma chave à direita (**}**). Entretanto, se a última cláusula em um **if for then ou else**, e não um composto, não haverá nenhuma entidade sintática para marcar o fim da construção de seleção inteira. O uso de uma palavra especial resolveria a questão da semântica de seletores aninhados e também aumentaria a legibilidade da construção. Esse é o projeto da construção de seleção no FORTRAN 90 e na Ada. Por exemplo, considere a seguinte construção Ada:

```
if A > B then
    SOMA := SOMA + A;
    ACONT := ACONT + 1;
else
    SOMA := SOMA + B;
    BCONT := BCONT + 1;
end if;
```

Seu desenho é mais regular do que o das construções de seleção Pascal e ALGOL 60, porque a forma é a mesma, independentemente do número de instruções nas cláusulas **then** e **else**. Estas consistem em sequências de instruções em vez de instruções compostas. A primeira interpretação do exemplo de seletor no início da Seção 8.3.1.3 pode ser escrita em Ada da seguinte maneira:

```
if SOMA = 0 then
    if CONT = 0 then
        RESULTADO := 0;
    else
        RESULTADO := 1;
    end if;
end if;
```

Uma vez que as palavras reservadas **end if** fecham o **if** aninhado, é evidente que a cláusula **else** coincide com a **then** interna.

A segunda interpretação da construção de seleção da Seção 8.3.1.3 pode ser escrita em Ada da seguinte maneira:

```
if SOMA = 0 then
    if CONT = 0 then
        RESULTADO := 0;
    end if;
else
    RESULTADO := 1;
end if;
```

### 8.3.2 Construções de Seleção Múltipla

A construção de **seleção múltipla** permite a seleção de uma instrução, dentre qualquer número de instruções ou de grupos de instrução. Portanto, ela é uma generalização de um seletor. De fato, seletores unidirecionais e bidirecionais podem ser construídos com um múltiplo. Suas formas originais pertencem ao FORTRAN, como você deve ter imaginado.

A necessidade de escolher entre mais de dois caminhos de controle em um programa é comum. Não obstante um seletor múltiplo possa ser construído a partir de bidirecionais e gotos, as estruturas resultantes são confusas, difíceis de escrever e de ler e pouco confiáveis. Portanto, fica clara a necessidade de uma estrutura especial.

#### 8.3.2.1 Questões de Projeto

Algumas das questões de projeto referentes aos seletores múltiplos são semelhantes às relativas aos bidirecionais. Por exemplo, uma questão é se instruções únicas, compostas ou seqüências de instruções podem ser selecionadas. Se a estrutura de seleção múltipla inteira estiver encapsulada, todos os segmentos selecionáveis deverão estar juntos. Isso impede o fluxo de controle de escapar para instruções que não fazem parte da estrutura de seleção múltipla durante sua execução. Uma vez que isso afeta a legibilidade, o encapsulamento é uma questão. Outra questão relativa aos seletores bidirecionais é a do tipo de expressão na qual o seletor baseia-se. Nesse caso, a faixa de possibilidades é maior, em parte porque o número de seleções possíveis é maior. Um seletor bidirecional precisa de uma expressão com somente dois valores possíveis. Em seguida, há a questão de se somente um único segmento selecionável pode ser executado quando a construção é executada. Essa não é uma questão referente aos seletores bidirecionais porque todo projeto permite que somente uma das cláusulas esteja em um caminho de controle durante uma execução. Como veremos, a resolução dessa questão dos seletores múltiplos é um compromisso entre confiabilidade e flexibilidade. Finalmente, há a questão de qual será o resultado da expressão seletora ser avaliada para um valor que não seleciona um dos segmentos. A escolha aqui é entre simplesmente impedir que a situação apareça e ter uma regra que descreva o que acontecerá quando ela surgir. Os valores de expressão seletora não-representados serão discutidos na Seção 8.3.2.3.

Apresentamos, a seguir, um resumo dessas questões de projeto:

- Qual é a forma e o tipo da expressão que controla a seleção?
- Instruções únicas, seqüências de instruções ou instruções compostas podem ser selecionadas?
- A construção inteira é encapsulada em uma estrutura sintática?
- O fluxo de execução através da estrutura restringe-se a incluir apenas um único segmento selecionável?
- Como os valores da expressão seletora não-representados devem ser manipulados, se for o caso?

#### 8.3.2.2 Seletores Múltiplos Antigos

Os seletores múltiplos introduzidos no FORTRAN I são incluídos, aqui, por motivos históricos e, também, porque eles ainda fazem parte da versão mais recente desta linguagem, o FORTRAN 90. À semelhança de outras instruções de controle do FORTRAN I, seus seletores múltiplos baseiam-se diretamente nas instruções do IBM 704.

Conforme declarou-se anteriormente, o **seletor tridirecional** do FORTRAN, chamado de **IF aritmético**, é um caso degenerado especial de instrução de seleção múltipla. Entretanto, uma vez que somente três coleções de instruções, ou menos, podem ser selecionadas, ela não é, estritamente falando, uma instrução de seleção múltipla.

O **IF aritmético** escolhe entre três endereços-alvo de desvio baseando-se no valor de uma expressão. O desvio baseia-se matematicamente na tricotomia de números, o que significa que determinado valor numérico é maior do que zero, igual a zero ou menor do que zero. O **IF aritmético** tem a forma

**IF (expressão aritmética) N1, N2, N3**

em que N1, N2 e N3 são rótulos de instrução para os quais o controle deve ser transferido se o valor da expressão for negativo, zero ou maior do que zero, respectivamente. Por exemplo, se um **IF aritmético** for usado para selecionar entre três sequências de instrução, sua forma geral freqüentemente será a seguinte:

```
IF (expressão) 10, 20, 30
 10 ...
  ...
  GO TO 40
 20 ...
  ...
  GO TO 40
 30 ...
  ...
 40 ...
```

De fato, esse tipo de seletor poderia ser muito mais prejudicial à legibilidade do que o exemplo ilustra porque as sequências de instruções a serem selecionadas podem estar, literalmente, em qualquer lugar na unidade de programa que contém a **GO TO**. Não existe nenhum encapsulamento sintático da **GO TO** e de suas sequências selecionáveis. Uma vez que o usuário é receptivo a colocá-la nas extremidades dos segmentos selecionáveis, uma execução da construção pode fazer com que o fluxo de controle percorra qualquer número de segmentos selecionáveis. O problema é que o erro de deixar um desses desvios não é detectado pelo compilador. Mesmo quando uma execução de segmentos múltipla é desejada, o aumento resultante de complexidade da estrutura é altamente prejudicial à legibilidade. Esse projeto é um compromisso entre confiabilidade e certa flexibilidade adicionada.

Pode-se entrar em um **IF aritmético** através de qualquer uma de suas instruções, vindo de qualquer parte do programa.

As duas primeiras instruções de seleção múltipla reais apareceram no FORTRAN I. Igual ao **IF aritmético**, elas fazem parte de todas as versões do FORTRAN. A **GO TO** computada do FORTRAN tem a forma

**GO TO (rótulo 1, rótulo 2, ..., rótulo n), expressão**

em que a expressão tem um valor inteiro, e os rótulos são todos definidos como de instrução no programa. A semântica da instrução define-se no valor da expressão usada para escolher um rótulo para o qual o controle deve ser transferido. O primeiro é associado ao valor 1, o segundo ao 2 e assim por diante. Se o valor estiver fora da faixa 1 a n, a instrução nada fará. Não há nenhuma detecção de erros incorporada.

O outro antigo seletor múltiplo do FORTRAN, a GO TO atribuída, tem uma forma similar à GO TO computada. Ambos os seletores múltiplos sofrem as mesmas deficiências do IF aritmético — a ausência de encapsulamento e possivelmente múltiplas entradas. Além disso, nenhuma delas restringe um fluxo de controle a um único segmento selecionável.

### 8.3.2.3 Seletores Múltiplos Modernos

Uma forma melhor de seletor múltiplo, chamada **case**, foi sugerida por C.A.R Hoare e incluída no ALGOL-W (Wirth e Hoare, 1966). Essa estrutura é encapsulada e tem uma única entrada. Desvios implícitos para um único ponto no final de toda a construção também são oferecidos para cada instrução selecionável ou composta. Isso restringe o fluxo de controle pela estrutura a um único segmento selecionável.

A forma geral do seletor múltiplo de Hoare é

```
case expressao_inteira of
  begin
    instrucao_1;
    ...
    instrucao_n
  end
```

em que as instruções poderiam ser únicas ou compostas. Escolhe-se a instrução executada pelo valor da expressão. Um valor igual a 1 escolhe a primeira e assim por diante.

A **case** Pascal é muito semelhante à do ALGOL-W, exceto que os segmentos selecionáveis são rotulados. Ela tem a forma

```
case expressao of
  lista_de_constants_1: instrucao_1;
  ...
  lista_de_constants_n: instrucao_n
end
```

em que a expressão é do tipo ordinal (inteiro, booleano, caractere ou enumeração). Como acontece com a maioria das instruções de controle Pascal (mas não com todas), as instruções selecionáveis podem ser instruções únicas ou instruções compostas.

A semântica da **case** Pascal é a seguinte: a expressão é avaliada e o valor comparado com as constantes na lista de constantes. Se uma igualdade for encontrada, o controle transfere-se para a instrução anexada à constante coincidente. Quando conclui-se a execução da instrução, o controle transfere-se para a primeira instrução seguinte à **case** inteira.

As listas de constantes devem ser, obviamente, do mesmo tipo que a expressão. Elas devem ser mutuamente exclusivas, mas não precisam ser exaustivas; ou seja, uma constante pode não aparecer em mais de uma de suas listas, mas nem todos os valores na faixa do tipo da expressão precisam estar presentes nestas listas.

Note que, embora as listas de constantes dos segmentos selecionáveis tenham uma forma similar à dos rótulos, elas não são os alvos legais das instruções de desvio.

Estranhamente, a primeira definição amplamente usada do Pascal (Jensen e Wirth, 1974) não se preocupou com a possibilidade de ocorrer valores de expressão seletora não-representados (a expressão que assume um valor que não apareceu em nenhuma das listas de constantes). Dizia-se que essas ocorrências causavam resultados indefinidos. Tal incerteza, entretanto, significava que o problema era simplesmente ignorado. O ANSI/IEEE Pascal

Standard (Ledgard, 1984) posterior é mais concreto; ele especifica que essas ocorrências são erros, que presumivelmente devem ser detectados e registrados durante a execução pelo código gerado pelos compiladores Pascal.

Muitos dialetos do Pascal incluem, agora, uma cláusula opcional para ser executada quando o valor da expressão não aparecer em nenhuma lista de constantes na **case**, como na seguinte:

```

case indice of
  1, 3: begin
    impar := impar + 1;
    somaímpar := somaímpar + indice
  end;
  2, 4: begin
    par := par + 1;
    somapar := somapar + indice
  end
else writeln('Erro na instrução case, indice =', indice)
end

```

Quando *indice* não estiver na faixa de 1 a 4 e a instrução **case** for executada, a mensagem de erro será exibida.

Note que a cláusula **else** não precisa ser usada exclusivamente para condições de erro. Às vezes, também é conveniente usá-la para a condição normal e usar as outras cláusulas para as circunstâncias incomuns.

A semântica operacional é uma maneira efetiva de descrever a semântica de algumas construções de controle. Para tal finalidade, ampliamos a semântica operacional introduzida no Capítulo 3 para incluirmos instruções de atribuição com expressões gerais, como LDs. Também permitimos descrições em português de algumas operações. Essas descrições aparecerão entre colchetes. Por fim, permitimos a inclusão de instruções de saída da linguagem que está sendo descrita.

Uma descrição semântica operacional da instrução **case** precedente é apresentada a seguir:

```

if indice = 1 then goto um_tres;
if indice = 3 then goto um_tres;
if indice = 2 then goto dois_quatro;
if indice = 4 then goto dois_quatro;
writeln('Erro na instrução case, indice = ', indice);
goto fora;
um_tres:
  impar := impar + 1;
  somaímpar := somaímpar + indice;
  goto fora;
dois_quatro:
  par := impar + 1;
  somapar := somapar + indice;
  goto fora;
fora: ...

```

A construção seletora múltipla do C, **switch**, que também aparece no C++ e no Java, é um projeto relativamente primitivo. Sua forma geral é

```

switch (expressão) {
    case expressão_constante_1: instrução_1;
    ...
    case expressão_constante_n: instrução_n;
    [default: instrução_n + 1]
}

```

em que a expressão de controle e as expressões de constantes são do tipo inteiro. As instruções selecionáveis podem ser seqüências de instruções, instruções compostas ou blocos.

A **switch** encapsula os segmentos de código selecionáveis, como acontece com a **case** Pascal, mas ela não rejeita entradas múltiplas e não oferece desvios implícitos no final desses segmentos de código. Isso permite que o controle flua por mais de um segmento de código selecionável em uma única execução. Considere o exemplo seguinte, similar à construção **case** Pascal acima:

```

switch (índice) {
    case 1:
    case 3: ímpar += 1;
              somaímpar += índice;
    case 2:
    case 4: par += 1;
              somapar += índice;
    default: printf("Erro na switch, índice = %d\n", índice);
}

```

Esse código imprime a mensagem de erro em toda execução. De maneira semelhante, executa-se o código para as constantes 2 e 4 todas as vezes que executa-se o código nas constantes 1 e 3. Para separar logicamente esses segmentos, um desvio explícito deve ser usado. O C inclui uma instrução **break** para sair tanto da **switch** como dos corpos das estruturas de laço. A instrução **break** é, de fato, uma goto restringida.

A seguinte construção **switch** C usa **break** e possui a semântica do exemplo **case** Pascal acima:

```

switch (índice) {
    case 1:
    case 3: ímpar += 1;
              somaímpar += índice;
              break;
    case 2:
    case 4: par += 1;
              somapar += índice;
              break;
    default: printf("Erro na switch, índice = %d\n", índice);
}

```

Ocasionalmente, convém permitir que o controle flua de um segmento de código selecionável para outro. Essa é, evidentemente, a razão pela qual não há desvios implícitos na construção **switch**. O problema de confiabilidade com tal projeto surge quando a ausência errônea de uma instrução **break** em um segmento permite que o controle flua incorretamente para o segmento seguinte. Os projetistas da **switch** do C, iguais aos da **GOTO** computada do FORTRAN, optaram por trocar alguma redução de confiabilidade por algum

aumento de flexibilidade. Estudos têm mostrado, entretanto, que a capacidade de fazer o controle fluir de um segmento selecionável para outro raramente é usada. A **switch** do C é modelada na instrução de seleção múltipla do ALGOL 68, que também não possui desvios implícitos de segmentos selecionáveis.

A **case** Ada permite subfaixas, como por exemplo 10..15, e também operadores OR especificados pelo símbolo |, como em 10 | 15 | 20 nas listas de constantes. Uma cláusula **others** está disponível para valores não-representados. A restrição adicional da Ada sobre as listas de constantes serem exaustivas oferece um pouco mais de confiabilidade porque impede o erro de omissão inadvertida de um ou de mais valores de constantes. Somente os tipos inteiro e enumerado são permitidos para a expressão **case**. A maioria das instruções **case** Ada incluem uma cláusula **others** porque é a melhor maneira de assegurar que a lista de constantes seja exaustiva.

A CASE do FORTRAN 90 é semelhante à da Ada.

Em muitas situações, uma construção **case** é inadequada para seleção múltipla. Por exemplo, quando seleções devem ser feitas baseando-se em uma expressão booleana em vez de em algum tipo ordinal, seleções bidirecionais aninhadas podem ser usadas para simular um seletor múltiplo. Para aliviar a má legibilidade de seletores bidirecionais aninhados profundamente, algumas linguagens, como o FORTRAN 90, a Perl e a Ada, foram ampliadas especificamente para esse uso. A extensão permite que algumas das palavras especiais sejam deixadas de lado. Em particular, sequências **else-if** são substituídas por uma única palavra especial, e retira-se a palavra especial de fechamento do **if** aninhado. O seletor aninhado passa, então, a chamar-se **cláusula elsif**. Considere a seguinte construção de seletor Ada:

```
if CONT < 10 then BAG1 := TRUE;
elseif CONT < 100 then BAG2 := TRUE;
elseif CONT < 1000 then BAG3 := TRUE;
end if;
```

que é equivalente à seguinte:

```
if CONT < 10 then
    BAG1 := TRUE;
else
    if CONT < 100 then
        BAG2 := TRUE;
    else
        if CONT < 1000 then
            BAG3 := TRUE;
        end if;
    end if;
end if;
```

A versão **elsif** é a mais legível das duas. Note que esse exemplo não é facilmente simulado com uma instrução **case** porque cada instrução selecionável é escolhida em função de uma expressão booleana. Portanto, a construção **elsif** não é uma forma redundante de **case**. De fato, nenhum dos seletores múltiplos nas linguagens contemporâneas é tão geral quanto a simulação **if-then-elsif**. Uma descrição semântica operacional de uma instrução seletora com cláusulas **elsif**, nas quais os Es são expressões lógicas e os Ss são instruções, é apresentada abaixo:

```

if E1 goto 1
if E2 goto 2
...
1 : S1
    goto fora
2: S2
    goto fora
...
fora: ...

```

A partir dessa descrição, podemos ver a diferença entre as estruturas de seleção múltipla e as construções **elsif**: em uma construção de seleção múltipla, todos os Es restringiriam-se a comparações entre o valor de uma única expressão e alguns outros valores.

Linguagens que não incluem a construção **elsif** podem usar a mesma estrutura de controle, somente com um pouco mais de digitação.

As construções **elsif** baseiam-se na construção matemática comum: a expressão condicional. As linguagens de programação funcionais, discutidas no Capítulo 15, muitas vezes, usam expressões condicionais como uma de suas construções de controle básicas.

## 8.4 Instruções Iterativas

Uma **instrução iterativa** faz com que uma instrução ou uma coleção de instruções seja executada zero, uma ou mais vezes. Todas as linguagens de programação a partir da Plankalkül incluíram algum método de repetir a execução de segmentos de código. A iteração é a própria essência do poder dos computadores. Se ela não fosse possível, os programadores seriam obrigados a declarar cada ação em seqüência; os programas úteis seriam enormes, inflexíveis, demorariam uma quantidade enorme de tempo para serem escritos e demandariam enorme capacidade de memória para serem armazenados.

A execução repetida de uma instrução em uma linguagem funcional, muitas vezes, é realizada por meio de recursão em vez de por construções iterativas. A recursão em linguagens funcionais será discutida no Capítulo 15.

As primeiras construções iterativas nas linguagens de programação estavam diretamente relacionadas com matrizes. Isso resultava do fato de que, nos primórdios da era dos computadores, a computação era esmagadoramente numérica por natureza, freqüentemente usando laços para processar dados em matrizes.

Diversas categorias de instruções de controle de iteração foram desenvolvidas. As principais categorias são definidas pela maneira como os projetistas respondiam a duas questões de projeto básicas:

- Como a iteração é controlada?
- Onde o mecanismo de controle deve aparecer no laço?

As principais possibilidades para controle de iteração são a lógica, a contagem ou uma combinação das duas. As principais opções para a localização do mecanismo de controle são a parte superior ou a parte inferior do laço. Partes superior e inferior aqui, são denotações lógicas, não-físicas. A questão não é a colocação física do mecanismo de controle; ao contrário, é se o mecanismo executado pode afetar o controle antes ou depois da

execução do corpo do laço. Uma terceira opção permite que o usuário decida onde colocar o controle e será discutida na Seção 8.4.3. O **corpo** de um laço é a coleção de instruções cuja execução é controlada pela instrução de iteração. Usamos o termo **pré-teste** querendo dizer que o teste para finalização do laço ocorre antes que o corpo do laço seja executado; e **pós-teste** significando que ele ocorre depois que o corpo do laço é executado. A instrução de iteração e o corpo do laço associado formam, juntos, uma **construção de iteração**.

Além das principais instruções de iteração, discutimos uma forma alternativa contida em uma classe por si mesma: o controle de iterações definido pelo usuário.

### 8.4.1 Laços Controlados por Contador

Uma instrução iterativa de controle de contagem tem uma **variável de laço**, na qual o valor da contagem é mantido. Ela também inclui alguns meios de especificar os valores **inicial** e **terminal** da variável de laço e a diferença entre seus valores seqüenciais, freqüentemente chamado **tamanho do passo**. A especificação inicial, a terminal e o tamanho do passo de um laço são chamados de **parâmetros do laço**.

Embora os laços controlados logicamente sejam mais gerais do que os controlados por contador, eles não são necessariamente mais comumente usados. Uma vez que os laços controlados por contador são mais complexos, o projeto deles é mais exigente.

Laços controlados por contador freqüentemente são suportados por instruções de máquina. Infelizmente, a arquitetura de máquina, muitas vezes, sobrevive às abordagens dominantes para a programação na época em que é feito o projeto da arquitetura. Por exemplo, os computadores VAX têm uma instrução muito mais conveniente para a implementação de laços pós-teste controlados por contador, do que o FORTRAN possuía na época do projeto do computador VAX. Mas o FORTRAN não tinha mais esse laço na época em que os computadores VAX popularizaram-se.

Obviamente, também é verdade que as construções de linguagem sobrevivem à arquitetura de máquina. Por exemplo, o autor não conhece nenhuma máquina contemporânea que tenha uma instrução de desvio tridirecional para implementar a instrução IF aritmética do FORTRAN.

#### 8.4.1.1 Questões de Projeto

Há muitas questões de projeto referentes às instruções iterativas controladas por contador. A natureza da variável de laço e os seus parâmetros apresentam uma série de questões de projeto. O tipo da variável de laço e os seus parâmetros, obviamente, devem ser os mesmos ou pelo menos compatíveis, mas quais tipos devem ser permitidos? Uma escolha aparente é o tipo inteiro, mas, e os tipos enumeração, caractere e reais? Outra questão é se a variável de laço é normal, em termos de escopo, ou se ela deve ter algum escopo especial. Esta questão está relacionada com o valor da variável de laço depois da finalização dele. Permitir que o usuário mude a variável de laço ou os seus parâmetros dentro dele pode levar a um código muito difícil de entender, de modo que outra questão é se a flexibilidade adicional que poderia ser obtida ao permitir tais mudanças vale essa complexidade adicional. Surge uma questão semelhante a respeito do número de vezes e do tempo específico em que os parâmetros de laços são avaliados: se eles forem avaliados apenas uma vez, isso resultará em laços mais simples, mas menos flexíveis.

A seguir apresentamos um resumo destas questões de projeto:

- Qual é o tipo e o escopo da variável de laço?
- Que valor a variável de laço tem na sua finalização?
- Deve ser legal que a variável de laço ou os seus parâmetros sejam mudados no laço e, se assim for, a mudança afeta o seu controle?
- Os parâmetros de laço devem ser avaliados somente uma vez, ou uma vez para cada iteração?

#### 8.4.1.2 A Instrução DO do FORTRAN 90

O FORTRAN I incluía uma instrução iterativa de controle de contagem, que permaneceu a mesma nos FORTRAN II e IV. A característica peculiar dessa instrução é que ela era pós-teste, tornando-a diferente das instruções iterativas de contagem existentes em todas as outras linguagens de programação. Sua forma geral é

`DO rótulo variável = inicial, final [, tamanho do passo]`

em que o rótulo é o da última instrução no corpo do laço e o tamanho do passo, quando ausente, é fixado em 1 como padrão. Os parâmetros de laço restringem-se a constantes de números inteiros sem sinal ou a variáveis de números inteiros simples com valores positivos.

A forma da instrução DO do FORTRAN 90 é semelhante à do FORTRAN IV, exceto que ela é pré-teste. Embora a FORTRAN 77 permitisse que a variável de laço pudesse ser do tipo INTEGER, REAL ou DOUBLE PRECISION, no FORTRAN 90 ela deve ser do tipo INTEGER. Permite-se que os parâmetros de laço sejam expressões e possam ter valores positivos ou negativos. Eles são avaliados no início da execução da instrução DO e o valor é usado para computar uma **contagem de iteração**, que tem o número de vezes que o laço deve ser executado. O laço é controlado pela contagem de iteração, não pelos seus parâmetros; assim, mesmo que os parâmetros sejam mudados no laço, o que é legal, essas mudanças não poderão afetar o seu controle. A contagem de iteração é uma variável interna inacessível ao código do usuário.

As construções DO somente podem ser penetradas pela sua instrução, transformando-a, assim, em uma estrutura de entrada única. Quando uma instrução DO encerra-se — independentemente de como se encerra — a variável de laço tem seu valor mais recentemente atribuído. Dessa forma, a sua utilidade independe do método pelo qual ele encerra-se. Uma descrição semântica operacional da instrução DO do FORTRAN 90 é apresentada a seguir:

```

valor_inicial = expressão_inicial
valor_final = expressão_final
valor_passo = expressão_passo
var_do = valor_inicial
cont_iteração =
    max(int((valor_final - valor_inicial + valor_passo) / valor_passo), 0)
laço:
    if cont_iteração ≤ 0 goto fora
    [corpo do laço]
    var_do := var_do + valor_passo
    cont_iteração := cont_iteração - 1
    goto laço
falsa: ...

```

O FORTRAN 90 inclui uma outra forma:

```
[nome:] DO variável = inicial, final [, tamanho do passo]
...
END DO [nome]
```

Essa instrução DO usa uma palavra (ou frase) especial de fechamento específica, END DO, em vez de uma instrução rotulada.

#### 8.4.1.3 A Instrução for do ALGOL 60

Aqui, uma descrição da instrução **for** do ALGOL 60 é incluída para mostrar como a busca de flexibilidade pode levar rapidamente a excessiva complexidade. A instrução **for** do ALGOL 60 é uma generalização significativa da instrução DO do FORTRAN, como é mostrado em sua descrição EBNF:

```
<for_stmt> → for var := <elemento_da_lista> {, <elemento_da_lista>} do
    <instrução>
<elemento_da_lista> → <expressão>
    | <expressão> step <expressão> until <expressão>
    | <expressão> while <expr_boleana>
```

Uma diferença significativa entre esse e a maioria dos outros laços controlados por contador é que tal construção pode combinar um contador e uma expressão booleana para controle do laço. As três formas mais simples são exemplificadas pelo seguinte:

```
for cont := 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 do
    lista[cont] := 0

for cont := 1 step 1 until 10 do
    lista[cont] := 0

for cont := 1, cont + 1 while (cont <= 10) do
    lista[cont] := 0
```

Essa instrução é bem mais complexa quando se combinam suas diferentes formas simples, como na seguinte:

```
for indice := 1, 4, 13, 41
    step 2 until 47,
    3 * indice while indice < 1000,
    34, 2, -24 do
        soma := soma + indice
```

A instrução adiciona os seguintes valores a soma:

```
1, 4, 13, 41, 43, 45, 47, 147, 441, 34, 2, -24
```

Embora possa haver ocasiões em que essa complicada instrução seja conveniente, essas ocasiões podem ocorrer muito raramente para justificar a inclusão desse tipo de complexidade em uma linguagem.

A instrução **for** do ALGOL 60 é ainda mais difícil de entender do que parece a princípio, porque todas as expressões nas listas **for** são avaliadas para cada iteração ou para cada execução das instruções do laço. Dessa forma, se uma expressão **step** incluir uma referência à variável **cont**, por exemplo, e se as instruções de laço mudarem o valor de

cont, o tamanho do passo irá modificar-se com cada iteração. Por exemplo, considere o laço

```
i := 1;
for cont := 1 step cont until 3 * i do
    i := i + 1
```

A instrução **for** faz com que a atribuição ( $i := i + 1$ ) seja executada repetidamente enquanto **cont** duplique-se a cada iteração (porque o passo - **step** - é sempre o valor anterior de **cont**). Uma vez que **cont** fica cada vez mais rápido do que a expressão da cláusula **until** ( $3 * i$ ), o laço não é infinito, ainda que isso não seja evidente à primeira vista. Os valores das variáveis e as expressões de controle para esse laço são os seguintes:

i	cont	passo	until
1	1	1	3
2	2	2	6
3	4	4	9
4	8	8	12
5	16	16	15 - fim do laço

As opções de projeto da **for** ALGOL 60 são as seguintes: a variável de laço pode ser do tipo inteiro ou real, e é declarada como qualquer outra, de modo que seu escopo seja o de sua declaração. Como acontece no FORTRAN 90, a variável de laço tem seu valor mais recentemente atribuído depois da finalização do laço, independentemente da causa dessa finalização. Seus parâmetros, mas não sua variável, podem ser mudados no corpo do laço. É ilegal desviar para o corpo do laço, e os parâmetros de laço são avaliados para cada iteração.

Não é viável apresentar uma descrição semântica operacional da instrução **for** do ALGOL 60 completa com todas suas opções. Em vez disso, apresentamos uma descrição de uma instrução **for** geral somente com a forma **step-until**:

```
var_for := expr_inicial
laço:
    until := expr_until
    passo := expr_passo
    temp := (var_for - until) * SIGN(passo)
    if temp > 0 goto fora
    [corpo do laço]
    var_for := var_for + passo
    goto laço
falsa: ...
```

A seguir, apresentamos uma descrição semântica operacional de um exemplo mais complexo de uma instrução **for**:

```
for cont := 10 step 2 * cont until init * init,
            3 * cont while soma <= 1000 do
    soma := soma + cont

    cont := 10
laçol:
    until = init * init
    passo = 2 * cont
```

```

temp = cont - until
if temp > 0 goto laço2
soma := soma + cont
cont := cont + passo
goto laço1
laço2:
    cont := 3 * cont
    if soma > 1000 goto fora
    soma := soma + cont
    goto laço2
falsa: ...

```

#### 8.4.1.4 A Instrução **for** Pascal

A instrução **for** Pascal é o modelo de simplicidade. Sua forma é

```
for variável := valor_inicial (to | downto) valor_final do instrução
```

A opção **to** | **downto** permite que o valor da variável cresça ou se reduza em passos de 1. As opções de projeto para a instrução **for** do Pascal são as seguintes: a variável de laço deve ser de um tipo ordinal e tem o escopo de sua declaração. Em uma finalização de laço normal, sua variável é indefinida. Se o laço for finalizado prematuramente, ela terá seu último valor. A variável de laço não pode ser mudada no corpo do laço. Os valores iniciais e finais, que podem ser expressões de qualquer tipo compatível com o da variável de laço, podem ser mudados nele, mas, à medida que são avaliados somente uma vez, não podem afetar seu controle.

#### 8.4.1.5 A Instrução **for** Ada

A instrução **for** Ada é semelhante à versão Pascal. Ele é um laço pré-teste com a forma

```
for variável in [reverse] faixa_discreta loop
    ...
end loop
```

Uma faixa discreta é uma subfaixa de um tipo inteiro ou de enumeração, como, por exemplo, `1..10`.

O novo recurso mais interessante da instrução **for** da Ada é o escopo da variável de **loop**, o qual é a extensão do laço. A variável é declarada implicitamente na instrução **for** e implicitamente liberada depois da finalização do laço. Por exemplo, em

```
CONT : FLOAT := 1.35;
for CONT in 1..10 loop
    SOMA := SOMA + CONT;
end loop
```

a variável `CONT` do tipo `FLOAT` não é afetada pelo laço **for**. Após a finalização do laço, a variável `CONT` ainda é do tipo `FLOAT` com o valor 1.35. Além disso, a variável `CONT` do tipo `FLOAT` é oculta no código no corpo do laço, sendo mascarada pelo contador de laços `CONT`, implicitamente declarado para ser o tipo da faixa discreta, `INTEGER`.

Não pode ser atribuído um valor à variável de laço Ada no corpo do laço. As variáveis usadas para especificar a faixa discreta podem ser mudadas no laço, mas, como a faixa é avaliada somente uma vez, essas mudanças não afetam o controle do laço. Não é válido

desviar para o corpo do laço **for** Ada. Abaixo, apresentamos uma descrição semântica operacional do laço **for** Ada:

```
[defina var_for (seu tipo é o da faixa discreta)]
[avalie a faixa discreta]
laço:
  if [não há quaisquer elementos à esquerda da faixa discreta] goto fora
  var_for := [o próximo elemento da faixa discreta]
  [corpo do laço]
  goto laço
falsa:
  [acabe com a definição de var_for]
```

#### 8.4.1.6 A Instrução **for** do C, do C++ e do Java

A forma geral da instrução **for** do C é

```
for (expressão_1; expressão_2; expressão_3)
  corpo do laço
```

O corpo do laço pode ser uma única instrução, uma instrução composta ou uma instrução nula.

Uma vez que as instruções em C produzem resultados e, dessa forma, podem ser consideradas expressões, estas, em uma instrução **for**, freqüentemente são instruções. A primeira expressão serve para a inicialização e é avaliada somente uma vez, quando se inicia a execução da instrução **for**. A segunda expressão é o controle do laço, avaliada antes de cada execução do corpo deste. Como é comum no C, um valor igual a zero significa falso e valores diferentes de zero significam verdadeiro. Portanto, se o valor da segunda expressão **for** igual a zero, a instrução **for** será finalizada; caso contrário, as instruções de laço serão executadas. A última expressão na instrução **for** é executada depois de cada execução do corpo do laço. Muitas vezes ela é usada para incrementar o contador de laços. Uma descrição semântica operacional da instrução **for** C é mostrada a seguir. Uma vez que as expressões C também são instruções, mostramos avaliações de expressões como instruções.

```
expressão_1
laço:
  if expressão_2 = 0 goto fora
  [corpo do laço]
  expressão_3
  goto laço
falsa: ...
```

Um laço de contagem típico do C é:

```
for (índice = 0; índice <= 10; índice++)
  soma = soma + lista[índice];
```

Todas as expressões da instrução **for** do C são opcionais. Considera-se verdadeira uma segunda expressão ausente, de modo que uma **for** sem esta expressão é potencialmente um laço infinito. Se a primeira e/ou terceira expressões estiverem ausentes, não se faz nenhuma pressuposição. Por exemplo, se a primeira expressão estiver ausente, simplesmente significa que nenhuma inicialização está sendo levada a efeito.

Note que a **for** do C não precisa de um contador. Ela pode modelar facilmente estruturas de laço de contagem e lógicas, conforme será demonstrado na próxima seção.

As opções de projeto da instrução **for** da linguagem C são as seguintes: não há nenhuma variável de laço explícita ou seus parâmetros. Todas as variáveis envolvidas podem ser mudadas no corpo do laço. As expressões são avaliadas na ordem declarada acima. Apesar do fato de poder gerar confusão, é válido desviar-se para um corpo de laço **for** C.

A instrução **for** do C é mais flexível do que a de outras linguagens que discutimos porque cada uma das expressões pode compreender instruções múltiplas, o que, por sua vez, permite variáveis de laço múltiplas que podem ser de qualquer tipo. Quando instruções múltiplas são usadas em uma única expressão de uma instrução **for**, elas são separadas por vírgulas. Todas as instruções C têm valores, e essa forma de instrução múltipla não é exceção. O valor de instruções múltiplas é o valor do último componente:

Considere a seguinte instrução **for**:

```
for (cont1 = 0, cont2 = 0.0;
    cont1 <= 10 && cont2 <= 100.0;
    soma = ++cont1 + cont2, cont2 *= 2.5);
```

A descrição semântica operacional dessa é

```
cont1 = 0
cont2 = 0.0
laço:
if cont1 > 10 goto fora
if cont2 > 100.0 goto fora
cont1 = cont1 + 1
soma = cont1 + cont2
cont2 = cont2 * 2.5
goto laço
falsa: ...
```

A instrução **for** C acima não precisa e, assim, não tem um corpo de laço. Todas as ações desejadas vêm a fazer parte da própria instrução **for**, não de seu corpo. A primeira e a terceira expressões são instruções múltiplas. Em ambos os casos, a expressão inteira é avaliada, mas o valor resultante não é usado no controle do laço.

A instrução **for** do C++ difere da instrução do C de duas maneiras. Primeiro, além da expressão aritmética, ela pode usar uma expressão **booleana** para controle do laço. Segundo, a primeira expressão pode incluir definições de variáveis. Por exemplo,

```
for (int cont = 0; cont < comp; cont++) { ... }
```

O escopo de uma variável definida na instrução **for** vai de sua definição até o final do corpo do laço.

A instrução **for** do Java é semelhante à do C++, exceto que a expressão de controle de laço se restringe ao tipo **booleano**.

#### 8.4.2 Laços Controlados Logicamente

Em muitos casos, coleções de instruções devem ser executadas repetidamente, mas o controle de execução baseia-se em uma expressão booleana em vez de em um contador. Para essas situações, um laço controlado logicamente é conveniente. De fato, estes são mais

gerais do que os controlados por contador. Todo laço de contagem pode ser construído por meio de um laço lógico, mas a recíproca não é verdadeira. Lembre-se também de que somente instruções de seleção e laços lógicos são essenciais para expressar a estrutura de controle de qualquer fluxograma.

#### 8.4.2.1 Questões de Projeto

Uma vez que são muito mais simples do que os laços controlados por contador, os controlados logicamente têm uma lista relativamente curta de questões de projeto, uma das quais também é uma questão relativa aos controlados por contador. Há pouca coisa aqui que seja controversa ou difícil.

- O controle deve ser pré-teste ou pós-teste?
- O laço controlado logicamente deve ser uma forma especial de laço de contagem ou uma instrução separada?

#### 8.4.2.2 Exemplos

Algumas linguagens imperativas — por exemplo, o Pascal, o C, o C++ e o Java — incluem tanto laços pré-teste como pós-teste controlados logicamente que não são formas especiais de suas instruções iterativas controladas por contador. No C++, os laços lógicos pré-teste e pós-teste têm as seguintes formas:

```
while (expressão)
    corpo do laço
e
do
    corpo do laço
while (expressão)
```

Essas duas formas de instrução são exemplificadas pelos seguintes segmentos de código C++:

```
soma = 0;
cin >> indat;
while (indat >= 0) {
    soma += indat;
    cin >> indat;
}

cin >> valor;
do {
    valor /= 10;
    digitos++;
} while (valor > 0);
```

Note que todas as variáveis desses exemplos são do tipo inteiro; `cin` é o *stream*\* de entrada padrão (o teclado) e `>>` é o operador de entrada.

\*N. de T. Stream: é um canal por meio do qual os dados fluem de/para o disco, o teclado, a impressora, etc.

Na versão pré-teste (**while**), a instrução é executada contanto que a expressão seja avaliada como verdadeira. Na instrução pós-teste (**do**) do C, do C++ e do Java, o corpo do laço é executado até que a expressão seja avaliada como falsa. A única diferença real entre o **do** e o **while** é que o primeiro sempre faz com que o corpo do laço seja executado pelo menos uma vez. Em ambos os casos, a instrução pode ser composta. As descrições semânticas operacionais dessas duas instruções são apresentadas abaixo:

<pre><b>while</b>   laço:     if expressão = 0 <b>goto</b> fora     [corpo do laço]     <b>goto</b> laço   fora: ...</pre>	<pre><b>do-while</b>   laço:     [corpo do laço]     if expressão ≠ 0 <b>goto</b> laço</pre>
--	--

É válido no C e no C++ desviar-se tanto para corpos de laço **while** como **do**.

As instruções **while** e **do** do Java são similares às do C e do C++, exceto que a expressão de controle deve ser do tipo **booleano** e, uma vez que o Java não tem **goto**, não se pode entrar nos corpos de laço, a não ser no início.

O FORTRAN 90 não tem nem um laço lógico pré-teste nem um pós-teste. A Ada tem um laço lógico pré-teste, mas não uma versão pós-teste do laço lógico.

A Perl tem dois laços pré-teste **while** e **until** e nenhum pós-teste. O **until** é similar ao **while**, mas usa o inverso do valor da expressão de controle.

A instrução de laço lógico pós-teste do Pascal, **repeat-until**, difere da **do-while** do C, do C++ e do Java em termos de que a lógica da expressão de controle é trocada. O corpo do laço é executado enquanto a expressão de controle está sendo avaliada como falsa, em vez de verdadeira, como acontece com o C, com o C++ e com o Java.

A instrução **repeat-until** é ímpar, porque seu corpo pode ser ou uma instrução composta ou uma sequência de instruções. Ela é a única estrutura de controle em Pascal que tem tal flexibilidade. Eis outro exemplo de falta de ortogonalidade no projeto do Pascal.

Laços pós-teste são infreqüentemente úteis e também podem ser bastante perigosos em razão de que os programadores, às vezes, esquecem de que o corpo do laço *sempre* será executado pelo menos uma vez. O projeto sintático de colocar um controle pós-teste depois do corpo do laço, onde ele tem seu efeito semântico, ajuda a evitar esses problemas ao tornar a lógica clara.

#### 8.4.3 Mecanismos de Controle de Laços Localizados pelo Usuário

Em algumas situações, é conveniente para o programador escolher uma localização para o controle de laço que não seja o seu topo nem a sua base. Como consequência, algumas linguagens oferecem essa capacidade. Um mecanismo sintático para controle de laço localizado pelo usuário pode ser relativamente simples, de modo que seu projeto não é difícil. Talvez a questão mais interessante seja se um único ou diversos laços aninhados podem ser finalizados. As questões de projeto referentes a esse mecanismo são as seguintes:

- O mecanismo condicional deve ser uma parte integrante da saída?
- Deve-se permitir que o mecanismo apareça em um laço controlado ou somente em um laço sem qualquer outro controle?
- Somente um corpo de laço deve ser finalizado, ou laços envolventes também podem ser finalizados?

Diversas linguagens, inclusive a Ada, têm instruções de laço sem nenhum controle de iteração; eles são laços infinitos, a menos que controles sejam adicionados pelo programador. A forma do laço infinito Ada é

```
loop
  ...
end loop;
```

O **exit** Ada pode ser incondicional ou condicional e pode aparecer em qualquer laço. Sua forma geral é

```
exit [rótulo_do_laço] [when condição]
```

Sem nenhuma das partes opcionais, **exit** provoca o encerramento somente do laço no qual ele aparece. Por exemplo, em

```
loop
  ...
  if SOMA >= 10000 then exit; end if;
  ...
end loop;
```

O **exit**, quando executado, transfere o controle para a primeira instrução depois do final do laço.

Um **exit** com uma condição **when** encerra seu laço somente se a condição especificada for verdadeira. Por exemplo, o laço acima pode ser escrito como

```
loop
  ...
  exit when SOMA >= 10000;
  ...
end loop;
```

Qualquer laço pode ser rotulado e, quando um rótulo dele é incluído no **exit**, transfere-se o controle para a instrução imediatamente seguinte ao laço referenciado. Por exemplo, considere o seguinte segmento de código:

```
LACO_EXTERNO:
  for LINHA in 1 .. MAX_LINHAS loop
    LACO_INTERNO:
      for COL in 1 .. MAX_COLS loop
        SOMA := SOMA + MAT(LINHA, COL);
        exit LACO_EXTERNO when SOMA > 1000.0;
      end loop LACO_INTERNO;
    end loop LACO_EXTERNO;
```

Nesse exemplo, o **exit** é um desvio condicional para a primeira instrução depois do laço externo. Se, em vez disso, o **exit** fosse

```
exit when SOMA > 1000.0;
```

ele seria um desvio condicional para a primeira instrução depois do laço interno. Note que as instruções **exit** freqüentemente são usadas para manipular condições incomuns ou de erro.

O C, o C++ e a Perl têm saídas incondicionais não-rotuladas (**break** no C e no C++ e **last** na Perl); o FORTRAN 90 e o Java têm saídas rotuladas incondicionais (**EXIT** no

FORTRAN 90 e **break** no Java), como a Ada, exceto que na versão Java o alvo pode ser qualquer instrução composta envolvente.

O C e o C++ incluem um mecanismo de controle, **continue**, que o transfere para o mecanismo de controle do laço envolvente de menor tamanho. Isso não é uma saída, mas, ao contrário, uma maneira de pular o resto das instruções de laço na iteração atual sem finalizar a sua estrutura. Por exemplo, considere o seguinte:

```
while (soma < 1000) {
    getnext(valor);
    if (valor < 0) continue;
    soma += valor;
}
```

Um valor negativo faz com que a instrução de atribuição seja pulada e, assim, o controle seja transferido para a condicional no topo do laço. Por outro lado, em

```
while (soma < 1000) {
    getnext(valor);
    if (valor < 0) break;
    soma += valor;
}
```

um valor negativo encerra-o.

O FORTRAN 90, a Perl e o Java têm instruções similares a **continue**, exceto que podem incluir rótulos que especificam qual laço deve ser prosseguido.

Tanto **exit** como **break** prestam-se a saídas múltiplas de laços, o que atrapalha bastante a legibilidade. Entretanto, condições inesperadas que exigem finalização do laço são tão comuns que essa construção justifica-se. Além disso, a legibilidade não é seriamente prejudicada, uma vez que o alvo de todas essas saídas de laço é a instrução depois dele, não simplesmente qualquer lugar no programa. A **break** do Java e a **last** da Perl são exceções, porque seus alvos podem ser qualquer instrução composta envolvente.

#### 8.4.4 Iteração Baseada em Estruturas de Dados

Somente um tipo adicional de estrutura de laço falta ser considerado aqui: a iteração que depende de estruturas de dados. Em vez de fazer um contador ou uma expressão booleana controlar as iterações, esses laços são controlados pelo número de elementos em uma estrutura de dados. A COMMON LISP e a Perl têm essas instruções.

Na COMMON LISP, a função **dolist** itera em listas simples, as quais são a estrutura de dados mais comuns em programas LISP. Por causa dessa restrição, **dolist** é automática em termos de que sempre itera implicitamente nos elementos da lista. Isso acarreta a execução de seu corpo uma vez para cada elemento da lista dada. A instrução **foreach** da Perl é similar; ela itera nos elementos de listas de vetores. Por exemplo,

```
@nomes = ("Bob", "Carol", "Ted", "Beelzebub");
...
foreach $nome (@nomes) {
    print $nome;
}
```

Uma instrução de iteração baseada em dados mais geral usa uma estrutura de dados definida pelo usuário e uma função definida pelo usuário para percorrer os elementos da estrutura. Essa função é chamada **iterador**. Este último é chamado no início de cada iteração, e cada vez que é chamado, retorna um elemento de uma estrutura de dados particular em alguma ordem específica. Por exemplo, suponhamos que o programa tenha uma árvore binária de vértices de dados, e que os dados em cada vértice devam ser processados em certa ordem particular. Uma instrução de iteração definida pelo usuário para a árvore definiria de maneira bem-sucedida a variável de laço para que apontasse para os vértices da árvore, um para cada iteração. A execução inicial da instrução de iteração definida pelo usuário precisa emitir uma chamada especial ao iterador para obter o primeiro elemento da árvore. O iterador sempre deve lembrar-se de qual vértice ele apresentou por fim, para que ele visite todos os vértices sem repetir a visita. Desse modo, um iterador deve ser sensível à história. Uma instrução de iteração definida pelo usuário encerra-se quando o iterador não consegue encontrar mais elementos.

A construção **for** do C, do C++ e do Java, devido à sua grande flexibilidade, pode ser usada para simular uma instrução de iteração definida pelo usuário. Novamente, suponhamos que os vértices de uma árvore binária devam ser processados. Se a raiz da árvore for apontada por uma variável chamada **root**, e se **traverse** for uma função que ajusta seu parâmetro para apontar para o próximo elemento de uma árvore na ordem desejada, o seguinte poderia ser usado:

```
for (ptr = root; ptr != null; traverse(ptr)) {
    ...
}
```

Nessa instrução, **traverse** é o iterador.

As instruções de iteração definidas pelo usuário são mais importantes na programação orientada a objeto do que eram em paradigmas de desenvolvimento de software anteriores. Isso resulta do fato de que os usuários, agora, constroem rotineiramente tipos de dados abstratos para estruturas de dados. Nestes, uma instrução de iteração definida pelo usuário e seu iterador devem ser oferecidos pelo autor da abstração de dados, porque a representação dos objetos do tipo não é conhecida pelo usuário. No C++, os iteradores para tipos definidos pelo usuário ou pelas classes, muitas vezes, são implementados como funções amigas da classe ou como classes de iteradores separadas.

## 8.5 Desvio Incondicional

Uma **instrução de desvio incondicional** transfere o controle da execução para um lugar especificado no programa.

### 8.5.1 Problemas com os Desvios Incondicionais

O mais acalorado debate no projeto de linguagens do final da década de 60 girou sobre a questão de se os desvios incondicionais deveriam fazer parte de qualquer linguagem de alto nível e, assim sendo, se seu uso deveria ser restrito.

O desvio incondicional ou **goto** é a instrução mais poderosa para controlar o fluxo de execução de instruções de um programa. Porém, usar a **goto** descuidadamente pode criar

problemas. Ela tem um poder formidável e uma grande flexibilidade (todas as outras estruturas de controle podem ser construídas com goto e com um seletor), mas esse próprio poder torna seu uso perigoso. Sem restrições ao uso, impostas pelo projeto da linguagem ou por padrões de programação, as instruções goto podem tornar os programas virtualmente ilegíveis e, em consequência, pouco confiáveis e difíceis de serem mantidos.

Esses problemas decorrem diretamente da capacidade que uma goto tem de forçar qualquer instrução de programa a seguir qualquer outra na seqüência de instrução, independentemente de se essa instrução precede ou se vem depois da primeira na ordem textual. A legibilidade é melhor quando a ordem de execução das instruções é quase a mesma em que aparecem — em nosso caso, isso significaria de cima a baixo, que é a ordem à qual estamos acostumados. Desse modo, restringir gotos a fim de que elas possam transferir o controle somente para baixo em um programa, alivia parcialmente o problema. Isso permite que as gotos transfiram o controle em torno de seções de código em resposta a erros ou a condições incomuns, mas rejeita seu uso para construir qualquer tipo de laço.

Ainda que diversas pessoas atentas tenham sugerido anteriormente, foi Edsger Dijkstra que fez, ao mundo da computação, a primeira exposição amplamente lida sobre os perigos da goto. Em seu documento, ele observou: "A instrução goto, como está, é simplesmente primitiva demais; ela é um grande convite para que se transforme o programa de alguém em uma bagunça" (Dijkstra, 1968a). Durante os primeiros anos após a publicação dos pontos de vista de Dijkstra sobre a goto, um grande número de pessoas defendeu publicamente o seu imediato banimento ou, pelo menos, restrições a ela. Entre os que não concordavam com a sua completa eliminação estava Donald Knuth, que argumentava haver ocasiões em que a eficiência da goto superava seu dano à legibilidade (Knuth, 1974).

Algumas linguagens foram projetadas sem a goto — por exemplo, o Modula-2 e o Java. Porém, a maioria das linguagens populares atualmente incluem a instrução goto. Kernighan e Ritchie (1978) chamam a goto de infinitamente sujeita a abusos, mas, ainda assim, ela é incluída na linguagem de Ritchie, o C. As linguagens que eliminaram a goto ofereceram instruções de controle adicionais, usualmente na forma de saídas de laços e de subprogramas, para substituir muitas das suas aplicações típicas.

Todas as instruções de saída de laço discutidas na seção 8.4.3 são na verdade goto camufladas. Contudo, elas são restritas e não prejudicam a legibilidade. De fato, pode ser argumentado que elas aumentam a legibilidade porque sem o seu uso o código ficaria confuso e antinatural, e seria muito penoso para entendê-lo.

### 8.5.2 Formas de Rótulo

Algumas linguagens, como o ALGOL 60 e o C, usam suas formas identificadoras para rótulos. O FORTRAN e o Pascal usam constantes inteiras sem sinal para rótulos. A Ada usa sua forma identificadora como a parte-alvo de sua instrução goto, mas, quando o rótulo aparece em uma instrução, ele deve ser delimitado pelos símbolos <<OUT>>. Por exemplo, considere o seguinte:

```
goto TERMINOU;
...
<<TERMINOU>> SOMA := SOMA + PROXIMO;
```

O uso de colchetes angulados (<< >>) torna os rótulos mais fáceis de serem localizados quando o programa é lido. Na maioria das outras linguagens, os rótulos são anexados a instruções por meio de dois-pontos, como em

```
terminou: soma := soma + proximo
```

Em seu projeto de rótulos, a PL/I novamente leva uma construção ao seu limite de flexibilidade e de complexidade. Em vez de tratá-los como simples constantes, a PL/I permite que eles sejam variáveis. Nesta forma, podem ser atribuídos valores a eles que podem ser usados como parâmetros de subprograma. Isso permite que uma goto seja visada a virtualmente qualquer lugar em um programa, e o alvo não pode ser determinado estaticamente. Ainda que essa flexibilidade, às vezes, seja útil, ela é bem mais prejudicial à legibilidade para valer a pena. Imagine tentar ler e entender um programa com desvios cujos alvos dependem dos valores atribuídos em tempo de execução. Considere um subprograma com diversos rótulos e uma goto cujo rótulo-alvo seja um parâmetro formal. Para determinar o alvo da goto, deve-se saber a unidade de programa que chama e o valor de parâmetro real usado na chamada. A implementação de rótulos variáveis também é complexa, principalmente porque, de todas as maneiras possíveis, variáveis de rótulos podem ser vinculadas a valores.

## 8.6 Comandos Protegidos

Formas alternativas e diferentes de estruturas de seleção e de laço foram sugeridas por Dijkstra (1975). Sua motivação era oferecer instruções de controle que suportassem uma metodologia de projeto de programas que assegurasse a exatidão durante o desenvolvimento em vez de recorrer a verificação ou teste de programas completos para garantir sua exatidão. Essa metodologia é descrita em Dijkstra (1976).

Os comandos protegidos são enfocados neste capítulo, porque formam a base de dois mecanismos lingüísticos desenvolvidos depois para programação concorrente em duas linguagens, a CSP (Hoare, 1978) e a Ada. A concorrência na Ada será discutida no Capítulo 13.

A construção de seleção de Dijkstra tem a forma

```
if <expressão booleana> -> <instrução>
[ ] <expressão booleana> -> <instrução>
[ ] ...
[ ] <expressão booleana> -> <instrução>
fi
```

A palavra reservada de fechamento, **fi**, é a mesma de abertura grafada de trás para a frente. Essa forma de fechamento foi tomada do ALGOL 68. Os pequenos blocos, chamados *fatbars*, são usados para separar as cláusulas protegidas e para permitir que estas sejam seqüências de instruções.

Essa construção de seleção tem a aparência de uma seleção múltipla, mas sua semântica é diferente. Todas as expressões booleanas são avaliadas cada vez que se alcança a construção durante a execução. Se mais de uma expressão for verdadeira, uma das instruções correspondentes será não-deterministicamente escolhida para execução. Se nenhuma for verdadeira, ocorrerá um erro em tempo de execução que causará a finalização do programa. Isso obriga o programador a considerar e a listar todas as possibilidades, como acontece com a instrução **case** da Ada. Considere o seguinte exemplo:

```
if i = 0 -> soma := soma + i
[ ] i > j -> soma := soma + j
[ ] j > i -> soma := soma + i
fi
```

Se  $i = 0$  e  $j > i$ , essa construção escolhe não-deterministicamente entre a primeira e a terceira instruções de atribuição. Se  $i$  for igual a  $j$  e diferente de zero, ocorrerá um erro em tempo de execução porque nenhuma das condições é verdadeira.

Tal construção pode ser uma maneira elegante de permitir que o programador declare que a ordem de execução, em alguns casos, seja irrelevante. Por exemplo, para encontrar o maior de dois números, podemos usar

```
if x >= y -> max := x
[] y >= x -> max := y
fi
```

Isso computa o resultado desejado sem especificar demasiadamente a solução. Em especial, se  $x$  e  $y$  forem iguais, não importa qual deles atribuímos a  $max$ . Essa é uma forma de abstração oferecida pela semântica não-determinística da instrução.

Eis outra construção em que a construção de seleção de Dijkstra é valiosa: suponhamos escrever um programa que faça a manutenção de interrupções, e que as interrupções tenham a mesma prioridade. Para isso, precisamos de uma construção que escolha entre interrupções atuais de alguma maneira aleatória.

A semântica dos comandos protegidos é difícil de ser escrita com precisão. Ainda que os fluxogramas não sejam ferramentas tão boas para projeto de programas, por vezes eles são úteis para descrições da semântica. A Figura 8.1 (ver p. 322) é um fluxograma que descreve a abordagem usada pela instrução seletora de Dijkstra. Note que o fluxograma é relativamente impreciso, refletindo a dificuldade de captar a semântica dos comandos protegidos.

A estrutura de laço proposta por Dijkstra tem a forma

```
do <expressão booleana> -> <instrução>
[] <expressão booleana> -> <instrução>
[]
...
[] <expressão booleana> -> <instrução>
od
```

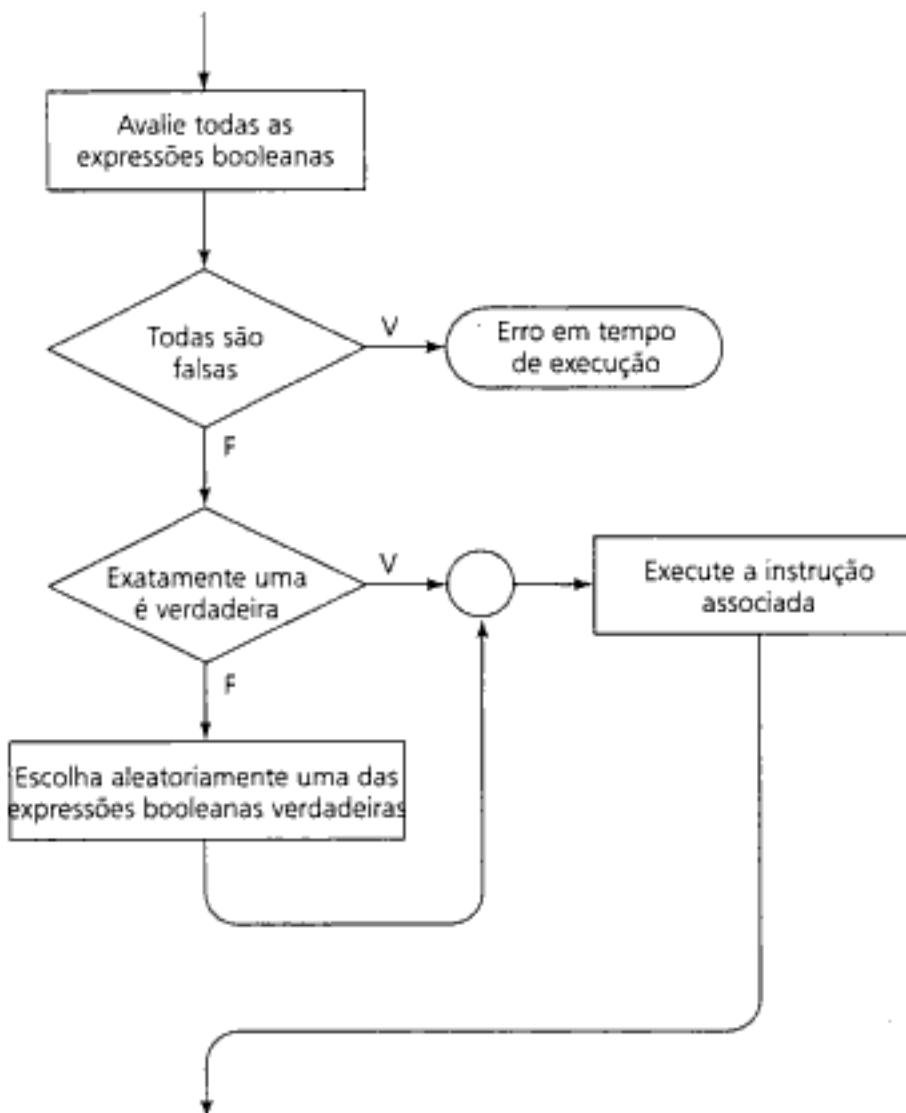
A semântica dessa construção é que todas as expressões booleanas são avaliadas em cada iteração. Se mais de uma for verdadeira, uma das instruções associadas é não-deterministicamente escolhida para execução, sendo que depois as expressões são novamente avaliadas. Quando todas as expressões são simultaneamente falsas, o laço encerra-se.

Considere o seguinte código, que aparece em uma forma ligeiramente diferente em Dijkstra (1975). As quatro variáveis  $q_1$ ,  $q_2$ ,  $q_3$  e  $q_4$  devem ter seus valores reorganizados de forma que  $q_1 < q_2 < q_3 < q_4$ .

```
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
```

Um fluxograma que descreve a abordagem usada pela instrução de laço de Dijkstra é mostrado na Figura 8.2 (ver p. 323). Mais uma vez, note que a semântica do controle de fluxo dessa construção não pode ser completamente descrita em um fluxograma.

Os comandos protegidos de Dijkstra, como são conhecidas essas construções, são interessantes em parte porque ilustram como a sintaxe e a semântica das instruções podem ter um impacto sobre a verificação do programa e vice-versa. A verificação do programa é virtualmente impossível quando instruções goto são usadas. A verificação é grandemente simplificada se somente laços lógicos e seleções, iguais as do Pascal, forem usados ou se



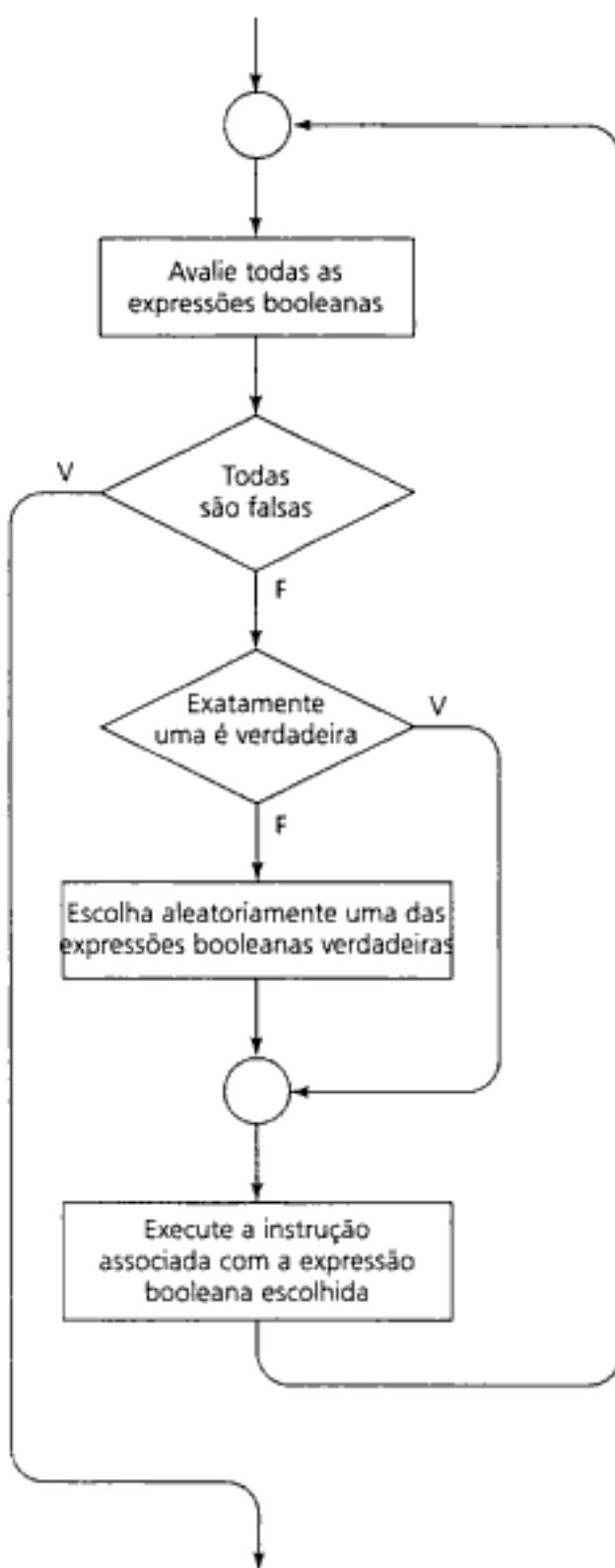
**FIGURA 8.1** Fluxograma da abordagem usada com a instrução seletora de Dijkstra.

somente comandos protegidos forem usados. A semântica axiomática dos comandos protegidos é convenientemente especificada (Gries, 1981). Deve estar claro, entretanto, que há uma complexidade consideravelmente maior na implementação dos comandos protegidos em relação a suas contrapartes determinísticas convencionais.

## 8.7 Conclusões

Descrevemos e discutimos uma variedade de estruturas de controle no nível da instrução. Agora parece caber uma breve avaliação.

Primeiro, temos o resultado teórico de que somente as seqüências, os laços lógicos de pré-teste e as seleções são absolutamente necessários para exprimir computações (Böhm e Jacopini, 1966). Esse resultado tem sido amplamente usado por aqueles que desejam banir completamente os desvios incondicionais. Obviamente, já existem suficientes problemas



**FIGURA 8.2** Fluxograma da abordagem usada com a instrução de laço de Dijkstra.

práticos com a goto para condená-la sem antes descobrir uma razão teórica. Uma aplicação da goto que muitos consideram justificável é seu uso para permitir saídas prematuras de laços em linguagens que não têm instruções de saída.

Um uso errôneo da conclusão de Böhm e de Jacopini é argumentar contra a inclusão de quaisquer estruturas de controle além da seleção e dos laços lógicos de pré-teste. Nenhu-

ma linguagem amplamente usada ainda deu esse passo. Além disso, duvidamos que alguma delas jamais venha a fazê-lo por causa do efeito sobre a capacidade de escrita e de legibilidade. Programas escritos somente com seleção e laços lógicos de pré-teste geralmente são menos naturais em termos de estrutura, mais complexos e, portanto, mais árduos de serem escritos e mais difíceis de serem lidos. Por exemplo, a estrutura de seleção múltipla da Ada é um grande impulso para a capacidade de escrita da Ada, sem nenhum aspecto negativo claro. Outro exemplo é a estrutura de laço de contagem de muitas linguagens, especialmente quando a instrução é simples, como no Pascal e na Ada.

Não é tão claro que a utilidade das outras estruturas de controle propostas valha a inclusão das mesmas nas linguagens (Ledgard e Marcotty, 1975). Essa questão repousa em grande parte na discussão fundamental de se o tamanho das linguagens deve ser minimizado. Tanto Wirth (1975) como Hoare (1973) endossam fortemente a simplicidade no projeto de linguagens. No caso das estruturas de controle, simplicidade significa que somente algumas instruções de controle devem constar em uma linguagem, e todas elas devem ser simples.

Uma rica variedade de estruturas de controle no nível da instrução que foram inventadas mostra a diversidade de opiniões entre os projetistas de linguagens. Depois de toda invenção, de discussão e de avaliação, ainda não há unanimidade de opinião sobre o conjunto preciso de instruções de controle que deve estar em uma linguagem. A maioria das linguagens contemporâneas, é claro, tem instruções de controle similares, mas ainda há alguma variação nos detalhes de sua sintaxe e de sua semântica. Além disso, ainda há discordância em relação a se a linguagem deve incluir ou não uma instrução goto; o C++ e a Ada 95 incluem-na, mas o Java não.

Uma observação final: as estruturas de controle de linguagens de programação funcionais e lógicas e da Smalltalk são bem diferentes das descritas neste capítulo. Esses mecanismos serão discutidos com certo nível de detalhes nos Capítulos 15, 16 e 12, respectivamente.

## RESUMO

As instruções de controle das linguagens imperativas ocorrem em diversas categorias: seleção, seleção múltipla, iterativas e desvio incondicional.

O FORTRAN introduziu um seletor de instrução unidirecional, o IF lógico. O seletor do ALGOL 60 é mais avançado, permitindo a seleção de instruções compostas e incluindo uma cláusula else opcional. Muitas estruturas de controle beneficiaram-se da instrução composta que o ALGOL 60 introduziu.

O IF aritmético do FORTRAN é um seletor tridirecional que normalmente exige desvios incondicionais.

O FORTRAN introduziu duas formas de instrução de seleção múltipla: a go to computada e a go to atribuída. O que é verdade em relação a seus nomes é que ambas são, de fato, desvios de caminhos múltiplos. A case Pascal é representativa das instruções de seleção múltipla modernas; ela inclui tanto encapsulamento dos segmentos selecionáveis como desvios implícitos no final de cada um para o ponto de saída único.

Inventou-se um grande número de instruções de laço para linguagens de alto nível, iniciando-se com a do do FORTRAN. A instrução for do ALGOL 60 era demasiadamente complexa, combinando controles lógicos e de contador em uma única instrução. A instrução for do Pascal, em termos de complexidade, é o oposto. Ela implementa elegantemente somente as formas de laço de contagem mais comumente necessárias. A instrução for do C é a construção de iteração mais flexível.

O C, o C++, o FORTRAN 90, o Java, a Perl e a Ada têm instruções de saída para seus laços; tais instruções ocupam o lugar de um dos usos mais comuns das instruções goto.

Iteradores baseados em dados são construções de laço para processar estruturas de dados, como, por exemplo, listas, hashes e árvores.

O desvio incondicional ou goto tem feito parte da maioria das linguagens imperativas. Seus problemas têm sido amplamente discutidos e debatidos. O consenso atual é que ele deve permanecer na maioria das linguagens, mas que seus perigos devem ser minimizados pela disciplina de programação.

Os comandos protegidos de Dijkstra são construções de controle alternativas com características teóricas positivas. Não obstante eles não terem sido adotados como construções de controle de uma linguagem, parte da semântica aparece nos mecanismos concorrentes da CSP e da Ada.

## QUESTÕES DE REVISÃO

1. Qual é a definição de estrutura de controle?
2. Qual é a definição de bloco?
3. Quais são as questões de projeto relativas às estruturas de seleção?
4. Quais são as soluções comuns para o problema do aninhamento de seletores bidirecionais?
5. Quais são as questões de projeto referentes às instruções de seleção múltipla?
6. Qual é a base para as instruções de controle do FORTRAN I?
7. O que está errado com a instrução IF aritmética do FORTRAN?
8. O que é incomum a respeito da instrução de seleção múltipla do C? Qual compromisso de projeto foi feito?
9. Quais são as questões de projeto referentes às instruções de laço controladas por contador?
10. O que é uma instrução de laço pré-teste? E uma instrução pós-teste?
11. Qual é a mudança mais significativa na instrução DO do FORTRAN IV e do FORTRAN 90?
12. Qual característica da instrução **for** do ALGOL 60 torna os programas que a usam difíceis de serem lidos?
13. Qual é a diferença entre a instrução **for** do C++ e a do Java?
14. Quais são as questões de projeto referentes às instruções de laço controladas logicamente?
15. Qual é a principal razão para que instruções de controle de laço localizadas pelo usuário tenham sido inventadas?
16. Que vantagem a instrução **exit** da Ada tem sobre a instrução **break** do C?
17. Quais são as diferenças entre a instrução **break** do C++ e a do Java?
18. O que é um controle de iteração definido pelo usuário?
19. Quais são as duas desvantagens das variáveis de rótulos da PL/I?
20. Quais linguagens de programação comuns tomam emprestado parte de seu projeto dos comandos protegidos de Dijkstra?

## PROBLEMAS

1. Redija uma defesa da afirmação segundo a qual a instrução de seleção tridirecional do FORTRAN I foi a melhor opção, dadas as circunstâncias da época.
2. Imagine uma situação na qual a variável de rótulo da PL/I seria uma grande vantagem.
3. Descreva três situações em que uma construção de laço de contagem e lógico combinados seja necessária.
4. Compare a GO TO computada do FORTRAN com a instrução **case** Pascal, especialmente em termos de legibilidade e de confiabilidade.
5. Quais são as possíveis razões para que o Pascal tenha um laço pós-teste lógico, enquanto o ALGOL 60 não?

Hidden page

d. Expressões **until** são avaliadas uma vez na entrada do laço, e expressões **step** são avaliadas antes de cada iteração, logo depois que o contador de laços é incrementado.

Em todos os casos, quando mais de uma expressão é avaliada ao mesmo tempo, elas são avaliadas na ordem esquerda para a direita. Além disso, a atribuição é sempre feita assim que seu LD é avaliado.

14. Use o *Science Citation Index* para encontrar um artigo que se refira a Knuth (1974). Leia o artigo e o documento de Knuth e redija um documento que resuma ambos os lados do argumento **goto**.
15. Nesse documento sobre a questão da **goto**, Knuth (1974) sugere uma construção de controle de laço que permite saídas múltiplas. Leia o documento e redija uma descrição semântica operacional da construção.
16. Considere a seguinte instrução **case** Pascal. Reescreva-a usando somente a seleção bidirecional.

```
case indice - 1 of
  2, 4 : par := par + 1;
  1, 3 : impar := impar + 1;
  0 : zero := zero + 1;
  else erro := true
end
```

17. Considere o seguinte segmento de programa C. Reescreva-o sem usar nenhuma **goto** ou **break**:

```
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
}
```

18. Em uma carta ao editor da CACM, Rubin (1987) usa o seguinte segmento de código como prova de que a legibilidade de códigos com **gotos** é melhor do que um código equivalente sem **gotos**. Esse código localiza a primeira linha de uma matriz de números inteiros  $n \times n$  chamada **x** que nada tem a não ser valores zero.

```
for i := 1 to n do
begin
  for j := 1 to n do
    if x[i, j] <> 0
      then goto rejeita;
    writeln('Primeira linha só com zeros é:' , i);
    break;
rejeita:
end;
```

Reescreva esse código sem **gotos** em uma das seguintes linguagens: C, C++, Pascal, Java ou Ada. Compare a legibilidade de seu código com a do código acima.

19. Quais são os argumentos (a favor e contra) com relação ao uso exclusivo de expressões booleanas nas instruções de controle em Java (em oposição a também permitir expressões aritméticas, como no C e no C++)?

Hidden page

# Capítulo 9

## Subprogramas



### Dennis Ritchie

Dennis Ritchie, do Bell Laboratories, foi um dos principais envolvidos no desenvolvimento do UNIX. Ele foi o projetista da primeira versão do C, então usada para reescrever o UNIX para computadores PDP-11.

- 9.1** Introdução
- 9.2** Fundamentos dos Subprogramas
- 9.3** Questões de Projeto Referentes aos Subprogramas
- 9.4** Ambientes de Referência Locais
- 9.5** Métodos de Passagem de Parâmetros
- 9.6** Parâmetros que São Nomes de Subprograma
- 9.7** Subprogramas Sobre carregados
- 9.8** Subprogramas Genéricos
- 9.9** Compilação Separada e Independente
- 9.10** Questões de Projeto Referentes a Funções
- 9.11** Acessando Ambientes Não-locais
- 9.12** Operadores Sobre carregados Definidos pelo Usuário
- 9.13** Co-Rotinas

Subprogramas são blocos de construção fundamentais de programas e, portanto, estão entre os conceitos mais importantes no projeto de linguagens de programação. Exploraremos, agora, o projeto de subprogramas, incluindo métodos de passagem de parâmetros, ambientes de referência locais e não-locais, subprogramas sobrecarregados, genéricos, compilação separada e independente, e os problemas de apelidos e de efeitos colaterais associados aos subprogramas. Incluímos, também, uma breve descrição das co-rotinas, que proporcionam controle simétrico de unidade.

Os métodos de implementação para subprogramas serão discutidos no Capítulo 10.

## 9.1 Introdução

Duas facilidades de abstração fundamentais podem ser incluídas em uma linguagem de programação: abstração de processo e abstração de dados. Nos primórdios das linguagens de programação de alto nível, somente a primeira era reconhecida e incluída. A abstração do processo tem sido um conceito central em todas as linguagens de programação. Na década de 80, entretanto, muitas pessoas começaram a acreditar que a abstração de dados era igualmente importante. Esta última será discutida detalhadamente no Capítulo 11.

O primeiro computador programável, a Máquina Analítica de Babbage, construída na década de 1840, tinha a capacidade de reutilizar coleções de cartões de instrução em diversos lugares diferentes em um programa quando isso era conveniente. Em uma linguagem de programação moderna, essa coleção de instruções é escrita como um subprograma. Tal reutilização resulta em diversos tipos diferentes de economia, de espaço de memória e tempo de codificação. A reutilização também é uma abstração porque os detalhes de computação do subprograma são substituídos em um programa por uma instrução que chama o subprograma. Em vez de explicar como algumas computações são feitas em um programa, essa explicação (a coleção de instruções no subprograma) é ordenada por uma instrução de chamada, efetivamente abstraindo os detalhes. Isso aumenta a legibilidade de um programa ao expor sua estrutura lógica ao mesmo tempo que oculta os detalhes de pequena escala.

## 9.2 Fundamentos dos Subprogramas

### 9.2.1 Características Gerais dos Subprogramas

Todos os subprogramas discutidos neste capítulo, exceto as co-rotinas descritas na Seção 9.13, têm as seguintes características:

- Cada subprograma tem um único ponto de entrada.
- Toda unidade de programa chamadora é suspensa durante a execução do programa chamado, o que implica na existência de somente um subprograma em execução em qualquer momento dado.
- O controle sempre retorna ao chamador quando a execução do subprograma encerra-se.

Não obstante os subprogramas FORTRAN poderem ter múltiplas entradas, esse tipo particular de entrada é relativamente pouco importante, porque não oferece quaisquer capaci-

dades fundamentalmente diferentes. Portanto, neste capítulo, ignoraremos a possibilidade de múltiplas entradas em subprogramas FORTRAN.

As alternativas para as pressuposições acima resultam em co-rotinas (Seção 9.13) e em unidades concorrentes a serem exploradas no Capítulo 13.

Os métodos de linguagens orientadas a objeto estão estreitamente relacionados com os subprogramas discutidos neste capítulo. As principais diferenças nos métodos está na maneira como eles são chamados e na sua associação com classes e com objetos. Ainda que essas características especiais dos métodos sejam discutidas no Capítulo 12, os recursos que eles partilham com os subprogramas, como, por exemplo, parâmetros e variáveis locais, são discutidos neste capítulo.

### 9.2.2 Definições Básicas

Uma **definição de subprograma** descreve a interface e as ações da abstração de subprograma. Uma **chamada a subprograma** é a solicitação explícita para executar o subprograma. Diz-se que um subprograma é **ativo** se, depois de ter sido chamado, ele iniciou a execução, mas ainda não a concluiu. Os dois tipos fundamentais de subprogramas, procedimentos e funções serão discutidos na Seção 9.2.4.

Um **cabeçalho de subprograma**, a primeira linha da definição, serve a diversos propósitos. Primeiro, ele especifica que a unidade sintática seguinte é uma definição de subprograma de algum tipo particular. Essa especificação freqüentemente é realizada com uma palavra especial. Segundo, ele oferece um nome para o subprograma. Em terceiro lugar, pode especificar opcionalmente uma lista de parâmetros. Opcionalmente porque nem todas as definições de subprograma têm parâmetros.

Considere os seguintes exemplos de cabeçalho:

```
SUBROUTINE SOMADORA(parâmetros)
```

Esse é o cabeçalho de um subprograma de sub-rotina FORTRAN chamado SOMADORA. Em Ada, o cabeçalho para SOMADORA seria

```
procedure SOMADORA(parâmetros)
```

Nenhuma palavra especial aparece no cabeçalho de um subprograma C. O C tem somente um tipo de subprograma, a função; e o cabeçalho de uma função é reconhecido pelo contexto em vez de por uma palavra especial. Por exemplo,

```
void somadora(parâmetros)
```

serviria como o cabeçalho de uma função chamada somadora, em que **void** indica que ela não retorna um valor.

O **perfil de parâmetro** de um subprograma é 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 linguagens em que os subprogramas têm tipos, estes são definidos pelo protocolo do subprograma.

Os subprogramas podem ter declarações e definições. Isso coloca em paralelo as declarações e as definições de variáveis no C, em que as primeiras podem ser usadas para oferecer informações sobre tipos, mas não para definir variáveis. Elas são necessárias quando uma variável deve ser referenciada antes que o compilador tenha visto sua definição. As declarações de subprograma oferecem informações sobre interfaces, principalmente tipos de parâmetros, mas não incluem corpos de subprograma. Elas são necessárias quando o compilador precisa converter uma chamada a um subprograma antes de ter visto sua defi-

nição. Tanto nos casos de variáveis como de subprogramas, declarações são necessárias para verificação estática de tipos. As declarações de subprograma são comuns em programas C, onde são chamadas de **protótipos**. Elas também são usadas na Ada e no Pascal, nas quais, às vezes, são chamadas de declarações de encaminhamento (*forward*) ou externas.

O Java não permite declarações de seus métodos porque as referências de encaminhamento a elas são implicitamente permitidas quando elas estão visíveis.

### 9.2.3 Parâmetros

Os subprogramas tipicamente descrevem computações. Há duas maneiras pelas quais um subprograma pode ganhar acesso aos dados que deve processar: pelo acesso direto a variáveis não-locais (declaradas em outro lugar, mas visíveis no subprograma) ou pela passagem de parâmetros. Os dados passados pelos 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 de parâmetro aos dados que deve processar, é uma computação parametrizada. Ele pode executar sua computação em qualquer dado que receba por seus parâmetros (presumindo que os tipos dos parâmetros estejam de acordo com o que é esperado pelo subprograma). Se o acesso a dados for feito por variáveis não-locais, a única maneira possível da computação ocorrer em diferentes dados é atribuindo novos valores àquelas entre as chamadas ao subprograma. O acesso extensivo a variáveis não-locais pode provocar reduzida confiabilidade. As visíveis ao subprograma cujo acesso é desejado, muitas vezes, também acabam sendo visíveis onde o acesso a elas não é necessário. Discutiu-se esse problema no Capítulo 5 e ele será revisto na Seção 9.11.

Em algumas situações, é conveniente ser capaz de transmitir computações, em vez de dados, como parâmetros a subprogramas. Nesses casos, o nome do subprograma que implementa essa computação pode ser usado como um parâmetro. Tal forma de parâmetro será discutida na Seção 9.6. Os parâmetros de dados serão discutidos na Seção 9.5.

Os parâmetros no cabeçalho de subprograma são chamados **parâmetros formais**. Por vezes, recebem o nome de variáveis fictícias, porque não as são no sentido usual: em alguns casos, vinculam-se ao armazenamento somente quando o subprograma é chamado, e essa vinculação freqüentemente se dá através de algumas variáveis do programa.

As instruções de chamada a subprogramas incluem o nome dele e uma lista de parâmetros a serem vinculados aos seus formais. Esses parâmetros são chamados de **reais**. Devem ser distinguidos dos parâmetros formais porque os dois podem ter diferentes restrições em suas formas e, é claro, suas aplicações são bem diferentes.

Em quase todas as linguagens de programação, a correspondência entre parâmetros reais e formais — ou a vinculação de ambos — é feita simplesmente pela posição: o primeiro parâmetro real é vinculado ao primeiro formal e assim por diante. Eles são chamados de **parâmetros posicionais**. Eis um bom método para listas de parâmetros relativamente curtas.

Quando as listas são longas, o escritor do programa pode cometer erros quanto à ordem de parâmetros da lista. Uma solução para o problema é fornecer **parâmetros de palavra-chave**, nos quais o nome do parâmetro formal a que um parâmetro real deve estar vinculado é especificado com este último. A vantagem dos parâmetros de palavra-chave está no fato deles poderem aparecer em qualquer ordem na lista de parâmetros reais. Procedimentos Ada podem ser chamados usando-se este método, como em

```
SOMADOR(COMPRIMENTO => MEU_COMPRIIMENTO,  
         LISTA => MEU_VETOR,
```

```
SOMA => MINHA_SOMA);
```

em que a definição de SOMADOR tem os parâmetros formais COMPRIMENTO, LISTA e SOMA.

A principal desvantagem dos parâmetros de palavra-chave reside no fato de o usuário do subprograma precisar saber os nomes dos formais.

Além dos parâmetros de palavra-chave, a Ada e o FORTRAN 90 permitem os posicionais. Os dois podem ser mesclados em uma chamada, como em

```
SOMADOR(MEU_COMPRIMENTO,
         SOMA => MINHA_SOMA,
         LISTA => MEU_VETOR);
```

A única restrição para isso é que, depois de um parâmetro de palavra-chave aparecer na lista, todos os restantes devem sê-lo também. Isso é necessário porque a posição não pode mais ser bem-definida depois que um parâmetro de palavra-chave apareceu.

No C++, no FORTRAN 90 e na Ada, parâmetros formais podem ter valores-padrão. Um valor-padrão é usado se nenhum parâmetro real for passado ao parâmetro formal no cabeçalho de subprograma. Considere o seguinte cabeçalho de função Ada:

```
function CALC_PAGAMENTO(RENDAS : FLOAT;
                           ISENCOES : INTEGER := 1;
                           TARIFA_IMPOSTO : FLOAT) return FLOAT;
```

O parâmetro ISENCOES pode estar ausente em uma chamada a CALC\_PAGAMENTO; quando ele está ausente, o valor 1 é usado. Não se inclui nenhuma vírgula para um parâmetro real ausente em uma chamada Ada, porque o único valor dessa vírgula seria para indicar a posição do parâmetro seguinte, o que, nesse caso, não é necessário. Isso porque todos os parâmetros reais depois destes ausentes devem ser parâmetros de palavra-chave. Por exemplo, considere a seguinte chamada:

```
PAGAMENTO := CALC_PAGAMENTO(20000.0, TARIFA_IMPOSTO => 0.15);
```

No C++, que não tem nenhum parâmetro de palavra-chave, as regras para parâmetros padrão são necessariamente diferentes. Os parâmetros padrão devem aparecer por último, porque os parâmetros são associados posicionalmente. Uma vez que um parâmetro padrão é omitido em uma chamada, todos os parâmetros formais restantes têm valores-padrão. Um cabeçalho de função C++ para CALC\_PAGAMENTO pode ser escrito da seguinte maneira:

```
float calculo_pagamento(float renda, float tarifa_imposto,
                           int isencoes = 1)
```

Note que os parâmetros são reorganizados de modo que o com um valor-padrão é o último. Um exemplo de chamada à calculo\_pagamento do C++ é

```
pagamento = calculo_pagamento(20000.0, 0.15);
```

Na maioria das linguagens sem valores-padrão para parâmetros formais, o número de parâmetros reais em uma chamada deve coincidir com o dos formais no cabeçalho de definição do subprograma. Porém, no C, no C++, na Perl e no JavaScript, isso não é exigido. Quando há menos parâmetros reais em uma chamada do que formais em uma definição, cabe ao programador garantir que a correspondência, sempre posicional, e a execução do subprograma sejam sensatas.

Não obstante tal projeto, que permite um número variável de parâmetros, ser claramente propenso a erros, ele, às vezes, também é conveniente. Por exemplo, a função prin-

Hidden page

Hidden page

- Se subprogramas puderem ser passados como parâmetros, os tipos de parâmetros são verificados em chamadas aos subprogramas passados?
- Definições de subprograma podem aparecer em outras definições de subprograma?
- Subprogramas podem ser sobre carregados?
- Subprogramas podem ser genéricos?
- A compilação separada ou independente é possível?

Essas questões e exemplos de projeto serão discutidos nas seções seguintes.

## 9.4 Ambientes de Referência Locais

Os subprogramas geralmente têm permissão para definir suas próprias variáveis, definindo, assim, os ambientes de referência locais. Variáveis que são definidas dentro de subprogramas são chamadas **variáveis locais**, porque o acesso a elas normalmente é restrito ao subprograma no qual são definidas.

Na terminologia do Capítulo 5, as variáveis locais podem ser estáticas ou dinâmicas na pilha. Se as variáveis forem dinâmicas na pilha, elas são vinculadas ao armazenamento quando o subprograma inicia a execução e desvinculadas do armazenamento quando ela encerra-se. Há diversas vantagens nas variáveis locais dinâmicas na pilha, sendo que a principal é a flexibilidade que proporcionam ao subprograma. É fundamental que os subprogramas recursivos tenham variáveis locais dinâmicas na pilha. Outra vantagem é que parte do armazenamento para variáveis locais de todos os subprogramas pode ser compartilhada. Esse compartilhamento evidentemente não pode se desenvolver entre subprogramas ativos ao mesmo tempo. Essa não é uma vantagem tão grande como era quando os computadores tinham memórias menores.

As principais desvantagens das variáveis locais dinâmicas na pilha são as seguintes: primeiro, há o custo do tempo necessário para alocar, para inicializar (quando necessário) e para desalocá-las para cada ativação. Segundo, os acessos a elas devem ser indiretos, ao passo que os acessos a variáveis estáticas podem ser diretos. Tal característica de indireção é necessária porque o lugar na pilha onde uma variável local particular residirá somente pode ser determinado durante a execução (veja o Capítulo 10). Na maioria dos computadores, o endereçamento indireto é mais lento do que o direto. Finalmente, com as variáveis locais dinâmicas na pilha, os subprogramas não podem ser sensíveis à história; ou seja, não podem reter valores de dados de variáveis locais entre chamadas. Algumas vezes é desejável escrever subprogramas sensíveis à história. Um exemplo comum de necessidade de ter um subprograma sensível à história é aquele cuja tarefa seja gerar números pseudo-aleatórios. Cada chamada a esse subprograma computa um número pseudo-aleatório, usando o último que ele computou. Ele deve, portanto, armazenar o último em uma variável local estática. As co-rotinas e os subprogramas usados nas construções do laço com iterador (discutidos no Capítulo 8) são outros exemplos da necessidade de ser sensível à história.

A principal vantagem das variáveis locais estáticas é que elas são muito eficientes — normalmente podem ser acessadas mais rapidamente porque não há indireção. Além disso, não exigem nenhuma sobrecarga em tempo de execução para alocação e desalocação. E, é claro, elas permitem que os subprogramas sejam sensíveis à história. A maior desvantagem é a incapacidade de suportar recursão.

No ALGOL 60 e em suas linguagens descendentes, as variáveis locais em um subprograma são, como padrão, dinâmicas na pilha. Em funções C e C++, as variáveis locais são dinâmicas na pilha, a menos que sejam especificamente declaradas como **static**. Por exemplo, na seguinte função C (ou C++), a variável **soma** é estática e **cont** é dinâmica na pilha.

```
int somador(int lista[], int complis) {
    static int soma = 0;
    int cont;
    for (cont = 0; cont < complis; cont++)
        soma += lista [cont];
    return soma;
}
```

Os subprogramas Pascal e Ada têm somente variáveis locais dinâmicas na pilha. Os métodos Java também têm somente variáveis locais dinâmicas na pilha.

Como foi discutido no Capítulo 5, os implementadores do FORTRAN 90 podem escolher se as variáveis locais serão estáticas ou dinâmicas na pilha. A maioria dos implementadores fica com a tradição dos primeiros FORTRANs e torna-as estáticas. De fato, uma vez que os FORTRANs anteriores ao 90 não permitem recursão, não há realmente nenhuma razão obrigatória para torná-las dinâmicas na pilha. A economia de armazenamento normalmente não é considerada algo que valha a perda de eficiência. Os usuários do FORTRAN 90 podem forçar uma ou mais variáveis locais a serem estáticas, independentemente da implementação, listando seus nomes em uma instrução **SAVE**.

No FORTRAN 90, um subprograma pode ser explicitamente especificado como recurso, em cujo caso suas variáveis locais serão dinâmicas na pilha. Essa idéia de especificar que um subprograma particular pode ser recursivamente chamado originou-se com a PL/I. O propósito da especificação explícita é permitir que subprogramas não-recursivos sejam implementados de uma maneira mais eficiente.

## 9.5 Métodos de Passagem de Parâmetros

Os métodos de passagem de parâmetros são as maneiras pelas quais se transmitem parâmetros para subprogramas chamados e/ou de subprogramas chamados. Primeiro, focalizaremos os principais modelos semânticos de métodos de passagem de parâmetros. Depois, discutiremos os vários métodos de implementação, inventados por projetistas de linguagens para esses modelos semânticos. Em seguida, pesquisaremos as opções de projeto das várias linguagens imperativas e discutiremos os métodos reais usados para implementar os modelos. Por fim, refletiremos sobre as considerações de projeto que um projetista de linguagem deve ter em mente ao escolher entre os métodos.

### 9.5.1 Modelos Semânticos de Passagem de Parâmetros

Os parâmetros formais são caracterizados por um de três modelos semânticos distintos: (1) eles podem receber dados do parâmetro real correspondente; (2) podem transmitir dados ao parâmetro real; (3) podem fazer ambos. Os três modelos semânticos são chamados **modo**

**entrada (in mode)**, **modo saída (out mode)** e **modo entrada/saída (inout mode)**, respectivamente.

Existem dois modelos conceituais de como as transferências de dados desenvolvem-se na transmissão de parâmetros: um valor real é transferido fisicamente (para o chamador, para o chamado, ou para ambos) ou um caminho de acesso é transmitido. Mais comumente, o caminho de acesso é um simples ponteiro. A Figura 9.1 ilustra os três modelos semânticos de passagem de parâmetros quando mudanças físicas são usadas.

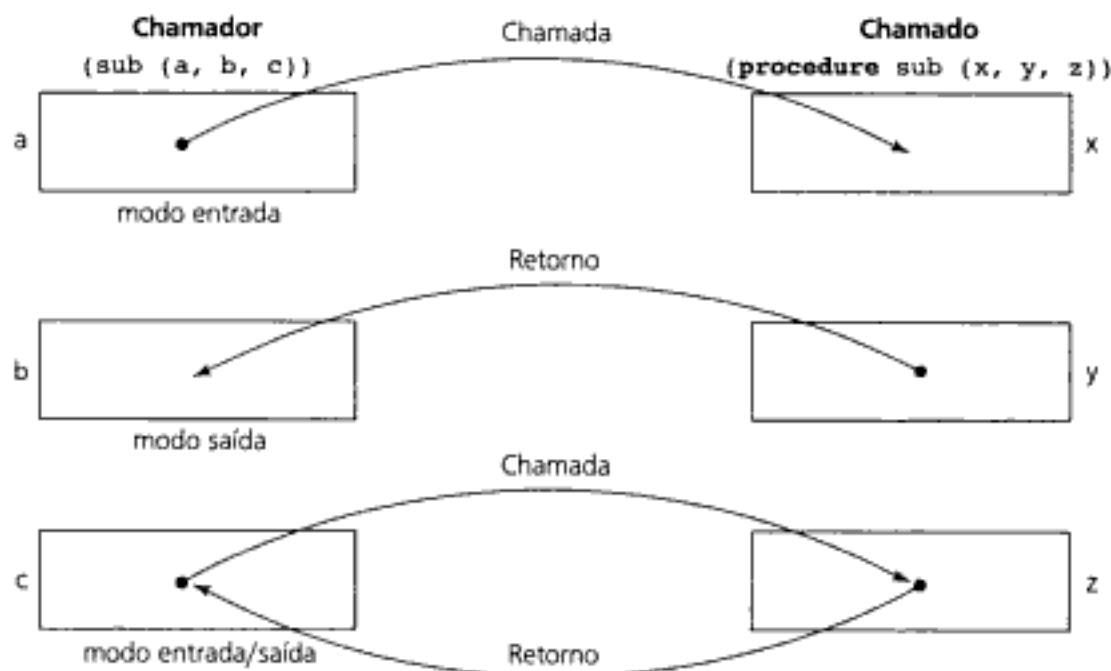
### 9.5.2 Modelos de Implementação de Passagem de Parâmetros

Os projetistas de linguagem desenvolveram uma variedade de modelos para orientar a implementação dos três modelos básicos de transmissão de parâmetros. Nas seções seguintes, discutiremos diversos modelos, suas potencialidades e sua fragilidade.

#### 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, então, age como uma variável local no subprograma, implementando, assim, a semântica de modo entrada. Em programação, isso é uma chamada a uma sub-rotina, passando dados reais dos parâmetros usados nela.

A passagem por valor normalmente é implementada pela transferência de dados real, porque os acessos usualmente são mais eficientes com tal método. Ela poderia ser implementada transmitindo-se um caminho de acesso para o valor do parâmetro real no chamador, mas isso exigiria que o valor estivesse em uma célula protegida contra a escrita (uma que somente pudesse ser lida). Impor a proteção contra a escrita nem sempre é uma ques-



**FIGURA 9.1** Os três modelos semânticos de passagem de parâmetros quando mudanças físicas são usadas.

tão simples. Por exemplo, suponhamos que o subprograma para que o parâmetro foi passado passe-o, por sua vez, a outro subprograma. Eis outra razão para usar-se a transferência física. Conforme veremos na Seção 9.5.3, o C++ oferece um método conveniente e efetivo de impor proteção contra a escrita em parâmetros passados por valor e transmitidos pelo caminho de acesso.

A principal desvantagem do método de passagem por valor, se forem feitas transferências físicas, está no fato de que um armazenamento adicional é necessário para o parâmetro formal no subprograma chamado ou em alguma área fora tanto do subprograma chamador como do chamado. Além disso, o parâmetro real deve ser fisicamente transferido para a área de armazenamento do parâmetro formal correspondente. As operações de armazenamento e de transferência podem ser custosas se o parâmetro for grande, como, por exemplo, um vetor longo.

### 9.5.2.2 Passagem por Resultado

A **passagem por resultado** é um modelo de implementação para parâmetros em modo 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, imediatamente antes que o controle seja transferido de volta para o chamador, seu valor é passado de volta para o parâmetro real do chamador, o qual, evidentemente, deve ser uma variável (como o chamador referenciaria o resultado computado se ele fosse uma literal ou uma expressão?). Se valores forem retornados (em vez do caminho de acesso), como tipicamente ocorre, a passagem por resultado também exigirá o armazenamento e operações de cópia extras exigidos pelo método de passagem por valor. Como acontece com a passagem por valor, a dificuldade de implementar a passagem por resultado ao transmitir um caminho de acesso usualmente resulta na sua implementação pela transferência de dados. Nesse caso, o problema está em assegurar que o valor inicial do parâmetro real não seja usado no subprograma chamado.

Um problema com o modelo de passagem por resultado é que pode haver uma colisão de parâmetros reais, como, por exemplo, um criado com a chamada

```
sub(p1, p1)
```

Em `sub`, supondo-se que os dois parâmetros formais tenham nomes diferentes, ambos, evidentemente, podem ter diferentes valores. Então, qualquer um dos dois que seja atribuído por último ao seu parâmetro real correspondente irá se tornar o valor de `p1`. Dessa forma, a ordem em que os parâmetros reais são atribuídos determina seu valor. Uma vez que a ordem normalmente depende da implementação, podem ocorrer problemas de portabilidade difíceis de diagnosticar.

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 discutiremos na Seção 9.5.2.4.

Outro problema que pode ocorrer com a passagem por resultado é que o implementador pode ser capaz de escolher entre dois tempos diferentes para avaliar os endereços dos parâmetros reais: no momento da chamada ou no momento do retorno. Por exemplo, suponhamos que um subprograma tenha o parâmetro `lista[indice]`. Se `indice` for modificado pelo subprograma por acesso global ou como um parâmetro formal o endereço de `lista[indice]` irá modificar-se entre a chamada e o retorno. O implementador deve escolher o tempo em que será determinado o endereço para retornar o valor, no momento da chamada ou no momento do retorno. Isso torna os programas não-portáveis entre implementações que adotam soluções diferentes em relação a essa questão.

### 9.5.2.3 Passagem por Valor-Resultado

A **passagem por valor-resultado** é um modelo de implementação de parâmetros em modo entrada/saída no qual valores reais são transferidos. Ela é, com efeito, uma combinação da passagem por valor e por resultado. O valor do parâmetro real é usado para inicializar o parâmetro formal correspondente, que age, então, como uma variável local. De fato, parâmetros formais passados por valor-resultado devem ter um armazenamento local associado com o subprograma chamado. Na finalização do subprograma, o valor do parâmetro formal é transmitido de volta para o parâmetro real.

A passagem por valor-resultado, às vezes, é chamada de passagem por cópia porque o parâmetro real é copiado para o parâmetro formal na entrada do subprograma e depois é copiado de volta na finalização do subprograma.

A passagem por valor-resultado partilha com a passagem por valor e com a passagem por resultado as desvantagens de exigir armazenamento múltiplo para parâmetros e tempo de cópia de valores. Ela partilha com a passagem por resultado os problemas associados com a ordem em que os parâmetros reais são atribuídos.

### 9.5.2.4 Passagem por referência

A **passagem por referência** é o segundo modelo de implementação de parâmetros em modo entrada/saída. Entretanto, em vez de transmitir valores de dados para lá e para cá, como na passagem por valor-resultado, o método de passagem por referência transmite um caminho de acesso, usualmente apenas um endereço, para o subprograma chamado. Isso proporciona o caminho de acesso à célula que armazena o parâmetro real. Assim, é permitido que o subprograma chamado acesse o parâmetro real na unidade de programa que faz a chamada. Com efeito, o parâmetro real é partilhado com o subprograma chamado.

A vantagem da passagem por referência é que o próprio processo de passagem é eficiente, tanto em termos de tempo como de espaço. Não é necessário espaço duplicado, nem qualquer atividade de cópia.

Entretanto, há diversas desvantagens no método de passagem por referência. Primeiro, o acesso aos parâmetros formais provavelmente será mais lento porque mais um nível de endereçamento indireto é necessário do que quando valores de dados são transmitidos, como acontece com a passagem por valor-resultado. Segundo, se for exigida somente uma comunicação unidirecional com o subprograma chamado, mudanças inadvertidas e errôneas poderão ser feitas no parâmetro real.

Outro problema sério da passagem por referência é que apelidos podem ser criados. Isso deve ser esperado, porque a passagem por referência torna disponíveis caminhos de acesso aos subprogramas chamados, ampliando o acesso dos mesmos a variáveis não-locais. Há diversas maneiras de criar apelidos quando parâmetros são passados por referência.

Primeiro, podem ocorrer colisões entre parâmetros reais. Considere um procedimento de função C que tem dois parâmetros que devem ser passados por referência, como em

```
void fun(int *primeiro, int *segundo)
```

Se a chamada a `fun` vier a passar a mesma variável duas vezes, como em

```
fun(&total, &total)
```

`primeiro` e `segundo` em `fun` serão apelidos.

Colisões entre elementos de matriz também podem causar apelidos. Por exemplo, suponhamos que a função `fun` seja chamada com dois elementos de vetor especificados com subscritos variáveis, como em

```
fun(&lista[i], &lista[j])
```

Se acontecer de `i` ser igual a `j`, primeiro e segundo novamente serão apelidos.

Colisões entre parâmetros de elementos de matriz e elementos de matrizes passados como parâmetros de nomes de matriz são outra causa possível de apelidos. Se dois dos parâmetros formais de um subprograma forem uma escalar e um vetor com elementos do mesmo tipo, uma chamada como

```
fun1(&lista[i], &lista)
```

poderia resultar em apelidos em `fun1`, porque `fun1` pode acessar todos os elementos de `lista` pelo segundo parâmetro e acessar um único elemento pelo seu primeiro parâmetro.

Uma outra maneira ainda de obter apelidos com parâmetros de passagem por referência acontece por meio de colisões entre parâmetros formais e variáveis não-locais visíveis.

Por exemplo, considere o seguinte código C:

```
int * global;
void principal () {
    extern int * global;
    ...
    sub(global);
    ...
}
void sub (int * local) {
    extern int * global;
    ...
}
```

Dentro de `sub`, `local` e `global` serão apelidos.

O problema com essa espécie de apelidos é o mesmo que nas outras circunstâncias. É prejudicial para a legibilidade e, assim, para a confiabilidade. Faz também com que a verificação do programa fique extremamente difícil.

Todas estas situações de apelidos são eliminadas se for usada a passagem por valor-resultado em vez da passagem por referência. Porém, em vez dos apelidos, outros problemas, às vezes, aparecem, conforme discutimos na Seção 9.5.2.3.

### 9.5.2.5 Passagem Por Nome

A **passagem por nome** é um método de transmissão de parâmetros em modo entrada/saída que não corresponde a um único modelo de implementação, conforme explicamos a seguir. Quando parâmetros são passados por nome, o parâmetro real, com efeito, textualmente substitui o parâmetro formal correspondente em todas as suas ocorrências no subprograma. Isso é bem diferente dos métodos discutidos até aqui. Nesses casos, parâmetros formais são vinculados a valores ou a endereços reais no momento da chamada ao subprograma. Um parâmetro formal de passagem pelo 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 retardada até que o parâmetro formal seja atribuído ou referenciado.

Hidden page

fazer com que um subprograma troque os valores de seus dois parâmetros (veja o Problema 10).

O conceito de vinculação tardia, no qual baseia-se a passagem por nome, de modo algum é estranho ou desacreditado. O poderoso mecanismo de vinculação dinâmica e polimorfismo, uma parte integrante da programação orientada a objeto, é simplesmente uma vinculação tardia de chamadas a subprogramas ou de mensagens a objetos (o polimorfismo será discutido na Seção 9.8). A avaliação preguiçosa é outro mecanismo útil e uma forma de vinculação tardia. Colocando-a brevemente, é o processo de avaliar somente partes do código funcional quando se torna certo que a avaliação desse código é necessária. Nas linguagens imperativas, a avaliação curto-circuito de expressões booleanas é um exemplo de avaliação preguiçosa. Toda avaliação de expressões na linguagem funcional Haskell é preguiçosa, como discutiremos no Capítulo 15.

### 9.5.3 Métodos de Passagem de Parâmetros das Principais Linguagens

O FORTRAN sempre usou o modelo semântico de modo entrada/saída para passagem de parâmetros, mas a linguagem não especifica se a passagem por referência ou a passagem por valor-resultado deve ser usada. Na maioria das implementações FORTRAN anteriores ao FORTRAN 77, os parâmetros eram passados por referência. Nas implementações posteriores, entretanto, a passagem por valor-resultado é freqüentemente usada para parâmetros variáveis simples.

O ALGOL 60 introduziu o método de passagem por nome. Ele também permite a passagem por valor como uma opção. Principalmente devido à dificuldade de implementá-los, os parâmetros passados por nome não foram transportados do ALGOL 60 para quaisquer linguagens subsequentes que se tornaram populares, a não ser a SIMULA-67.

O C usa passagem por valor. Obtém-se a semântica da passagem por referência usando ponteiros como parâmetros. O valor do ponteiro é passado para a função chamada e nada é passado de volta. Contudo, o que foi passado é um caminho de acesso para os dados da função que chamou, portanto a função chamada pode alterar os dados da outra. O C copiou isso do ALGOL 68. Tanto no C como no C++, parâmetros formais podem ser ponteiros para constantes. Os parâmetros reais correspondentes não precisam ser constantes, porque, nesses casos, eles são coagidos para constantes. Isso permite que parâmetros de ponteiro ofereçam a eficiência da passagem por referência com a semântica da passagem por valor. A proteção de escrita a esses parâmetros na função chamada é implícita.

O C++ inclui um tipo de ponteiro especial, chamado tipo de referência, como discutimos no Capítulo 6, freqüentemente usado para parâmetros. Os parâmetros de referência são implicitamente desreferenciados e sua semântica é a passagem por referência. O C++ também permite que parâmetros de referência sejam definidos como constantes. Por exemplo, teríamos

```
void fun(const int &p1, int p2, int &p3) { ... }
```

em que *p1* é passado por referência, mas não pode ser mudado na função *fun*; o parâmetro *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 em modo entrada não são exatamente a mesma coisa. Os parâmetros constantes implementam claramente o modo entrada. Porém, em todas as linguagens imperativas comuns, exceto na Ada, parâmetros de modo de entrada podem ser alterados no subprograma, ainda que essas mudanças jamais se refletam nos

valores dos parâmetros reais correspondentes. Parâmetros constantes jamais podem ser alterados.

Como acontece com o C e com o C++, todos os parâmetros Java são passados por valor. Porém, uma vez que objetos somente podem ser acessados por meio de variáveis de referência, os parâmetros de objeto são efetivamente passados por referência. Similarmente, uma vez que as variáveis de referência não podem apontar para variáveis escalares diretamente, e o Java não possui ponteiros, variáveis escalares não podem ser passadas por referência em Java (ainda que um objeto que contém uma escalar possa).

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.

No Pascal, o método de passagem de parâmetros padrão é a passagem por valor, e a passagem por referência pode ser especificada prefaciando-se os parâmetros formais com a palavra reservada **var**.

Os projetistas da Ada definiram versões dos três modos semânticos de transmissão de parâmetros: **entrada** (*in*), **saída** (*out*) e **entrada/saída** (*inout*). Os três modos são nomeados de maneira apropriada com as palavras reservadas **in**, **out** e **in out**, em que **in** é o método padrão. Por exemplo, considere o seguinte cabeçalho Ada:

```
procedure SOMADOR(A : in out INTEGER;
                    B : in INTEGER;
                    C : out FLOAT)
```

Os parâmetros formais da Ada 83 declarados em modo **out** podem ser alterados mas não referenciados. Parâmetros que estão em modo **in** podem ser referenciados, mas não alterados. Muito naturalmente, parâmetros em modo **in out** podem ser tanto referenciados como alterados. A questão de como os parâmetros Ada **in out** são implementados é interessante e será discutida na Seção 9.5.5.

#### 9.5.4 Verificação de Tipos

Agora é amplamente aceito que a confiabilidade de software exige que os tipos dos parâmetros reais sejam verificados quanto à coerência com os tipos dos parâmetros formais correspondentes. Sem essa verificação de tipos, pequenos erros tipográficos podem levar a erros de programa difíceis de diagnosticar porque não são detectados pelo compilador ou pelo sistema em tempo de execução. Por exemplo, na chamada à função

```
RESULT = SUB1(1)
```

o parâmetro real é uma constante inteira. Se o parâmetro formal de **SUB1** for do tipo real, nenhum erro será detectado sem a verificação de tipo de parâmetro. Não obstante o número inteiro 1 e o real 1 terem o mesmo valor, as suas representações são muito diferentes. **SUB1** não pode produzir um resultado correto, dado um valor inteiro, quando ela espera um valor real.

O FORTRAN 77 não exige verificação de tipo de parâmetro; o Pascal, o FORTRAN 90, o Java e a Ada exigem-na.

O C e o C++ requerem alguma discussão especial quanto à questão da verificação de tipo de parâmetro. No C original, nem o número de parâmetros, nem seus tipos eram verificados. No ANSI C, os parâmetros formais de funções podem ser definidos de duas mane-

ras. Primeiro, eles podem ser como eram no C original; ou seja, os nomes dos parâmetros são listados entre parênteses, e as declarações de tipo relativas a eles vêm depois, como em

```
double sin(x)
double x;
{ ... }
```

Usar esse método evita a verificação de tipos, permitindo que chamadas como

```
double valor;
int cont;
...
valor = sin(cont);
```

sejam legais, mas sem sentido.

A alternativa é chamada método **protótipo**, cujos tipos de parâmetro formais são incluídos na lista, como em

```
double sin(double x)
{ ... }
```

Se esta versão de `sin` for executada com a mesma chamada acima, ou seja

```
valor = sin(cont);
```

ela também será legal. O tipo do parâmetro real (`int`) é verificado em relação ao do parâmetro formal (`double`). Ainda que eles não coincidam, `int` está sujeito a ser coagido para `double`, de modo que a conversão é feita. Se a conversão não for feita (por exemplo, se o parâmetro real fosse um vetor) ou se o número de parâmetros estiver errado, um erro de sintaxe será detectado. Assim, no ANSI C, o usuário escolhe se quer ou não que parâmetros sejam verificados quanto ao tipo.

No C++, todas as funções devem ter seus parâmetros formais sob a forma de protótipo. Porém, a verificação de tipos pode ser evitada para alguns dos parâmetros, substituindo-se a última parte da lista de parâmetros por três pontos (...), como em

```
printf(const char*...);
```

Uma chamada a `printf` deve incluir pelo menos um parâmetro, um ponteiro para uma cadeia de caracteres constante. Fora isso, qualquer coisa (inclusive nada) é legal. A maneira pela qual `printf` determina se há parâmetros adicionais é pela presença de símbolos especiais no parâmetro da cadeia. Por exemplo, o código de formato para saída de números inteiros é `%d`. Esse aparece como parte da cadeia, como em

```
printf("A soma é %d\n", soma);
```

O `%` diz para a função de biblioteca `printf` que há mais um parâmetro.

### 9.5.5 Implementando Métodos de Passagem de Parâmetros

Agora, encaminharemos a questão de como os vários modelos de implementação de passagem de parâmetros são de fato implementados.

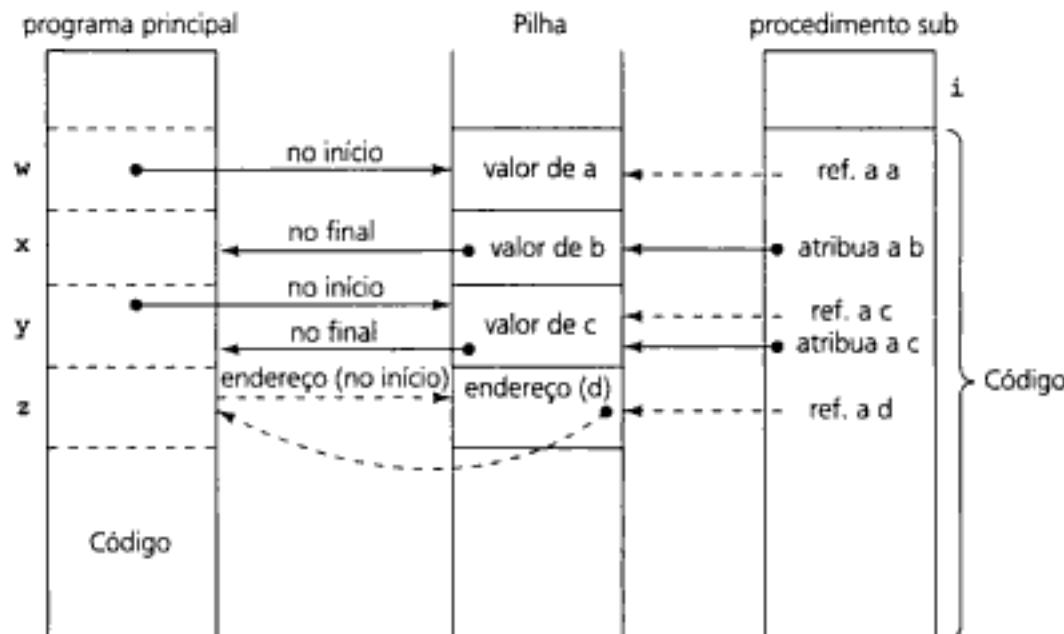
No ALGOL 60 e em suas linguagens descendentes, a comunicação de parâmetros desenvolve-se pela pilha em tempo de execução. A pilha em tempo de execução é inicializa-

da e mantida pelo sistema enquanto ativo, que é um programa de sistema que gerencia a execução de programas. A pilha em tempo de execução é usada extensamente para a ligação do controle de subprogramas, como será discutido no Capítulo 10. Na discussão seguinte, presumimos que a pilha seja usada para todas as transmissões de parâmetros.

Os parâmetros de passagem por valor têm seus valores copiados para localizações da pilha. Estas servem, então, como armazenamento para os parâmetros formais correspondentes. Os de passagem por resultado são implementados como o oposto dos de passagem por valor. Os valores atribuídos aos parâmetros reais passados por resultado são colocados na pilha, de onde podem ser recuperados pela unidade de programa chamadora após o encerramento do subprograma chamado. Os parâmetros passados por valor-resultado podem ser implementados diretamente a partir de sua semântica como uma combinação dos dois tipos de passagem. A localização da pilha referente aos parâmetros é inicializada pela chamada e, depois, usada como uma variável local no subprograma chamado.

Os parâmetros de passagem por referência são, talvez, os mais simples de implementar. Independentemente do tipo do parâmetro real, somente seu endereço deve ser colocado na pilha. No caso das literais, o endereço delas é transmitido. No caso de uma expressão, o compilador precisa criar um código para avaliar a expressão imediatamente antes da transferência de 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 precisará certificar-se de evitar que o subprograma chamado modifique parâmetros literais ou expressões, conforme discutiremos a seguir. O acesso aos parâmetros formais no subprograma chamado é feito pelo endereçamento indireto da localização do endereço na pilha. A implementação da passagem por valor, resultado, valor-resultado e referência, onde se usa a pilha em tempo de execução, é mostrada na Figura 9.2.

Um erro sutil, mas fatal, pode ocorrer com parâmetros passados por referência e passados por valor-resultado se não se tomar cuidado em sua implementação. Suponhamos que um programa contém duas referências à constante 10; a primeira como um parâmetro real em uma chamada a um subprograma. Suponhamos, ainda, que o subprograma modifi-



**FIGURA 9.2** Uma possível implementação da pilha para os métodos comuns de passagem de parâmetros.

que erroneamente o parâmetro formal que corresponde ao 10 para o valor 5. O compilador desse programa pode ter criado uma única localização para o 10 durante a compilação, como os compiladores fazem freqüentemente, e usar essa localização para todas as referências à constante 10 no programa. Mas depois do retorno do subprograma, todas as ocorrências subsequentes de 10 serão, de fato, referências ao valor 5. Permitir que isso aconteça criará um problema de programação muito difícil de diagnosticar. Isso, de fato, aconteceu com muitas implementações do FORTRAN IV.

Os parâmetros passados por nome normalmente são implementados com procedimentos sem parâmetros ou com segmentos de código chamados **thunks**<sup>1</sup>. Um thunk deve ser chamado para toda referência a um parâmetro passado por nome no subprograma chamado. O thunk avalia a referência no ambiente de referenciamento apropriado: o do subprograma que passou o parâmetro real. Eles são vinculados ao seu ambiente de referenciamento no momento da chamada que passou o parâmetro por nome. Um thunk retorna o endereço do parâmetro real. Se a referência a este estiver em uma expressão, o código dela deve incluir o desreferenciamento necessário para obter o valor da célula cujo endereço foi retornado pelo thunk. Em sua totalidade, esse é um processo custoso em relação ao endereçamento indireto simples usado pelos parâmetros passados por referência. Lembre-se: para parâmetros reais, variáveis escalares, passagem por nome e passagem por referência são semanticamente equivalentes. O custo de implementação da passagem por referência é o do endereçamento indireto, enquanto a passagem por nome exige uma chamada a subprograma — embora sem parâmetros — e sua execução para realizar a mesma coisa.

A definição da linguagem Ada 83 especifica que parâmetros escalares (não-estruturados) devem ser passados por cópia; ou seja, parâmetros em modo **in** e **in out** devem ser variáveis locais inicializadas quando se copia o valor do parâmetro real correspondente. Parâmetros simples no modo **out** ou **in out** devem ter seus valores copiados de volta para o parâmetro real correspondente quando da finalização do subprograma. A ordem dessas cópias, quando houver mais de uma, não é definida pela linguagem. A avaliação de parâmetros em modo **out** e de **in out** é feita antes que ocorra a transferência de controle para o subprograma chamado. Por exemplo, suponhamos que um parâmetro real em modo **out** tenha a forma

#### LISTA(INDICE)

O valor de seu endereço é computado no momento da chamada. Se acontecesse de **INDICE** estar visível no subprograma chamado e este o modificasse, o endereço do parâmetro não seria afetado.

No caso de parâmetros formais que são matrizes ou registros, os implementadores da Ada 83 podem optar entre a passagem por valor-resultado e a passagem por referência. Ao deixar de especificar o método de implementação para passar parâmetros estruturados, os projetistas da Ada 83 deixaram aberta a possibilidade de um problema sutil: os dois métodos de implementação podem levar a diferentes resultados de programa para certos programas. Tal diferença pode ocorrer porque o método de passagem por referência oferece

<sup>1</sup>N. de T. Thunk: som oco (a bola de basquetebol fez um "thunk" ao bater no aro da cesta). **Nota histórica:** Há dois mitos onomatopáicos que circulam sobre a origem desse termo. O mais comum é que se trata do som feito pelos dados quando atingem a pilha; outro, sustenta que o som é dos dados atingindo um acumulador. Outro ainda, sugere que é o som da expressão sendo descongelada no momento da avaliação do argumento. De fato, de acordo com os inventores, o termo foi cunhado depois que eles perceberam (nas horas e horas de discussão que avançavam pela madrugada) que o tipo de um argumento do ALGOL 60 poderia ser calculado antecipadamente com um pouco de reflexão na hora de compilar, simplificando a maquinaria de avaliação. Em outras palavras, ele "já havia sido pensado"; dessa forma, foi batizado de "thunk", que é o "pretérito de ' pensar às duas da madrugada. '" (think at two in the morning).

Hidden page

contornar o problema devido à inclusão que eles fazem da aritmética de ponteiros. A matriz pode ser passada como um ponteiro, e as dimensões reais da matriz podem ser incluídas como parâmetros. Então, a função pode avaliar a função de associação de armazenamento usando a aritmética de ponteiros cada vez que um elemento da matriz precisar ser referenciado. Por exemplo, considere o seguinte protótipo de função:

```
void fun(float *mat_ptr, int num_linhas, int num_cols);
```

A seguinte instrução pode ser usada para mudar o valor da variável *x* para o elemento *[linha][col]* da matriz de parâmetro em *fun*:

```
* (mat_ptr + (linha * num_cols) + col) = x;
```

Embora isso funcione, obviamente é difícil de ler e, devido à sua complexidade, é propenso a erro. A dificuldade com a leitura pode ser aliviada usando-se uma macro para definir a função de associação de armazenamento, como, por exemplo

```
#define mat_ptr(r, c)
    (* (mat_ptr + ((r) * num_cols) + (c)))
```

Assim, a atribuição acima pode ser escrita como

```
mat_ptr(linha,col) = x;
```

Outras linguagens lidam de maneira diferente com o problema de passar matrizes multidimensionais. Os compiladores Ada são capazes de determinar o tamanho definido das dimensões de todas as matrizes usadas como parâmetros no momento em que os programas são compilados. Em Ada, os tipos de matriz não-restrigidas podem ser parâmetros formais. Um tipo de matriz não-restrigida é aquele em que as faixas de índices não são fornecidas na definição do tipo de matriz. As definições de variáveis de tipos de matriz não-restrigidas devem incluir faixas de índices. O código em um subprograma no qual é passada uma matriz não-restrigida pode obter informações sobre a faixa de índices do parâmetro real associado com esses parâmetros. Por exemplo, considere as seguintes definições:

```
type TIPO_MATRIZ is array (INTEGER range <>,
                           INTEGER range <>) of FLOAT;
MATTRIZ_1 : TIPO_MATRIZ( 1.. 100, 1.. 20);
```

Segue-se uma função que retorna a soma dos elementos de matrizes do TIPO\_MATRIZ:

```
function SOMADOR(MAT : in TIPO_MATRIZ) return FLOAT is
    SOMA : FLOAT := 0.0;
begin
    for LINHA in MAT'range(1) loop
        for COL in MAT'range(2) loop
            SOMA := SOMA + MAT(LINHA, COL);
        end loop; - for COL...
    end loop; - for LINHA...
    return SOMA;
end SOMADOR;
```

O atributo **range** retorna a faixa de subscritos do subscrito nomeado da matriz passada, de modo que isso funciona independentemente do tamanho ou das faixas de índices do parâmetro.

Nas versões anteriores à versão 90 do FORTRAN, o problema é encaminhado da seguinte maneira. Parâmetros formais do tipo matriz devem ter uma declaração depois do

cabeçalho. Em relação às matrizes unidimensionais, os subscritos são irrelevantes nessas declarações. Mas em relação às matrizes multidimensionais, os subscritos permitem, nessas declarações, que o compilador crie a função de associação de armazenamento. Considere o seguinte exemplo de sub-rotina FORTRAN esquemática:

```
SUBROUTINE SUB(MATRIZ, LINHAS, COLS, RESULT)
    INTEGER LINHAS, COLS
    REAL MATRIZ(LINHAS, COLS), RESULT
    ...
END
```

Isso funciona perfeitamente, contanto que o parâmetro real COLS tenha o valor usado para o número de colunas na definição da matriz passada. Se a matriz a ser passada não estiver preenchida atualmente com dados úteis para o tamanho definido, tanto os tamanhos de índice definidos como os de preenchidos poderão ser passados ao subprograma. Então, os tamanhos definidos são usados na declaração local da matriz, e os de índice preenchidos são usados para controlar a computação na qual os elementos da matriz são referenciados. Por exemplo, considere o seguinte subprograma FORTRAN 90:

```
SUBROUTINE SOMAMAT(MATRIZ, LINHAS, COLS, LINHAS_PREENCHIDAS,
                     COLS_PREENCHIDAS, SOMA)
    INTEGER LINHAS, COLS, LINHAS_PREENCHIDAS, COLS_PREENCHIDAS,
           INDICE_LIN, INDICE_COL
    REAL MATRIZ(LINHAS, COLS), SOMA
    SOMA = 0.0
    DO 20 INDICE_LIN = 1, LINHAS_PREENCHIDAS
        DO 10 INDICE_COL = 1, COLS_PREENCHIDAS
            SOMA = SOMA + MATRIZ(INDICE_LIN, INDICE_COL)
10    CONTINUE
20    CONTINUE
    RETURN
END
```

O Java usa uma técnica para passar matrizes multidimensionais como parâmetros similar à da Ada. Em Java, matrizes são objetos. Eles são todos unidimensionais (vetores) mas seus elementos podem por sua vez ser vetores. Cada um deles herda uma constante nomeada (`length`) fixada no tamanho do vetor quando o objeto vetor é criado. O parâmetro formal para uma matriz aparece com dois conjuntos de colchetes vazios, como no método abaixo, que faz aquilo que a função do exemplo Ada, `somador`, faz.

```
float somador(float mat[][]) {
    float soma = 0.0f;
    for (int linha = 0; linha < mat.length; linha++) {
        for (int col = 0; col < mat[linha].length; col++) {
            soma += mat[linha][col];
        } //** for (int linha -
    } //** for (int col -
    return soma;
}
```

Uma vez que cada vetor tem seu próprio valor de tamanho, em uma matriz as linhas podem ter diferentes tamanhos.

### 9.5.7 Considerações de Projeto

Duas importantes considerações estão envolvidas na escolha de métodos de passagem de parâmetros: eficiência, e se é necessária uma transferência unidirecional ou bidirecional de dados.

Os princípios de engenharia de software contemporâneos determinam que o acesso por meio de código de subprograma a dados externos a este seja minimizado. Com o objetivo em mente, parâmetros em modo entrada devem ser usados quando nenhum dado deve ser retornado por meio de parâmetros ao chamador. Parâmetros em modo saída devem ser usados quando nenhum dado for transferido ao subprograma chamado, mas o subprograma deve transmitir dados de volta ao chamador. Por fim, parâmetros em modo entrada/saída devem ser usados somente quando dados precisarem mover-se em ambas as direções entre o subprograma chamador e o chamado.

Há uma consideração prática em conflito com esse princípio. Às vezes, é justificável passar caminhos de acesso para transmissão de parâmetros unidirecional. Por exemplo, quando uma matriz grande deve ser passada para um subprograma que não a modifica, um método unidirecional talvez seja preferível. Porém, a passagem por valor exigiria que a matriz inteira fosse transferida para uma área de armazenamento local do subprograma. Isso seria custoso tanto em termos de tempo como de espaço. Por causa disso, matrizes grandes, às vezes, são passadas por referência. Eis o exato motivo pelo qual a definição da Ada 83 permite que os implementadores escolham entre os dois métodos para parâmetros estruturados. Os parâmetros de referência a constantes do C++ oferecem outra solução. Outra abordagem alternativa seria permitir que o usuário escolha entre os métodos.

A escolha de um método de passagem de parâmetros para funções está relacionada a outra questão de projeto: os efeitos colaterais funcionais. Essa questão será discutida na Seção 9.10.

### 9.5.8 Exemplos de Passagem de Parâmetros

Considere a seguinte função C:

```
void troca (int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

Suponhamos que a função seja chamada com

```
troca(c, d);
```

Lembre-se que o C usa passagem por valor. As ações de `troca` podem ser descritas pelo seguinte pseudocódigo:

```
a = c      - Mova o primeiro valor de parâmetro para dentro
b = d      - Mova o segundo valor de parâmetro para dentro
temp = a
a = b
b = temp
```

Ainda que `a` acabe ficando com o valor de `d`, e `b` acabe ficando com o valor de `c`, os valores de `c` e `d` não são modificados porque nada é transmitido de volta para o chamador.

Hidden page

```
troca2(i, lista[i]);
```

Nesse caso, as ações são descritas por

```
a = &i      - Mova o primeiro endereço de parâmetro para dentro
b = &lista[i] - Mova o segundo endereço de parâmetro para dentro
temp = *a
*a = *b
*b = temp
```

Não obstante o valor de `*a` (que é `i`) ser mudado antes de `*b` (que é `lista[i]`), isso não afeta a exatidão da troca, porque o endereço de `lista[i]` é computado no momento da chamada e não se modifica depois disso, independentemente do que acontecer a `i`.

A semântica da passagem por valor-resultado é idêntica à da passagem por referência, exceto quando apelidos estão envolvidos. Lembre-se de que a Ada usa passagem por valor-resultado para parâmetros escalares em modo entrada/saída. Para explorar a passagem por valor-resultado, considere a seguinte função, `troca3`, a qual supomos usar parâmetros passados por valor-resultado. Ela é escrita em uma sintaxe similar à da Ada.

```
procedure troca3(a : in out integer, b : in out integer) is
    temp : integer;
begin
    temp := a;
    a := b;
    b := temp;
end troca3;
```

Suponhamos que `troca3` seja chamada com

```
troca3(c, d);
```

As ações de `troca3` com essa chamada são

```
end_c = &c      - Mova o primeiro endereço de parâmetro para dentro
end_d = &d      - Mova o segundo endereço de parâmetro para dentro
a = *end_c      - Mova o primeiro valor de parâmetro para dentro
b = *end_d      - Mova o segundo valor de parâmetro para dentro
temp = a
a = b
b = temp
*end_c = a      - Mova o primeiro valor de parâmetro para fora
*end_d = b      - Mova o segundo valor de parâmetro para fora
```

Assim, mais uma vez, esse subprograma de troca opera corretamente. Em seguida, considere a chamada

```
troca3(i, lista[i]);
```

Nesse caso, as ações são

```
end_i = &i      - Mova o primeiro endereço de parâmetro para dentro
end_listai = &lista[i] - Mova o segundo endereço de parâmetro para dentro
a = *end_i      - Mova o primeiro valor de parâmetro para dentro
```

```

b = *end_listai      - Mova o segundo valor de parâmetro para
                      dentro
temp = a
a = b
b = temp
*end_i = a           - Mova o primeiro valor de parâmetro para
                      fora
*end_listai = b      - Mova o segundo valor de parâmetro para
                      fora

```

Novamente, o subprograma opera corretamente; nesse caso, porque os endereços para os quais retorna os valores dos parâmetros são computados no momento da chamada, em vez de no momento de retorno. Se os endereços dos parâmetros reais fossem computados no momento de retorno, os resultados seriam errados.

Finalmente, devemos explorar o que acontece quando é invocado o apelido com passagem por valor-resultado e com passagem por referência. Considere o seguinte programa esquemático escrito em uma sintaxe semelhante à C:

```

int i = 3; /* i é uma variável global */
void fun(int a, int b) {
    i = b;
}
void principal() {
    int lista[10];
    lista[i] = 5;
    fun(i, lista[i]);
}

```

Em fun, se for usada passagem por referência, i e a serão apelidos. Se for usada passagem por valor-resultado, i e a não serão apelidos. As ações de fun, supondo passagem por valor-resultado, são

```

end_i = &i           - Mova o primeiro endereço de parâmetro
                      para dentro
end_listai = &lista[i] - Mova o segundo endereço de parâmetro
                      para dentro
a = *end_i           - Mova o primeiro valor de parâmetro
                      para dentro
b = *end_listai      - Mova o segundo valor de parâmetro para
                      dentro
i = b                - Ajusta i em 5
*end_i = a           - Mova o primeiro valor de parâmetro para
                      fora
*end_listai = b      - Mova o segundo valor de parâmetro
                      para fora

```

Nesse caso, a atribuição para a global i em fun muda seu valor de 3 para 5, mas a cópia de retorno do primeiro parâmetro formal (a segunda linha de baixo para cima) fixa seu valor novamente em 3. A observação importante aqui é que se for usada passagem por referência, o resultado será que a cópia de retorno não fará parte da semântica, e i permanecerá sendo 5. Note também que, uma vez que o endereço do segundo parâmetro é computado no início de fun, qualquer mudança na global i não terá nenhum efeito sobre o endereço usado no final para retornar o valor de lista[i].

## 9.6 Parâmetros Que São Nomes de Subprograma

Ocorre um grande número de situações em programação mais convenientemente manipuladas se nomes de subprograma puderem ser enviados como parâmetros a outros subprogramas. Uma das mais comuns ocorre quando um subprograma precisa avaliar alguma função matemática. Por exemplo, um subprograma que faz integração numérica estima a área sob uma curva avaliando a função correspondente em uma série de diferentes pontos. Quando este programa é escrito, deve ser utilizável para qualquer função; não precisa ser reescrito para toda função que deve ser integrada. Portanto, é natural que o nome de uma função de programa que avalia a função matemática a ser integrada seja enviado ao subprograma integrador como um parâmetro.

Não obstante a idéia ser natural e aparentemente simples, os detalhes de como ela funciona podem ser confusos. Se somente a transmissão do código do subprograma fosse necessária, isso poderia ser feito passando-se um ponteiro único. Porém, surgem diversas complicações.

Primeiro, há a questão da verificação de tipo dos parâmetros das ativações do subprograma passado como parâmetro. A definição original do Pascal (Jensen e Wirth, 1974) permitia que subprogramas fossem passados como parâmetros sem incluir suas informações de tipo de parâmetro. Se for possível uma compilação independente (o que não era possível no Pascal original), não será permitido que o compilador nem mesmo verifique o número correto de parâmetros. Na ausência de compilação independente, a verificação da coerência de parâmetros é possível, embora seja uma tarefa muito complexa, e usualmente não é feita. O FORTRAN 77 padece do mesmo problema, mas, uma vez que jamais se verifica a coerência de tipos nele, isso não é um problema adicional.

Quando um nome de subprograma é passado como um parâmetro no ALGOL 68 ou nas versões posteriores do Pascal, os tipos dos parâmetros formais são incluídos na lista de parâmetros formais do subprograma recebedor. Assim, a coerência de tipos de parâmetro na chamada real ao subprograma passado pode ser estaticamente verificada. Por exemplo, considere o seguinte código Pascal:

```

procedure integral(function fun(x : real) : real;
                    liminf, limsup : real;
                    var result : real);

...
var funval : real;
begin
...
funval := fun(liminf);
...
end;

```

O parâmetro real na chamada a `fun` em `integral` pode ser estaticamente verificado quanto à coerência com o tipo do parâmetro formal de `fun`, que aparece na lista dos formais de `integral`.

No C e no C++, funções não podem ser passadas como parâmetros, mas ponteiros para elas podem. O tipo de um ponteiro para uma função é o protocolo da função. Uma vez que o protocolo inclui todos os tipos de parâmetro, estes podem ser completamente verificados quanto ao tipo.

O FORTRAN 90 tem um mecanismo para oferecer tipos de parâmetros para subprogramas passados como parâmetro, e eles devem ser verificados. A Ada não permite que

Hidden page

A vinculação rasa não é apropriada para linguagens estruturadas em bloco devido à vinculação estática de variáveis. Por exemplo, suponhamos que o procedimento **SENDER** passe o procedimento **SENT** como um parâmetro ao procedimento **RECEIVER**. O problema é que **RECEIVER** pode não estar no ambiente estático de **SENT**, tornando, assim, muito pouco natural que esta tenha acesso às variáveis daquela. Por outro lado, é perfeitamente normal em uma linguagem de escopo estático que qualquer subprograma, inclusive enviado como um parâmetro, tenha seu ambiente de referenciamento determinado pela posição léxica de sua definição. Portanto, é mais lógico que as linguagens estruturadas em bloco usem a vinculação profunda. Algumas linguagens de escopo dinâmico, como a SNOBOL, usam vinculação rasa.

## 9.7 Subprogramas Sobrecarregados

Um operador sobrecarregado é o que tem múltiplos significados. O significado de uma instância particular de um operador sobrecarregado é determinado pelos tipos de seus operandos. Por exemplo, se o operador **\*** tiver dois operandos reais em um programa C, isso especificará uma multiplicação de números reais. No entanto, se o mesmo operador tiver dois operandos inteiros, isso especificará uma multiplicação de números inteiros.

Um **subprograma sobrecarregado** tem o mesmo nome que outro no mesmo ambiente de referenciamento. Cada versão dele deve ter um único protocolo; ou seja, ele deve ser diferente quanto ao número, à ordem ou aos tipos de seus parâmetros, ou em seu tipo de retorno, se for uma função. O significado de uma chamada a um subprograma sobrecarregado é determinado pela lista de parâmetros reais (e/ou possivelmente pelo tipo do valor retornado, no caso de uma função).

O C++, o Java e a Ada incluem subprogramas sobrecarregados predefinidos. Por exemplo, a Ada tem diversas versões da função de saída **PUT**. As versões mais comuns são as que aceitam valores de cadeias, inteiros e reais como parâmetros. Uma vez que cada versão de **PUT** tem um tipo de parâmetro único, o compilador pode distinguir ocorrências de chamadas a **PUT** por meio dos parâmetros de tipos diferentes.

Na Ada, o tipo de retorno de uma função sobrecarregada é usado para tirar a ambigüidade de chamadas. Portanto, duas funções sobrecarregadas podem ter o mesmo perfil de parâmetro e diferir somente em seus tipos de retorno. Isso funciona porque a Ada não permite expressões em modo misto, de modo que o contexto de uma chamada a função pode especificar o tipo retornado da função. O C++ e o Java permitem expressões em modo misto, e o tipo de retorno é irrelevante para tirar a ambigüidade de funções (ou de métodos) sobrecarregadas.

Os usuários também têm permissão para escrever múltiplas versões de subprogramas com o mesmo nome em Ada, em Java e em C++. Ainda que não seja necessário a esses subprogramas fornecerem basicamente o mesmo processo, eles usualmente o fazem. Por exemplo, um programa particular pode exigir dois procedimentos de classificação: um para vetores de números inteiros e um para vetores de números reais. Ambos podem ser nomeados como **ORDENA**, contanto que os tipos de seus parâmetros sejam diferentes. No seguinte programa Ada esquemático, dois procedimentos chamados **ORDENA** são incluídos:

```
procedure PRINCIPAL is
  type VETOR_REAL is array ( INTEGER range <>) of FLOAT;
  type VETOR_INT is array ( INTEGER range <>) of INTEGER;
```

```

    ...
procedure ORDENA(LISTA_REAL : in out VETOR_REAL;
                  LIM_INF : in INTEGER;
                  LIM_SUP : in INTEGER) is
    ...
end ORDENA;
procedure ORDENA(LISTA_INT : in out VETOR_INT;
                  LIM_INF : in INTEGER;
                  LIM_SUP : in INTEGER) is
    ...
end ORDENA;
...
end PRINCIPAL;

```

Os subprogramas sobrecarregados que têm parâmetros-padrão podem levar a chamadas a subprograma ambíguas. Por exemplo, considere o seguinte código C++:

```

void fun(float b = 0.0);
void fun();
...
fun();

```

A chamada é ambígua e causará um erro de compilação.

## 9.8 Subprogramas Genéricos

A reutilização de software pode ser um importante contribuinte para os aumentos de sua produtividade. Uma maneira de aumentar a sua capacidade de reutilização é diminuir a necessidade de criar diferentes subprogramas que implementem o mesmo algoritmo em diferentes tipos de dados. Por exemplo, o programador não precisa escrever quatro diferentes subprogramas de classificação para quatro vetores diferentes somente em termos de tipo de elemento.

Um **subprograma genérico** ou **polimórfico** leva parâmetros de diferentes tipos em várias ativações. Os subprogramas sobrecarregados apresentam uma espécie particular de polimorfismo chamado **polimorfismo ad hoc**. Um tipo mais geral deste último é apresentado pelas funções da APL. Por causa da vinculação dinâmica de tipos da APL, os tipos de parâmetros podem permanecer sem serem especificados; eles simplesmente são vinculados ao tipo dos parâmetros reais correspondentes.

O **polimorfismo paramétrico** é apresentado por um subprograma que leva um parâmetro genérico usado em uma expressão de tipos que descreve os tipos dos parâmetros do subprograma. Tanto a Ada como o C++ apresentam um tipo de polimorfismo paramétrico em tempo de compilação.

### 9.8.1 Subprogramas Genéricos em Ada

A Ada apresenta um polimorfismo paramétrico por meio de uma construção que suporta a criação de múltiplas versões de unidades de programa para aceitar parâmetros de diferentes tipos de dados. As versões diferentes do subprograma são instanciadas ou construídas

pelo compilador mediante pedido do programa usuário. Uma vez que as versões do subprograma têm todos o mesmo nome, isso proporciona a ilusão de que um único subprograma pode processar dados de diferentes tipos em diferentes chamadas. Uma vez que as unidades de programa desse tipo são genéricas por natureza, às vezes elas são chamadas **unidades genéricas**.

O mesmo mecanismo pode ser usado para permitir diferentes execuções de um subprograma chamar diferentes instâncias de um subprograma genérico. Isso é útil para proporcionar a funcionalidade de subprogramas passados como parâmetros.

O exemplo seguinte ilustra um procedimento de três parâmetros genéricos, permitindo que o subprograma assuma como parâmetro um vetor genérico. É um procedimento de classificação por trocas projetado para funcionar em qualquer vetor com elementos numéricos, usando qualquer faixa de subscrito do tipo ordinal:

```
generic
    type TIPO_INDICE is (<>);
    type TIPO_ELEMENTO is private;
    type VECTOR is array (TIPO_INTEIRO range <>) of
        TIPO_ELEMENTO;
    procedure ORD_GENERICA(LISTA : in out VECTOR);
procedure ORD_GENERICA(LISTA : in out VECTOR) is
    TEMP : TIPO_ELEMENTO;
begin
    for TOPO in LISTA'FIRST.. TIPO_INDICE'PRED(LISTA'LAST) loop
        for BASE in TIPO_INDICE'SUCC(TOPO).. LISTA'LAST loop
            if LISTA(TOPO) > LISTA(BASE) then
                TEMP := LISTA(TOPO);
                LISTA(TOPO) := LISTA(BASE);
                LISTA(BASE) := TEMP;
            end if;
        end loop; - for BASE...
    end loop; - for TOPO...
end ORD_GENERICA;
```

Partes desse procedimento genérico podem parecer bastante estranhas se você não estiver familiarizado com a Ada. Porém, não é importante entender todos os detalhes da sintaxe. O tipo de vetor e o tipo de seus elementos são os dois parâmetros genéricos do procedimento. O vetor é declarado para ter qualquer subscrito de tipo (ou seja, qualquer tipo que seja válido como subscrito) com qualquer faixa.

Tal classificação genérica nada mais é do que um modelo para um procedimento; nenhum código é gerado para ele pelo compilador, e ele não tem nenhum efeito sobre um programa, a menos que seja instanciado para algum tipo. A instânciação é realizada com uma instrução como a seguinte:

```
procedure ORD_INTEIRO is new ORD_GENERICA(
    TIPO_INDICE => INTEGER;
    TIPO_ELEMENTO => INTEGER;
    VECTOR => INT_ARRAY);
```

O compilador reage a essa instrução construindo uma versão de `ORD_GENERICA` chamada `ORD_INTEIRO` que classifica vetores do tipo `INT_ARRAY` com elementos do tipo `INTEGER` e subscritos do tipo `INTEGER`.

`ORD_GENERICO`, como está escrito, presume que o operador `>` é definido para os elementos do vetor a ser ordenado. A generalidade de `ORD_GENERICO` pode ser aumentada incluindo-se uma função de comparação entre os parâmetros genéricos.

Lembre-se de que a Ada não permite que subprogramas sejam passados a outros subprogramas. Para apresentar essa funcionalidade, a Ada usa subprogramas formais genéricos. Em uma linguagem como o Pascal, subprogramas são passados como parâmetros, a fim de que uma chamada particular a um subprograma possa executar usando o subprograma específico passado para computar seu resultado. Na Ada, o mesmo resultado é conseguido permitindo-se que o usuário instancie um subprograma genérico qualquer número de vezes, cada uma com um subprograma diferente que possa ser usado. Por exemplo, o procedimento `integral`, conforme definido na Seção 9.6, pode ser escrito em Ada como

```
generic
    with function FUN(x : FLOAT) return FLOAT;
    procedure INTEGRAL (LIMINFERIOR : in FLOAT;
                        LIMSUPERIOR : in FLOAT;
                        RESULT      : out FLOAT);
    procedure INTEGRAL (LIMINFERIOR : in FLOAT;
                        LIMSUPERIOR : in FLOAT;
                        RESULT      : out FLOAT) is
        FUNVAL : FLOAT;
    begin
        ...
        FUNVAL := FUN(LIMINFERIOR);
        ...
    end;
```

Isso poderia ser instanciado para uma função `FUN1` definida pelo usuário com

```
procedure INTEGRAL_FUN1 is new INTEGRAL(FUN => FUN1);
```

Agora, `INTEGRAL_FUN1` é um procedimento para integrar a função `FUN1`.

### 9.8.2 Funções Genéricas em C++

As funções genéricas em C++ têm o nome descritivo de funções modelo. A definição de uma função modelo tem a forma geral

```
template <parâmetros>
- uma definição de função que pode incluir parâmetros
```

Cada parâmetro (deve haver pelo menos um) tem uma das formas

```
class identificador
typename identificador
```

Os parâmetros podem também ser outra declaração de modelo mas não consideraremos esta opção aqui.

Como um exemplo, considere a função modelo

```
template <class Tipo>
tipo(Primeiro tipo, Segundo tipo) {
    return primeiro > segundo ? primeiro : segundo;
}
```

em que `Tipo` é o parâmetro que especifica o tipo de dados sobre o qual a função será operada. Essa função modelo pode ser instanciada para qualquer tipo cujo operador `>` seja definido. Por exemplo, se ela fosse instanciada com `int` como parâmetro, ela seria

```
int max(int primeiro, int segundo) {
    return primeiro > segundo ? primeiro : segundo;
}
```

Ainda que esse processo pudesse ser definido como uma macro, uma macro teria a desvantagem de não operar corretamente se os parâmetros fossem expressões com efeitos colaterais. Por exemplo, suponhamos que a macro fosse definida como

```
#define max(a, b) ((a) > (b)) ? (a) : (b)
```

Isso é genérico em termos de que funciona para qualquer tipo numérico. Porém, nem sempre funciona corretamente se for chamado com um parâmetro que tem um efeito colateral, como, por exemplo,

```
max(x++, y)
```

Isso produz

```
((x++) > (y) ? (x++) : (y))
```

Na verificação se o valor de `x` é maior do que o de `y`, `x` será incrementado duas vezes.

As funções modelo do C++ são instanciadas implicitamente ou quando a função é nomeada em uma chamada ou quando seu endereço é tomado com o operador `&`. Por exemplo, a função modelo definida acima seria instanciada por meio do seguinte segmento de código, uma para parâmetros do tipo `int` e uma para parâmetros do tipo `char`:

```
int a, b, c;
char d, e, f;
...
c = max(a, b);
f = max(d, e);
```

O exemplo seguinte é uma versão C++ do subprograma de classificação genérico apresentado na Seção 9.8.1. Ele é bem diferente porque os subscritos de vetor do C++ restringem-se a serem números inteiros com o limite inferior fixado em zero.

```
template <class Tipo>
void ord_generico(Tipo lista[], int comp) {
    int topo, base;
    Tipo temp;
    for (topo = 0; topo < comp - 2; topo++)
        for (base = topo + 1; base < comp - 1; base++)
            if (lista[topo] > lista[base]) {
                temp = lista[topo];
                lista[topo] = lista[base];
                lista[base] = temp;
            } //** fim do for (base = ...
} //** fim do ord_generico
```

Um exemplo de instanciação dessa função modelo é:

```
float lista_real[100];
...
```

```
ord_generico(lista_real, 100);
```

Os subprogramas genéricos da Ada e as funções modeladas do C++ são uma espécie de primo-pobre dos subprogramas cujos tipos dos parâmetros formais são vinculados dinamicamente aos tipos dos parâmetros reais em uma chamada. Nesse caso, somente uma única cópia do código é necessária, ao passo que com as abordagens Ada e C++ uma cópia precisa ser criada durante a compilação para cada tipo diferente exigido, e a vinculação de chamadas a subprograma é estática.

A Smalltalk, o Java, a Ada 95 e o C++ suportam métodos cujas chamadas sejam vinculadas dinamicamente à versão correta do método, de acordo com os tipos dos parâmetros reais. Isso será discutido no Capítulo 12.

## 9.9 Compilação Separada e Independente

A capacidade de compilar parte de um programa sem a necessidade de compilá-lo por inteiro é fundamental para a construção de sistemas de software grandes. Assim, linguagens projetadas para essas aplicações devem permitir esse tipo de compilação. Com essa capacidade, somente os módulos de um sistema em modificação precisam ser recompilados durante o desenvolvimento ou a manutenção. As unidades compiladas recentemente e as anteriormente são coletadas por um programa chamado *linkeditor*, que faz parte do sistema operacional. Sem essa capacidade, toda mudança em um sistema exigiria uma recompilação completa. No caso de um sistema grande, isso é custoso.

Nesta seção, discutiremos duas abordagens distintas para compilar partes de programas, chamadas compilação separada e compilação independente. As partes de programas que podem ser compiladas, às vezes, são chamadas unidades de compilação.

O termo **compilação separada** significa que as unidades de compilação podem ser compiladas em tempos diferentes, mas elas não são independentes uma da outra se qualquer uma delas acessar ou usar quaisquer entidades da outra. Tal interdependência é necessária se precisar ser feita verificação de interface. Primeiro, discutiremos a compilação separada no contexto da Ada.

Para fornecer uma compilação separada confiável de uma unidade, o compilador precisa ter acesso às informações sobre entidades de programa (variáveis, tipos e protocolos de subprogramas) que a unidade usa, mas que são declaradas em outro lugar. As informações sobre um pacote Ada visíveis em outras unidades, chamadas de entidades exportadas, formam a interface do pacote. Lembre-se de que o protocolo de um procedimento inclui o número, os nomes e os tipos de seus parâmetros, juntamente com a ordem em que eles aparecem. No caso de uma função, o tipo do valor retornado também é incluído. As implementações Ada mantêm esses tipos de informação sobre interfaces de unidades em uma biblioteca acessível ao compilador. Toda compilação faz com que a informação sobre a interface da mesma seja colocada em uma biblioteca.

As bibliotecas armazenam unidades de biblioteca, que são unidades compiladas. As unidades de compilação da Ada são entidades como cabeçalhos de subprograma, como declarações de pacote (veja o Capítulo 11) e como corpos de subprograma.

Durante a compilação de uma unidade de programa Ada, todas as entidades declaradas externamente usadas são verificadas quanto ao tipo em relação aos seus usos locais. No

caso de subprogramas, o protocolo inteiro é verificado quanto ao tipo. Nem todas as informações de biblioteca estão disponíveis para a compilação de uma unidade de programa particular. Os nomes das unidades que oferecem entidades externas necessárias são listados em uma instrução **with** no início da unidade em compilação. Usando **with**, o programador especifica as unidades externas que o código deve acessar. Por exemplo, o procedimento seguinte usa entidades de duas unidades externas, **GLOBALS** e **TEXT\_IO** e, dessa forma, especifica as duas

```
with GLOBALS, TEXT_IO;
procedure EXEMPLO is
    ...
end EXEMPLO;
```

O FORTRAN 90 também permite compilação separada de seus subprogramas e módulos. A compilação separada da Ada será discutida com mais detalhes no Capítulo 11, depois de termos discutido as facilidades de abstração de dados dessa linguagem.

Em algumas linguagens, mais notavelmente no C e no FORTRAN 77, a compilação independente é permitida. Com a **compilação independente**, unidades de programa podem ser compiladas sem informações sobre quaisquer outras unidades de programa.

A característica mais importante da compilação independente é que as interfaces entre as unidades compiladas separadamente não são verificadas quanto à coerência de tipos. A interface de uma sub-rotina FORTRAN 77 é sua lista de parâmetros. Quando uma sub-rotina é compilada independentemente, os tipos de seus parâmetros não são armazenados com seu código compilado ou em uma biblioteca. Portanto, quando outro programa que chama essa sub-rotina é compilado, os tipos dos parâmetros reais nas chamadas não podem ser verificados em relação aos tipos dos parâmetros formais da sub-rotina, mesmo que o código de máquina para a sub-rotina chamada esteja disponível.

Isso não é de surpreender no FORTRAN 77. Mesmo quando o programa chama um subprograma que é compilado a partir do mesmo arquivo, os dois são, com efeito, se não de fato, compilados independentemente. Assim, a interface de parâmetros entre unidades de programa FORTRAN 77 nunca é verificada quanto à compatibilidade de tipos.

Algumas linguagens não oferecem nem compilação separada, nem independente, significando que somente a unidade de compilação é um programa completo. O FORTRAN II e a versão original do Pascal estão entre essas linguagens. Eis uma grave restrição para a linguagem, tornando-a virtualmente inútil para aplicações industriais. É evidente que a compilação independente, não obstante permitir interfaces de unidade de programa não-verificadas, é melhor do que não possuir nenhum meio de compilar partes de programas. As versões posteriores do FORTRAN e do Pascal reconheceram isso.

## 9.10 Questões de Projeto Referentes a Funções

As duas questões de projeto seguintes são específicas a funções:

- Efeitos colaterais são permitidos?
- Quais tipos de valores podem ser retornados?

### 9.10.1 Efeitos Colaterais Funcionais

Devido aos problemas dos efeitos colaterais de funções que são chamadas em expressões, conforme descrevemos no Capítulo 5, os parâmetros para funções sempre devem estar em modo entrada. Algumas linguagens, de fato, exigem isso; por exemplo, as funções Ada podem ter somente parâmetros formais em modo entrada. Isso efetivamente impede que uma função cause efeitos colaterais por parâmetros ou por apelidos de parâmetros e globais. Na maioria das outras linguagens, entretanto, as funções podem ter ou parâmetros de passagem por valor ou por referência, permitindo, assim, funções que causam efeitos colaterais e apelidos.

### 9.10.2 Tipos de Valores Retornados

A maioria das linguagens de programação imperativas restringe os tipos que podem ser retornados por suas funções. As funções FORTRAN 77 e Pascal permitem que somente tipos não-estruturados sejam retornados. O C permite que qualquer tipo seja retornado por suas funções, exceto vetores e funções. Ambos podem ser manipulados pelos valores de retorno do tipo do ponteiro. O C++ é igual ao C, mas também permite que tipos definidos pelo usuário ou classes sejam retornados de suas funções. A Ada está sozinha entre as linguagens imperativas em termos de que suas funções podem retornar valores de qualquer tipo.

## 9.11 Acessando Ambientes Não-Locais

Ainda que grande parte da comunicação exigida entre os subprogramas possa ser realizada pelos parâmetros, a maioria das linguagens oferece algum outro método de acessar variáveis de ambientes externos.

As **variáveis não-locais** de um subprograma são visíveis dentro dele, mas não declaradas localmente. As **variáveis globais** são visíveis em todas as unidades de programa. No Capítulo 5, dois métodos de acessar variáveis não-locais foram discutidos: o escopo estático e o escopo dinâmico. Os problemas com esses métodos serão revisados nos parágrafos seguintes.

O principal problema de usar o escopo estático como o meio de compartilhar variáveis não-locais é o seguinte: uma boa quantidade da estrutura do programa pode ser ditada pela acessibilidade de subprogramas e de variáveis não-locais a outros subprogramas, em vez de soluções de problema bem trabalhadas. Adicionalmente, em todos os casos, é fornecido mais acesso a não-locais do que o necessário. Lembre-se de nossos exemplos desse problema no Capítulo 5.

Dois tipos de problemas de programação decorrem diretamente do escopo dinâmico. Primeiro, durante o intervalo de tempo que se inicia quando a execução de um subprograma é iniciada e que se encerra quando ela é finalizada, as variáveis locais do subprograma são visíveis a quaisquer outros subprogramas em execução, independentemente da proximidade textual deles. Não existe nenhuma maneira de proteger as variáveis locais dessa acessibilidade. Os subprogramas são sempre executados no ambiente imediato do chamar. Por causa disso, o escopo dinâmico resulta em programas menos confiáveis do que o escopo estático.

Um segundo problema com o escopo dinâmico é a incapacidade de fazer estaticamente a verificação de tipos de referências a não-locais, porque não é possível determinar estaticamente as declarações para uma variável referenciada como não-local.

A implementação de acessos a variáveis não-locais será discutida detalhadamente no Capítulo 10.

### 9.11.1 Blocos COMMON do FORTRAN

O FORTRAN oferece acesso a blocos de armazenamento global por meio de sua instrução COMMON. Pode haver qualquer número desses blocos, sendo que todos, menos um, devem ser nomeados (pode haver um COMMON não-nomeado, muitas vezes chamado COMMON em branco, que tem algumas propriedades diferentes dos blocos COMMON nomeados). Qualquer subprograma que deseje acessar ou criar um bloco comum tem uma instrução COMMON que dá nome ao bloco e apresenta uma lista das variáveis que ela pode usar para acessar o seu armazenamento. Um bloco comum é criado quando a primeira instrução COMMON que menciona o nome dele é encontrada pelo compilador.

Qualquer número de subprogramas pode incluir uma instrução COMMON que especifica o mesmo bloco. Adicionalmente, cada subprograma que especifica o bloco pode especificar uma lista diferente de variáveis, cujo número e tipos podem não estar relacionados com os de outros subprogramas. Por exemplo, é perfeitamente legal ter-se as declarações

```
REAL A(100)
INTEGER B(250)
COMMON /BLOCO1/ A, B
```

em um subprograma, e as declarações

```
REAL C(50), D(100)
INTEGER E(200)
COMMON /BLOCO1/ C, D, E
```

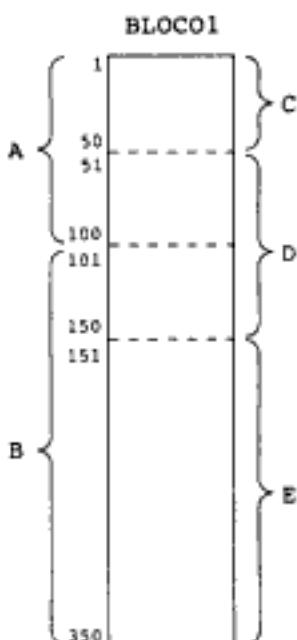
em outro subprograma. O identificador delimitado por barras diagonais, BLOCO1, é o nome do bloco. As duas visualizações das variáveis em BLOCO1 são mostradas na Figura 9.3 (ver p. 366). Note que presumimos que as variáveis integer e real ocupam a mesma quantidade de espaço.

Isso pode fazer sentido somente se BLOCO1 for usado simplesmente como um armazenamento compartilhado, não como dados compartilhados. De fato, o compartilhamento do armazenamento é a principal razão para permitir isso. Na maioria dos casos, os dados em um bloco COMMON devem ser compartilhados usando os mesmos nomes e tipos de variável. Um simples erro de disposição na lista de variáveis em uma instrução COMMON nessa situação pode provocar um inadvertido compartilhamento do armazenamento entre variáveis de diferentes tipos, o que pode causar um erro de difícil localização.

A definição do FORTRAN 90 declara que COMMON é um dos “recursos desaprovados”, o que significa que ela pode ser incluída em somente mais uma versão do FORTRAN. EQUIVALENCE também é um recurso desaprovado no FORTRAN 90.

### 9.11.2 Declarações e Módulos Externos

A Ada usa o escopo estático como um meio de compartilhar dados entre unidades de programa. Oferece um método alternativo de compartilhamento de dados ao permitir que



**FIGURA 9.3** Duas visualizações de um bloco COMMON.

unidades especifiquem os módulos externos para os quais é exigido acesso. Usando esse método, cada módulo pode especificar exatamente os outros módulos para os quais é necessário acesso, nem mais, nem menos. É permitido ao usuário especificar um nome de módulo externo, que então proverá o acesso a todos os seus tipos, variáveis e procedimentos.

O FORTRAN 90 também inclui módulos que podem proporcionar compartilhamento seletivo e verificado quanto ao tipo de dados não-locais.

Esses recursos de linguagem serão discutidos mais completamente no Capítulo 11 em conjunto com a abstração de dados.

Em linguagens orientadas a objeto, como o C++ e o Java, uma classe pode ser usada para encapsular coleções de dados para ser compartilhadas entre outras classes. As classes serão discutidas detalhadamente no Capítulo 12.

No C, não há aninhamento de subprogramas, de modo que há somente um único nível de escopo de subprograma. Variáveis globais podem ser criadas colocando-se suas declarações fora das definições da função. Garante-se acesso a uma variável de uma função que declara que tal variável deve ser externa, por meio de uma instrução **extern**. Todas as funções cujas definições seguem variáveis declaradas globalmente em um arquivo de código têm acesso a essas variáveis sem as declarar como externas. Esse método é elegante, mas inseguro e oferece bem mais acesso do que necessário.

Variáveis globais definidas em outros arquivos de código C incluídos em um programa também podem ser acessadas por funções que as declaram como externas. Dessa forma, escopos separados, não-compartilhados, podem ser introduzidos por meio de compilação separada de módulos. Com tal método, o acesso a variáveis nesses módulos compilados separadamente é restrito às funções que declaram as variáveis como externas.

## 9.12 Operadores Sobrecarregados Definidos pelo Usuário

Os operadores podem ser sobreescarregados pelo usuário em programas Ada e C++. Como um exemplo disso, considere a seguinte função Ada que sobreescarrega o operador de multiplicação (\*) para computar o produto escalar de dois vetores. O produto escalar de dois vetores é a soma dos produtos de cada um dos pares de elementos correspondentes dos dois vetores. Suponhamos que `TIPO_VETOR` tenha sido definido como um tipo de vetor com elementos `INTEIROS`.

```
function "*"(A, B : in TIPO_VETOR) return INTEGER is
    SOMA : INTEGER := 0;
begin
    for INDICE in A'range loop
        SOMA := SOMA + A(INDICE) * B(INDICE);
    end loop; - for INDICE...
    return SOMA;
end "*";
```

O produto escalar, conforme foi especificado nessa definição de função, será computado em todos os lugares onde o asterisco apareça com dois operandos do `TIPO_VETOR`. O asterisco pode ser ainda mais sobreescarregado qualquer número de vezes, contanto que as funções definidoras tenham protocolos únicos.

A função de produto escalar também poderia ser escrita em C++. O protótipo dessa função poderia ser

```
int operador *(const vector &a, const vector &b, int tamanho);
```

Naturalmente, surge a questão: quanta sobreescrita de um operador é boa, ou pode-se ter muita sobreescrita? A resposta é, em grande parte, uma questão de gosto. O argumento contra a demasiada sobreescrita de operador é principalmente uma questão de legibilidade. Em muitos casos, pode ser mais legível chamar uma função para executar uma operação do que usar um operador mais freqüentemente solicitado para outros tipos de operando. Mesmo no caso do produto escalar, pode ser muito fácil esquecer o que está envolvido quando uma simples instrução de atribuição, como

```
C := A * B
```

é encontrada em um programa. É fácil presumir que `A`, `B` e `C` são simples escalares.

Outra consideração é o processo de construir um sistema de software a partir de módulos criados por grupos diferentes. Se os diferentes grupos sobreescravassem os mesmos operadores de diversas maneiras, tais diferenças evidentemente precisariam ser eliminadas antes da montagem do sistema.

## 9.13 Co-Rotinas

Uma **co-rotina** é um tipo especial de subprograma. Em vez da relação mestre-escravo entre o subprograma chamador e o chamado que existe nos subprogramas convencionais, as co-rotinas chamadoras e chamadas estão em uma base mais igual. De fato, o mecanismo

de controle de co-rotinas freqüentemente é chamado modelo de controle de unidades simétrico.

A origem real do conceito de controle de unidades simétrico é difícil de determinar. Uma das primeiras aplicações de co-rotinas publicadas ocorreu na área da análise sintática (Conway, 1963). A primeira linguagem de programação de alto nível a incluir facilidades para co-rotinas foi a SIMULA 67. Lembre-se de que o propósito original da SIMULA era a simulação de sistemas, que, muitas vezes, exige a modelagem de processos independentes. Essa necessidade foi a motivação para o desenvolvimento das co-rotinas da SIMULA 67. Outras linguagens que suportam co-rotinas são a BLISS (Wulf et al., 1971), a INTERLISP (Teitelman, 1975) e o Modula-2 (Wirth, 1985).

As co-rotinas têm múltiplos pontos de entrada, controlados pelas próprias co-rotinas. Elas também têm os meios para manter seus *status* entre as ativações. Isso significa que as co-rotinas devem ser sensíveis à história e, dessa forma, ter variáveis locais estáticas. As execuções subsequentes de uma co-rotina freqüentemente têm início em pontos que não correspondem ao seu inicio. Por causa disso, a invocação de uma co-rotina é chamada **retomada** em vez de chamada.

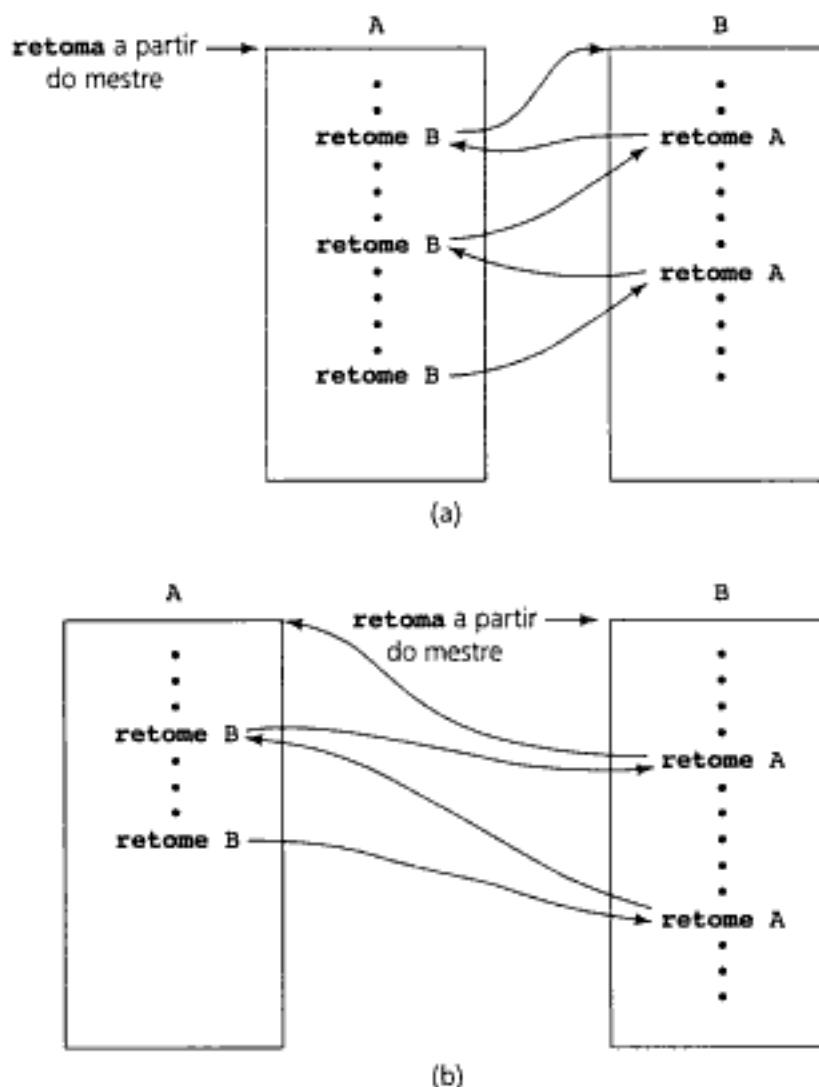
Uma das características usuais dos subprogramas é mantida nas co-rotinas. Somente uma delas é executada de fato em determinado momento. Em vez de executarem até o seu final, as co-rotinas freqüentemente o fazem parcialmente e depois transferem o controle para alguma outra. Quando reiniciada, a co-rotina retoma a execução logo depois da instrução usada para transferir o controle para outro lugar. Esse tipo de seqüência de execução relaciona-se com a maneira pela qual os sistemas operacionais de múltiplos processadores trabalham. Ainda que possa haver somente um processador, todos os programas em execução nesses sistemas parecem rodar de maneira concorrente enquanto compartilham o processador. No caso das co-rotinas, isso, às vezes, é chamado *quase concorrência*.

Tipicamente, as co-rotinas são criadas em uma aplicação por uma unidade de programa chamada **unidade-mestra**, a qual não é uma co-rotina. Quando criadas, elas executam seu código de inicialização e, depois, retornam o controle a essa unidade-mestra. Quando toda a família de co-rotinas é construída, o programa-mestre retoma uma das co-rotinas, e os membros da família retomam, entre si, em certa ordem, até que o seu trabalho seja concluído se, de fato, ele puder ser concluído. Se a execução de uma co-rotina atingir o final de sua seção de código, o controle será transferido para a unidade-mestra que a criou. Esse é o mecanismo para encerrar a execução da coleção de co-rotinas, quando isso for desejável. Em alguns programas, elas rodam sempre que o computador estiver em funcionamento.

Um exemplo de problema que pode ser resolvido com esse tipo de coleção de co-rotinas é a simulação de um jogo de cartas. Suponhamos que o jogo tenha quatro jogadores usando a mesma estratégia para jogar. Esse jogo pode ser simulado fazendo com que uma unidade de programa-mestre crie uma família de co-rotinas, cada uma com uma coleção, ou mão, de cartas. O programa-mestre poderia, então, iniciar a simulação retomando uma das co-rotinas jogadoras, a qual, depois de haver jogado sua vez, poderia retomar a seguinte e assim por diante, até que o jogo termine.

A mesma forma de instrução de retomada pode ser usada tanto para iniciar como para reiniciar a execução de uma co-rotina.

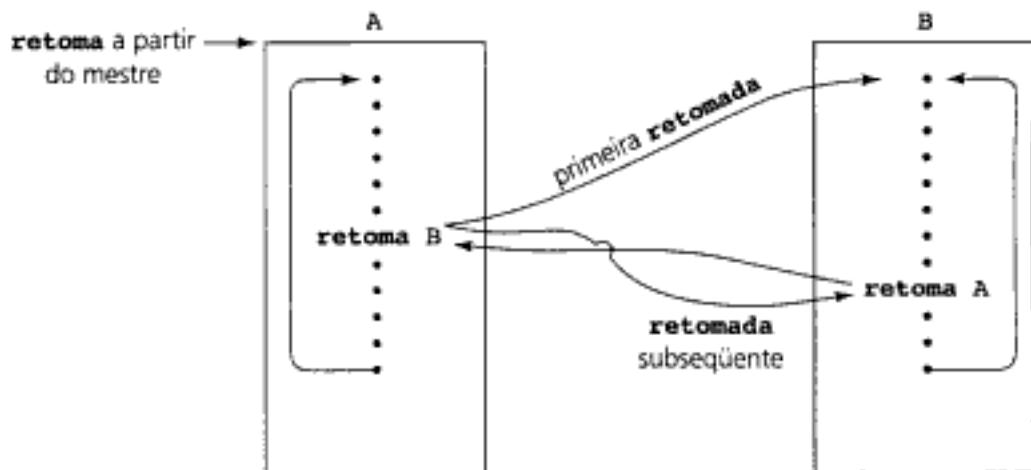
Suponhamos que as unidades de programa **A** e **B** sejam co-rotinas. A Figura 9.4 (ver p. 369) mostra duas maneiras pelas quais uma seqüência de execução envolvendo **A** e **B** poderia ser bem-sucedida.



**FIGURA 9.4** Duas seqüências de controle de execução possíveis para duas co-rotinas sem laços.

Na Figura 9.4a, a execução da co-rotina A é iniciada pela unidade-mestre. Depois de alguma execução, A inicia B. Quando a co-rotina B faz com que, pela primeira vez, o controle retorne à co-rotina A, a semântica é que A prossegue a partir de onde terminou sua última execução. Em especial, suas variáveis locais têm os valores deixados a elas pela ativação anterior. A Figura 9.4b mostra uma seqüência de execução alternativa das co-rotinas A e B. Neste caso, B é iniciada pela unidade-mestre.

Em vez de ter os padrões mostrados na Figura 9.4, uma co-rotina, muitas vezes, tem um laço que contém uma retomada. A Figura 9.5 (ver p. 370) mostra a seqüência de execução de tal cenário. Nesse caso, A é iniciada pela unidade-mestre. Dentro de seu laço principal, A retoma B, a qual, por sua vez, retoma A dentro de seu laço principal.



**FIGURA 9.5** Seqüência de execução de co-rotinas com laços.

## RESUMO

As abstrações de processo são representadas nas linguagens de programação pelos subprogramas. Uma definição destes descreve as ações representadas por eles. Uma chamada a subprograma ativa essas ações.

Parâmetros formais são os nomes que os subprogramas usam para fazer referência aos parâmetros reais dados em chamadas a subprograma.

Subprogramas podem ser funções, as quais têm o modelo das funções matemáticas e são usadas para definir novas operações, ou podem ser procedimentos, que definem novas instruções.

As variáveis locais em subprogramas podem ser alocadas dinamicamente a partir de uma pilha, oferecendo suporte para recursão, ou podem ser alocadas estaticamente, oferecendo eficiência e variáveis locais sensíveis à história.

Há três modelos semânticos fundamentais de passagem de parâmetros — modo entrada, modo saída e modo entrada/saída e uma série de abordagens de implementação.

Pode ocorrer apelido quando parâmetros passados por referência são usados, tanto entre dois ou mais parâmetros como entre um destes e uma variável não-local acessível.

O acesso a variáveis não-locais é fornecido de diversas maneiras: por meio de declarações externas, por blocos de dados globais, por módulos externos e por escopo estático e dinâmico.

Parâmetros que são nomes de subprogramas fornecem um serviço necessário, mas são difíceis de entender. A opacidade situa-se no ambiente de referenciamento disponível quando um subprograma passado como parâmetro está em execução.

A Ada e o C++ permitem tanto sobrecarga de subprograma como de operador. Subprogramas podem ser sobre carregados, contanto que as várias versões percam sua ambiguidade pelos tipos de seus parâmetros e pelos valores retornados. As definições de funções podem ser usadas para criar significados adicionais para os operadores.

Os subprogramas em Ada e em C++ podem ser genéricos, usando polimorfismo paramétrico, de modo que os tipos desejados de seus objetos de dados possam ser passados para o compilador, que poderá, então, construir unidades para os tipos solicitados.

Uma co-rotina é um subprograma especial que tem múltiplas entradas. Elas podem ser usadas para oferecer execução intercalada de subprogramas.

## QUESTÕES DE REVISÃO

1. Quais são as três características gerais dos subprogramas?
2. O que significa um subprograma estar ativo?
3. O que é um perfil de parâmetro? O que é um protocolo de subprograma?
4. O que são parâmetros formais? O que são parâmetros reais?
5. Quais são as vantagens e as desvantagens dos parâmetros de palavra-chave?
6. Quais são as questões de projeto referentes aos subprogramas?
7. Quais são as vantagens e as desvantagens das variáveis locais dinâmicas?
8. Quais são os três modelos semânticos de passagem de parâmetros?
9. Quais são os modos, os modelos conceituais de transferência, as vantagens e as desvantagens dos métodos de passagem de parâmetros por valor, por valor-resultado, por referência e por nome?
10. De quais maneiras podem ocorrer apelidos com parâmetros passados por referência?
11. Qual é a diferença na maneira pela qual o C e o ANSI C lidam com um parâmetro real cujo tipo não é idêntico ao do formal correspondente?
12. Qual é o problema com a política da Ada de permitir que os implementadores decidam quais parâmetros passar pela referência e quais passar pelo valor-resultado?
13. Quais são as duas considerações de projeto fundamentais referentes aos métodos de passagem de parâmetros?
14. Quais são as duas questões que surgem quando nomes de subprograma são parâmetros?
15. Defina vinculação rasa e vinculação profunda para ambientes de referenciamento de subprogramas passados como parâmetros.
16. O que é um subprograma sobrecarregado?
17. O que é polimorfismo paramétrico?
18. O que faz uma função modelo C++ ser instanciada?
19. Defina compilação separada e compilação independente.
20. Quais são as questões de projeto referentes às funções?
21. De quais maneiras as co-rotinas são diferentes dos subprogramas convencionais?

## PROBLEMAS

1. Quais são os argumentos favoráveis e quais os contrários a que um programa crie definições adicionais para operadores existentes, como pode ser feito em Ada e em C++? Você acha que essa sobrecarga de operador definida pelo usuário é boa ou má? Sustente sua resposta.
2. Na maioria das implementações FORTRAN IV, os parâmetros eram passados pela referência usando somente a transmissão do caminho de acesso. Exponha tanto as vantagens como as desvantagens dessa opção de projeto.
3. Argumente em defesa da decisão dos projetistas da Ada 83 de permitirem que o implementador escolha entre implementar parâmetros no modo **entrada/saída** por cópia ou por referência.
4. O FORTRAN tem dois tipos ligeiramente diferentes de COMMON: em branco e nomeados. Uma diferença entre eles é que os blocos COMMON em branco não podem ser inicializados durante a compilação. Veja se você pode determinar a razão pela qual o COMMON em branco foi projetado dessa maneira. Dica: essa decisão de projeto foi tomada no início do desenvolvimento do FORTRAN, quando as memórias dos computadores eram bem pequenas.
5. Suponhamos que você deseje escrever um subprograma que imprima um cabeçalho em uma nova página de saída, juntamente com um número de página que é 1 na primeira ativação e que aumenta 1 em cada ativação subsequente. Isso pode ser feito sem parâmetros e sem referência a variáveis não-locais em Pascal? Pode ser feito em FORTRAN? Pode ser feito em C?
6. O FORTRAN permite que um subprograma tenha múltiplas entradas. Por que isso, às vezes, é uma capacidade valiosa?
7. Escreva um procedimento Pascal SOMADOR que faça a adição de dois vetores de números inteiros. Ele deve ter somente dois parâmetros, os quais são os vetores a serem adicionados. O

segundo vetor também conterá o vetor da soma na saída. Ambos os parâmetros devem ser passados por referência. Teste SOMADOR com a chamada

SOMADOR(A, A)

em que A é um vetor a ser adicionado a si mesmo. Explique os resultados ao executar esse programa.

8. Repita o Problema 7 usando C.
9. Considere o procedimento BIGSUB da Seção 9.5.2.5. Mude as duas atribuições do vetor LISTA para
 

```
LISTA[1] := 3
LISTA[2] := 1
```

 Execute manualmente o novo programa sob as seguintes pressuposições e compare os valores resultantes do vetor LISTA em BIGSUB depois do retorno de SUB:
  - a. Parâmetros são passados por valor
  - b. Parâmetros são passados por referência
  - c. Parâmetros são passados por nome
  - d. Parâmetros são passados por valor-resultado
10. Considere o seguinte programa escrito na sintaxe C:

```
void principal() {
    int valor = 2, lista[5] = {1, 3, 5, 7, 9};
    troca(valor, lista[0]);
    troca(lista[0], lista[1]);
    troca(valor, lista[valor]);
}
void troca(int a, int b) {
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

Para cada um dos métodos de passagem de parâmetros seguintes, quais são todos os valores das variáveis valor e lista depois de cada uma das três chamadas a troca?

- a. Passados por valor
- b. Passados por referência
- c. Passados por nome
- d. Passados por valor-resultado
11. Apresente um argumento contrário a oferecer tanto variáveis locais estáticas como dinâmicas em subprogramas.
12. Argumente contrariamente ao fato do C oferecer somente subprogramas de função.
13. Em um livro didático sobre o FORTRAN, aprenda a sintaxe e a semântica das funções de instrução. Justifique a existência das mesmas no FORTRAN.
14. Estude os métodos de sobrecarga de operador definida pelo usuário no C++ e na Ada e escreva um relatório comparando os dois, usando nossos critérios para avaliar linguagens.
15. Considere o seguinte procedimento ALGOL 60, chamado Dispositivo de Jensen, em homenagem a J. Jensen, do Regnecentralen de Copenhague, que o projetou em 1960:

```
real procedure SOMA(SOMADOR, INDICE, COMPRIMENTO);
  value COMPRIMENTO
  real SOMADOR;
  integer INDICE, COMPRIMENTO;
begin
  real SOMATEMP;
  SOMATEMP := 0.0;
  for INDICE := 1 step 1 until COMPRIMENTO do
    SOMATEMP = SOMATEMP + SOMADOR;
```

```
Soma := SOMATEMP  
end;
```

O que é retornado para cada uma das chamadas seguintes a **SOMA**, lembrando que os parâmetros são passados por nome e observando que o valor de retorno é definido ao atribuí-lo ao nome do subprograma?

- a. **SOMA(A, I, 100)**, em que A é uma escalar
- b. **SOMA(A[I]\*A[I], I, 100)**, em que A é um vetor de 100 elementos
- c. **SOMA(A[I]\*B[I], I, 100)**, em que A e B são vetores de 100 elementos

Hidden page

# Capítulo 10

## Implementando Subprogramas



### **John Kemeny**

John Kemeny e seu colega Thomas Kurtz desenvolveram compiladores para diversos dialetos do ALGOL e do FORTRAN em Dartmouth no inicio da década de 60. Em 1963, Kemeny iniciou o projeto do BASIC, que se tornou operacional em 1964.

- 10.1** A Semântica Geral das Chamadas e dos Retornos
- 10.2** Implementando Subprogramas FORTRAN 77
- 10.3** Implementando Subprogramas em Linguagens Assemelhadas ao ALGOL
- 10.4** Blocos
- 10.5** Implementando o Escopo Dinâmico
- 10.6** Implementando Parâmetros que são Nomes de Subprograma

A finalidade deste capítulo é explorar métodos para implementar subprogramas. A discussão proporcionará ao leitor uma percepção sobre o “funcionamento” dessas linguagens e também porque o ALGOL 60 foi um desafio para os insuspeitos escritores de compiladores do início da década de 60. Iniciamos com o tipo mais simples de subprograma — os do FORTRAN 77 — e avançamos para os subprogramas mais complicados das linguagens de escopo estático, como o FORTRAN 90, o Delphi, o Pascal e a Ada. O acréscimo de dificuldade de implementar subprogramas nestas linguagens é causado pela necessidade de incluir suporte para recursão e mecanismos para acessar variáveis não-locais.

Dois métodos para acessar variáveis não-locais em linguagens de escopo estático, os encadeamentos estáticos e os *displays* serão discutidos detalhadamente e comparados. Técnicas para implementar blocos terão uma abordagem breve. Diversos métodos para implementar acesso a variáveis não-locais em uma linguagem de escopo dinâmico serão discutidos. Por fim, haverá a descrição de um método para implementar parâmetros que são nomes de subprograma.

## 10.1 A Semântica Geral das Chamadas e dos Retornos

As operações de chamada e de retorno a subprogramas de uma linguagem são chamadas conjuntamente de **ligação (linkage) de subprograma**. Qualquer método de implementação para subprogramas deve basear-se na semântica da ligação de subprograma.

Uma chamada a subprograma em uma linguagem típica tem numerosas ações associadas a ela. A chamada deve incluir o mecanismo para quaisquer métodos de passagem de parâmetros usados. Se as variáveis locais não forem estáticas, a chamada deverá fazer com que seja alocado armazenamento para as variáveis declaradas no subprograma chamado e vinculá-las a ele. Ela deverá salvar o *status* de execução da unidade de programa chamadora e organizar-se para transferir o controle para o código do subprograma e assegurar que esse controle possa retornar para o lugar apropriado quando a execução do subprograma for concluída. Por fim, a chamada deverá fazer com que algum mecanismo seja criado para fornecer acesso a variáveis não-locais visíveis ao subprograma chamado.

As ações exigidas do retorno de um subprograma também são complicadas. Se ele tiver parâmetros modo saída implementados por cópia, a primeira ação do processo de retorno será transferir os valores locais dos parâmetros formais associados para os parâmetros reais. Em seguida, ele deverá desalocar o armazenamento usado para as variáveis locais e restabelecer o *status* de execução da unidade de programa chamadora. Alguma ação deverá ser, então, posta em prática para devolver o mecanismo usado para referências não-locais à configuração que ele tinha antes da chamada. Por fim, o controle deve ser devolvido para a unidade de programa chamadora.

## 10.2 Implementando Subprogramas FORTRAN 77

Iniciamos com a situação relativamente simples dos subprogramas FORTRAN 77. Todo o referenciamento de variáveis não-locais nele dá-se pelo COMMON. Uma vez que COMMON não é relacionado ao mecanismo de ligação de subprograma, não será discutido aqui. Outra característica simplificadora do FORTRAN 77 é que os subprogramas não podem ser recur-

sivos. Além disso, na maioria das implementações, as variáveis declaradas em subprogramas são estaticamente alocadas.

A semântica de uma chamada a subprograma FORTRAN 77 exige as seguintes ações:

1. Salvar o status de execução da unidade de programa atual.
2. Executar o processo de passagem de parâmetros.
3. Passar o endereço de retorno para o chamado.
4. Transferir o controle para o chamado.

A semântica de um retorno de subprograma FORTRAN 77 exige as seguintes ações:

1. Se forem usados parâmetros passados por resultado, os valores atuais desses parâmetros serão transferidos para os parâmetros reais correspondentes.
2. Se o subprograma for uma função, o valor funcional será transferido para um lugar acessível ao chamador.
3. O status de execução do chamador será restabelecido.
4. O controle será transferido novamente para o chamador.

As ações de chamada e de retorno exigem armazenamento para:

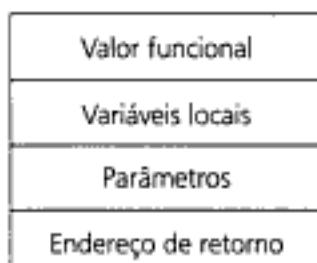
- informação do status sobre o chamador
- parâmetros
- endereço de retorno
- valor funcional para subprogramas da função

Essas, juntamente com as variáveis locais e com o código do subprograma, formam o conjunto completo de informações de que um subprograma precisa para executar e para, depois, retornar o controle para o chamador.

Um subprograma FORTRAN 77 consiste em duas partes distintas: seu código real, que é constante, e suas variáveis locais e os dados listados acima, que podem modificar-se quando o subprograma for executado. Ambas as partes têm tamanhos fixos.

O formato, ou o *layout*, da parte não-código de um subprograma é chamado **registro de ativação**, porque os dados que ele descreve somente são relevantes durante a ativação 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 deste.

Uma vez que o FORTRAN 77 não suporta recursão, pode haver somente uma versão ativa de determinado subprograma de cada vez. Portanto, pode haver somente uma única instância do registro de ativação de um subprograma. Um layout possível para os registros de ativação do FORTRAN 77 é mostrado na Figura 10.1. O status de execução salvado do chamador é omitido aqui e no restante deste capítulo simplesmente porque não é relevante para a discussão.

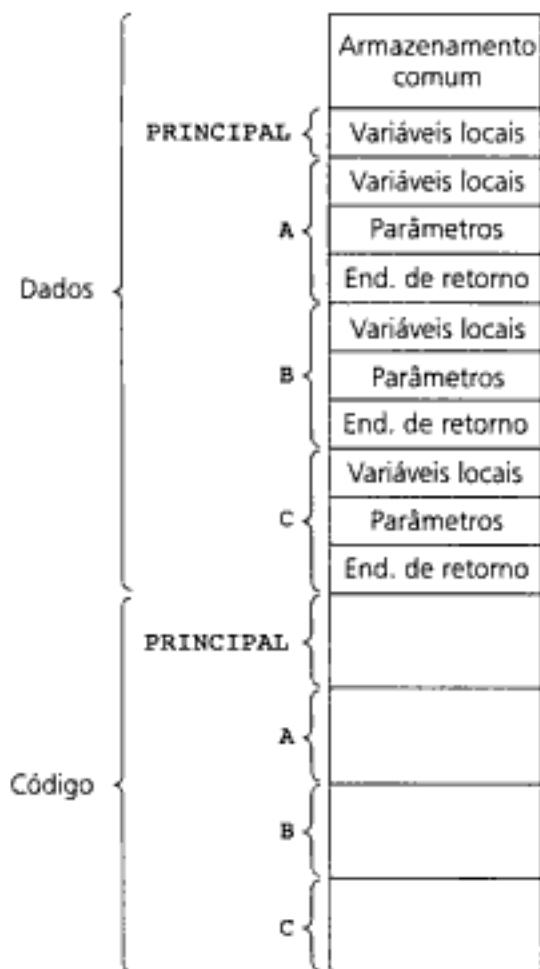


**FIGURA 10.1** Um registro de ativação FORTRAN 77.

Uma vez que uma instância de registro de ativação de um subprograma FORTRAN 77 tem tamanho fixo, ela pode ser alocada estaticamente. De fato, ela poderia ser anexada à parte de código do subprograma.

A Figura 10.2 mostra um programa FORTRAN 77 que consiste em um armazenamento COMMON, em um programa principal e em três subprogramas A, B e C. Embora a figura mostre todos os segmentos de código separados de todas as instâncias do registro de ativação, em alguns casos as instâncias do registro de ativação são anexadas aos seus segmentos de código associados.

A construção do programa FORTRAN 77 completo mostrado na Figura 10.2 não é feita inteiramente pelo compilador. De fato, por causa da compilação independente, as quatro unidades de programa — PRINCIPAL, A, B, e C — podem ter sido compiladas em dias diferentes, ou até em anos diferentes. No momento em que cada unidade é compilada, o código de máquina para ela, juntamente com uma lista de referências a subprogramas externos e variáveis COMMON dentro do código, é escrito em um arquivo. O programa executável mostrado na Figura 10.2 é montado pelo **linkeditor**, que faz parte do sistema operacional (os linkeditores também são chamados de *loaders* e *linker/loaders*). Quando o linkeditor é chamado para um programa principal, tem como primeira tarefa encontrar os arquivos que contêm os subprogramas traduzidos referenciados nesse programa, juntamente com suas instâncias de registro de ativação, e carregá-los na memória. Ele também deve deter-



**FIGURA 10.2** O código e os registros de ativação de um programa FORTRAN 77.

Hidden page

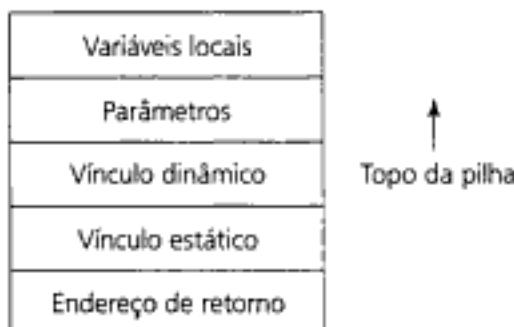
de fora deste e com uma ou mais chamadas recursivas. A recursão, por conseguinte, exige múltiplas instâncias de registros de ativação, uma para cada ativação de subprograma que possa existir ao mesmo tempo. Cada ativação exige sua própria cópia dos parâmetros formais e das variáveis locais alocadas dinamicamente, juntamente com os valores de retorno.

- Por fim, as linguagens assemelhadas ao ALGOL usam escopo estático para oferecer acesso a variáveis não-locais. O suporte para tais acessos a não-locais deve fazer parte do mecanismo de ligação.

O formato de um registro de ativação para determinado subprograma em uma linguagem de escopo estático é conhecido em tempo de compilação. Em procedimentos Pascal, o tamanho dos registros de ativação também é conhecido, porque todos os dados locais para um procedimento Pascal são de tamanho fixo. Isso não acontece em algumas outras linguagens, cujo tamanho de um vetor local pode depender do valor de um parâmetro real. Nesses casos, o formato é estático, mas o tamanho pode ser dinâmico. No caso das linguagens assemelhadas ao ALGOL, as instâncias do registro de ativação devem ser criadas dinamicamente. O registro de ativação típico para uma linguagem como o ALGOL é mostrado na Figura 10.3.

Uma vez que o endereço de retorno, o vínculo estático, o vínculo dinâmico e os parâmetros são colocados na instância do registro de ativação pelo chamador, essas entradas devem aparecer primeiro. Supomos, neste capítulo, que a pilha cresce para cima. Portanto, o endereço de retorno estará na base de um registro de ativação.

O endereço de retorno, muitas vezes, é composto de um ponteiro para o segmento de código do chamador e de um deslocamento nesse segmento de código da instrução seguinte à chamada. O **vínculo estático**, que, por vezes, é chamado ponteiro de escopo estático, aponta para a base da instância do registro de ativação de uma ativação do pai-estático. Ele é usado para acessos a variáveis não-locais. Os vínculos estáticos serão discutidos detalhadamente na Seção 10.3.4. O **vínculo dinâmico** é um ponteiro para o topo da instância do registro de ativação do chamador. Em linguagens de escopo estático, ele é usado na destruição da instância de registro de ativação atual, quando o procedimento conclui sua execução. O topo da pilha é ajustado para o vínculo dinâmico anterior. O vínculo dinâmico é necessário porque, em alguns casos, há outras alocações da pilha por um subprograma além de seu registro de ativação. Por exemplo, valores temporários de que a versão em linguagem de máquina do subprograma necessita podem ser alocados af. Assim, ainda que o tamanho do registro de ativação possa ser conhecido, ele não pode simplesmente ser subtraído do ponteiro para o topo da pilha para remover o registro de ativação. Os parâmetros reais no registro de ativação são os valores ou os endereços fornecidos pelo chamador.



**FIGURA 10.3** Um registro de ativação típico para uma linguagem assemelhada ao ALGOL.

As variáveis escalares locais são vinculadas ao armazenamento dentro de uma instância do registro de ativação. Variáveis locais que são estruturas, às vezes, são alocadas em outro lugar, e somente seus descritores e um ponteiro para esse armazenamento fazem parte do registro de ativação. As variáveis locais são alocadas e possivelmente inicializadas no subprograma chamado, de modo que aparecem por último.

Considere o seguinte procedimento Pascal esquemático:

```
procedure sub(var total : real; parte : integer);
  var lista : array [1..5] of integer;
      soma : real;
begin
  ...
end;
```

O registro de ativação para `sub` é mostrado na Figura 10.4.

A ativação de um procedimento exige a criação dinâmica de uma instância do registro de ativação para ele. Conforme afirmamos anteriormente, o formato do registro de ativação é fixado durante a compilação, ainda que seu tamanho possa depender da chamada em algumas linguagens diferentes do Pascal. Uma vez que a semântica das chamadas e dos retornos especifica que o subprograma chamado por último seja o primeiro a ser concluído, é razoável criar instâncias desses registros de ativação em uma pilha. Ela faz parte do sistema em tempo de execução e, portanto, é chamada *pilha de execução*, mesmo que apenas nos refiramos a ela comumente como *pilha*. Toda ativação de procedimento, seja recursivo ou não-recursivo, cria uma nova instância de um registro de ativação na pilha. Isso proporciona as necessárias cópias separadas dos parâmetros, das variáveis locais e do endereço de retorno.

Lembre-se, do Capítulo 9, de que um subprograma está **ativo** a partir do momento em que é chamado até quando a execução é concluída. No momento em que ele se torna

Local	soma
Local	lista [5]
Local	lista [4]
Local	lista [3]
Local	lista [2]
Local	lista [1]
Parâmetro	parte
Parâmetro	total
Vínculo dinâmico	
Vínculo estático	
Endereço de retorno	

**FIGURA 10.4** O registro de ativação para o procedimento `sub`.

inativo, seu escopo local deixa de existir e seu ambiente de referenciamento não é mais significativo. Sendo assim, nesse momento, sua instância de registro de ativação é destruída.

### 10.3.2 Um Exemplo Sem Recursão e Referências a Não-locais

Devido à complexidade de implementação da ligação de subprogramas, nós a consideramos em diversas etapas. Primeiro, examinamos um exemplo de programa que não referencia variáveis não-locais e não tem nenhuma chamada recursiva. Para esse exemplo, não é usado o vínculo estático. Depois, consideraremos como a recursão e o referenciamento não-local podem ser implementados.

Considere o seguinte exemplo de programa esquemático:

```

program PRINCIPAL_1;
  var P : real;
  procedure A(X : integer);
    var Y : boolean;
    procedure C(Q : boolean);
      begin { C }
      ... ←————— 3
      end; { C }
    begin { A }
    ... ←————— 2
    C(Y);
    ...
    end; { A }
  procedure B(R : real);
    var S, T : integer;
    begin { B }
    ... ←————— 1
    A(S);
    ...
    end; { B }
  begin { PRINCIPAL_1 }
  ...
  B(P);
  ...
end. { PRINCIPAL_1 }
```

A seqüência de chamadas a procedimento nesse programa é

```

PRINCIPAL_1 chama B
B chama A
A chama C
```

O conteúdo da pilha para os pontos rotulados 1, 2 e 3 são mostrados na Figura 10.5 (ver p. 383).

No ponto 1, somente as instâncias do registro de ativação para o programa PRINCIPAL\_1 e o procedimento B estão na pilha. Quando B chama A, uma instância do registro de ativação de A é criada na pilha. Quando A chama C, uma instância do registro de ativação de C é criada na pilha. Quando a execução de C encerra-se, a instância do seu registro de

Hidden page

para o vínculo dinâmico). A segunda variável local declarada estaria uma posição mais próxima do topo da pilha e assim por diante. Por exemplo, considere o exemplo do programa precedente. Em A, o deslocamento\_local de Y é 4. Similarmente, em B, o deslocamento\_local de S é 4; para T é 5.

### 10.3.3 Recursão

Considere o seguinte exemplo de programa C, que usa recursão para computar a função factorial:

```
int factorial(int n) {
    ←—————1
    if (n <= 1)
        return 1;
    else return (n * factorial(n - 1));
    ←—————2
}
void main() {
    int valor;
    valor = factorial(3);
    ←—————3
}
```

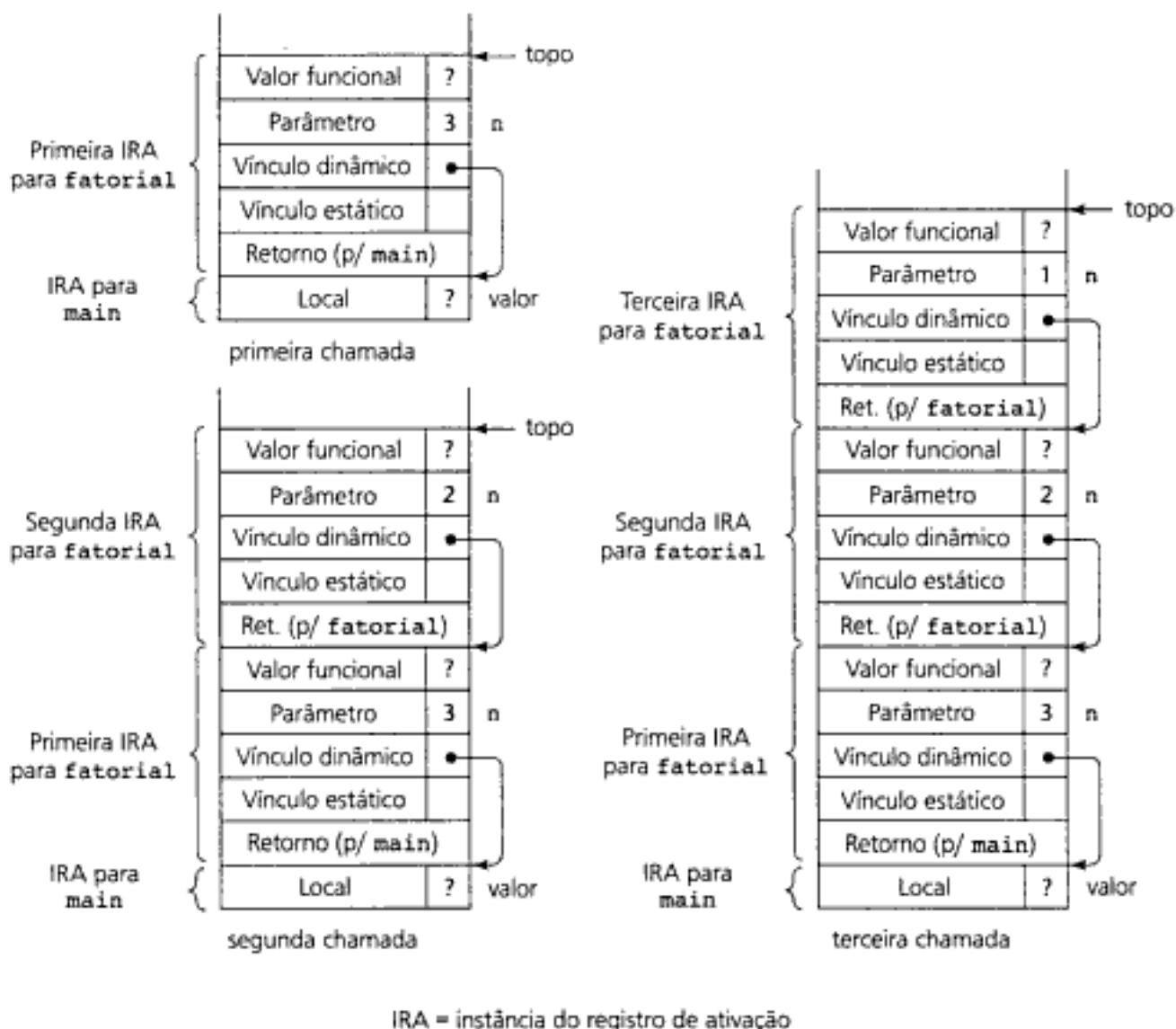
O formato do registro de ativação para a função factorial é mostrado na Figura 10.6. Note que ele tem uma entrada adicional para o valor retornado da função.

A Figura 10.7 (ver p. 385) mostra o conteúdo da pilha para as três vezes que a execução atinge a posição 1 na função factorial. Cada uma mostra mais uma ativação da função, com seu valor funcional não-definido. A primeira instância do registro de ativação tem o endereço de retorno para a função chamadora, **main**. As outras têm um endereço de retorno para a própria função; essas são para as chamadas recursivas.

A Figura 10.8 (ver p. 386) mostra o conteúdo da pilha referente às três vezes que a execução atinge a posição 2 na função factorial. A posição 2 pretende ser o momento depois que **return** é executado, mas antes que o registro de ativação tenha sido removido da pilha. Lembre-se de que o código para a função multiplica o valor atual do parâmetro *n* pelo retornado pela chamada recursiva à função. O primeiro retorno de factorial retorna 1. A instância do registro de ativação para ela tem o valor 1 para sua versão do parâmetro *n*. O

Valor funcional	N
Parâmetro	
Vínculo dinâmico	
Vínculo estático	
Endereço de retorno	

**FIGURA 10.6** Registro de ativação para factorial.



**FIGURA 10.7** Conteúdo da pilha na posição 1 em factorial.

resultado dessa multiplicação, 1, é retornado para a segunda ativação do `fatorial` para ser multiplicado pelo valor do paramétrico *n*, que é 2. O valor 2 é, então, retornado à primeira ativação do `fatorial` para ser multiplicado pelo valor do parâmetro *n*, que é 3, produzindo o valor funcional final 6, que é, então, retornado à primeira chamada a `fatorial` em `main`.

#### 10.3.4 Mecanismos para Implementar Referências a Não-locais

Existem duas técnicas principais de implementação para criar acessos a variáveis não-locais em uma linguagem de escopo estático: encadeamentos estáticos e displays. Ambas serão examinadas detalhadamente nas seções seguintes.

Uma referência a uma variável não-local exige um processo de acesso de dois passos. Todas as variáveis que podem ser acessadas não-localmente estão em instâncias do registro de ativação existentes e, portanto, em algum lugar na pilha. O primeiro passo do processo de acesso é localizar a instância do registro de ativação na pilha em que a variável foi

Hidden page

alocada. A segunda parte é usar o deslocamento \_local da variável (dentro da instância de registro de ativação) para acessá-la.

Encontrar a instância correta do registro de ativação é o mais interessante e o mais difícil dos dois passos. Primeiro, note que, em determinado subprograma, somente variáveis declaradas em escopos ancestrais estáticos são visíveis e podem ser acessadas. Além disso, as instâncias do registro de ativação de todos os ancestrais estáticos estão garantidas quanto a existirem na pilha quando variáveis existentes são referenciadas por um procedimento aninhado. Isso é garantido pelas regras semânticas estáticas das linguagens de escopo estático: um procedimento é “chamável” somente quando todas as suas unidades de programa ancestrais estáticas estão ativas. Se um ancestral estático particular não estivesse ativo, suas variáveis locais não seriam vinculadas ao armazenamento, de modo que não faria sentido permitir acessá-las.

Note que, não obstante o registro de ativação do escopo-pai precisar ter uma instância na pilha, ele não precisa aparecer de forma adjacente à instância de registro de ativação do filho. Isso é ilustrado em um exemplo na Seção 10.3.4.1.

A semântica das referências não-locais determina que a declaração correta seja a primeira a ser encontrada quando se examina os escopos envolventes, com os mais estreitamente aninhados em primeiro lugar. Assim, para suportar referências a não-locais, deve ser possível localizar todas as instâncias dos registros de ativação na pilha que correspondam a esses ancestrais estáticos. Essa observação leva aos dois métodos descritos nas seções seguintes.

Não encaminhamos a questão dos blocos até a Seção 10.4, de modo que no restante da Seção 10.3, todos os escopos são definidos por subprogramas. Uma vez que funções não podem ser aninhadas no C e no C++ (os únicos escopos estáticos são criados com blocos), a discussão desta seção não se aplica a essas linguagens.

#### 10.3.4.1 Encadeamentos Estáticos

Um **encadeamento estático** é uma cadeia de vínculos estáticos que conectam certas instâncias do registro de ativação na pilha.

Durante a execução de um procedimento *P*, o vínculo estático da sua instância do registro de ativação aponta para uma instância do registro de ativação da unidade de programa pai estática de *P*. Esse vínculo estático da instância aponta, por sua vez, para a instância do registro de ativação da unidade de programa avó estática de *P*, se houver uma. Assim, o encadeamento estático vincula todos os ancestrais estáticos de um subprograma em execução, com o pai estático em primeiro lugar. Tal encadeamento pode, evidentemente, ser usado para implementar os acessos a variáveis não-locais em linguagens de escopo estático.

Localizar a instância correta do registro de ativação de uma variável não-local, usando vínculos estáticos, é relativamente direto. Quando uma referência é feita a uma variável não-local, a instância do registro de ativação que a contém pode ser encontrada pesquisando-se o encadeamento estático até que uma instância ancestral estática do registro de ativação que contém a variável seja encontrada. Porém, é muito mais fácil do que isso. Uma vez que o aninhamento de escopos é conhecido no momento da compilação, o compilador pode determinar não somente que uma referência seja não-local, como também o tamanho do encadeamento estático necessário para alcançar a instância do registro de ativação que contém o objeto não-local.

Digamos que **profundidade\_estática** seja um número inteiro associado a um escopo estático que indica quão profundamente ele está aninhado no escopo mais externo. Um programa principal Pascal tem uma profundidade\_estática igual a 0. Se o procedimento

A for o único definido em um programa principal, sua profundidade\_estática será 1. Se o A contiver a definição de um procedimento aninhado B, a profundidade\_estática de B será 2.

O tamanho do encadeamento estático necessário para alcançar a instância correta do registro de ativação para uma referência não-local a uma variável x é exatamente a diferença entre a profundidade\_estática do procedimento que contém a referência a x e a profundidade\_estática do procedimento que contém a declaração para x. Essa diferença é chamada profundidade de aninhamento ou deslocamento do encadeamento da referência. A referência real pode ser representada por um par ordenado de números inteiros (deslocamento do encadeamento, deslocamento local), em que (deslocamento do encadeamento, deslocamento local) é o número de vínculos à instância correta do registro de ativação (deslocamento do encadeamento, deslocamento local, é descrito na Seção 10.3.2). Por exemplo, considere o seguinte programa esquemático:

```
program A;
  procedure B;
    procedure C;
    ...
  end; { C }
  ...
end; { B }
...
end; { A }
```

As profundidades estáticas de A, B e C são 0, 1 e 2, respectivamente. Se o procedimento C referenciar uma variável declarada em A, o deslocamento do encadeamento dessa referência seria 2 (a profundidade estática de C menos a profundidade estática de A). Se o procedimento C referenciar uma variável declarada em B, o deslocamento de encadeamento dessa referência seria 1. Referências a variáveis locais podem ser manipuladas usando-se o mesmo mecanismo, com um deslocamento de encadeamento igual a zero.

Para ilustrarmos o processo completo dos acessos não-locais, consideremos o seguinte programa esquemático em Pascal:

```
program PRINCIPAL_2;
  var X : integer;
  procedure BIGSUB;
    var A, B, C : integer;
    procedure SUB1;
      var A, D : integer;
      begin { SUB1 }
        A := B + C; ←—————1
      ...
    end; { SUB1 }
    procedure SUB2(X : integer);
      var B, E : integer;
      procedure SUB3;
        var C, E : integer;
        begin { SUB3 }
        ...
      SUB1;
      ...
    E := B + A; ←—————2
```

```

    end; { SUB3 }
begin { SUB2 }
...
SUB3;
...
A := D + E; ←—————3
end; { SUB2 }
begin { BIGSUB }
...
SUB2(7);
...
end; { BIGSUB }
begin { PRINCIPAL_2 }
...
BIGSUB;
...
end. { PRINCIPAL_2 }

```

A sequência de chamadas a procedimentos é:

```

PRINCIPAL_2 chama BIGSUB
BIGSUB chama SUB2
SUB2 chama SUB3
SUB3 chama SUB1

```

A situação da pilha quando a execução chega, pela primeira vez, ao ponto 1 nesse programa é mostrada na Figura 10.9 (ver p. 390).

Na posição 1 do procedimento **SUB1**, a referência é à variável local, **A**, não à não-local **A** de **BIGSUB**. Essa referência a **A** tem o par deslocamento de encadeamento, deslocamento local (0, 3). A referência a **B** é à não-local **B** de **BIGSUB**. Ela pode ser representada pelo par (1, 4). O deslocamento de encadeamento, deslocamento local é 4, porque um deslocamento 3 referenciaria a primeira variável local (**BIGSUB** não tem parâmetros). Note que, se o vínculo dinâmico fosse usado para fazer uma busca simples de uma instância do registro de ativação com uma declaração para a variável **B**, ele a encontraria 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. O vínculo estático, entretanto, aponta para o registro de ativação de **BIGSUB**, que tem a versão correta de **B**. A variável **B** de **SUB2** não está no ambiente de referenciamento neste ponto e não é (corretamente) acessível. A referência a **C** no ponto 1 é para a **C** definida em **BIGSUB**, representada pelo par (1, 5).

Depois que **SUB1** conclui sua execução, a instância do registro de ativação para **SUB1** é removida da pilha, e o controle retorna a **SUB3**. A referência à variável **E** na posição 2 em **SUB3** é local e usa o par (0, 4) para acesso. A referência à variável **B** é a declarada em **SUB2**, porque aquela é o ancestral estático mais próximo que contém essa 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 à variável **A** é ao **A** declarado em **BIGSUB**, porque tanto **SUB3** quanto seu pai estático **SUB2** não têm uma declaração para uma variável chamada **A**. Ela é referenciada com o par (2, 3).

Depois que **SUB3** conclui sua execução, a instância de registro de ativação para **SUB3** é removida da pilha, deixando somente as instâncias de registro de ativação para **PRINCIPAL\_2**, para **BIGSUB** e para **SUB2**. Na posição 3 em **SUB2**, a referência à variável **A** é ao **A** em **BIGSUB**, que tem a única declaração de **A** entre as rotinas ativas. Esse acesso é feito com

Hidden page

A essa altura, é razoável perguntarmos como o encadeamento estático é mantido durante a execução do programa. Se sua manutenção for demasiadamente complexa, o fato dele ser simples e eficiente será pouco importante. Nesta seção, presumimos que parâmetros passados por nome e parâmetros que são nomes de subprograma não são implementados.

O encadeamento estático deve ser modificado para cada chamada e para cada retorno a subprograma. A parte do retorno é banal: quando o subprograma encerra-se, sua instância do registro de ativação é removida da pilha. Depois, a nova instância do registro de ativação (topo) é a da unidade que chamou o subprograma cuja execução encerrou-se. Uma vez que o encadeamento estático desta instância do registro de ativação nunca é modificado, funciona corretamente, da mesma maneira que o fazia antes da chamada a outro subprograma. Por conseguinte, nenhuma outra ação é exigida.

A ação exigida em uma chamada a subprograma é mais complexa. Embora o escopo-pai correto seja facilmente determinado no momento da compilação, sua instância do registro de ativação mais recente deve ser localizada no momento da chamada. Isso pode ser feito examinando-se as instâncias do registro de ativação no encadeamento dinâmico até que a primeira do escopo-pai seja encontrada. Porém, tal busca pode ser evitada tratando-se as declarações de procedimento e referências exatamente como declarações de variáveis e referências. Quando o compilador encontra uma chamada a um procedimento, entre outras coisas, ele determina o que declarou o procedimento chamado, que deve ser um ancestral estático da rotina chamadora. Depois, ele computa a profundidade de aninhamento ou o número de escopos envolventes entre a chamadora e o procedimento que declarou o chamado. Essa informação é armazenada e pode ser acessada pela chamada ao procedimento durante a execução. No momento da chamada, o vínculo estático da instância do registro de ativação do procedimento chamado é determinado, movendo-se para baixo ao longo do encadeamento estático do chamador o número de vínculos igual à profundidade de aninhamento computada durante a compilação.

Considere novamente o programa PRINCIPAL\_2 e a situação da pilha mostrada na Figura 10.9. Na chamada a SUB1 em SUB3, o compilador determina a profundidade de aninhamento de SUB3 (o chamador) para que esteja dois níveis dentro do procedimento que declarou o procedimento chamado SUB1, que é BIGSUB. Quando a chamada a SUB1 em SUB3 é executada, essa informação é usada para definir o vínculo estático da instância do registro de ativação para SUB1. Este último é definido para direcionar para a instância do registro de ativação apontada pelo segundo vínculo estático no encadeamento estático a partir da instância do registro de ativação do chamador. Nesse caso, o chamador é SUB3, cujo vínculo estático aponta para a instância (a de SUB2) do registro de ativação de seu pai. O vínculo estático da instância do registro de ativação para SUB2 aponta para a instância do registro de ativação de BIGSUB. Desse modo, o vínculo estático para a nova instância do registro de ativação de SUB1 é definido para apontar para a instância do registro de ativação de BIGSUB.

Esse método funciona para toda ligação de procedimentos, exceto quando parâmetros que são nomes de subprograma estão envolvidos. Essa situação será discutida na Seção 10.6.

Uma crítica que se faz ao uso do encadeamento estático para acessar variáveis não-locais é que as referências a variáveis em escopos além do pai-estático são custosas. O encadeamento estático deve ser seguido, um vínculo por escopo envolvente, a partir da referência à declaração. Outra crítica está no fato de ser difícil para o programador trabalhando em um programa crítico quanto ao tempo, estimar os custos de referências a não-locais, porque o custo de cada uma delas depende da profundidade do seu aninhamento com o escopo da declaração. O que complica ainda mais o problema é que as modificações de código subsequentes podem modificar as profundidades do aninhamento, mudando o

Hidden page

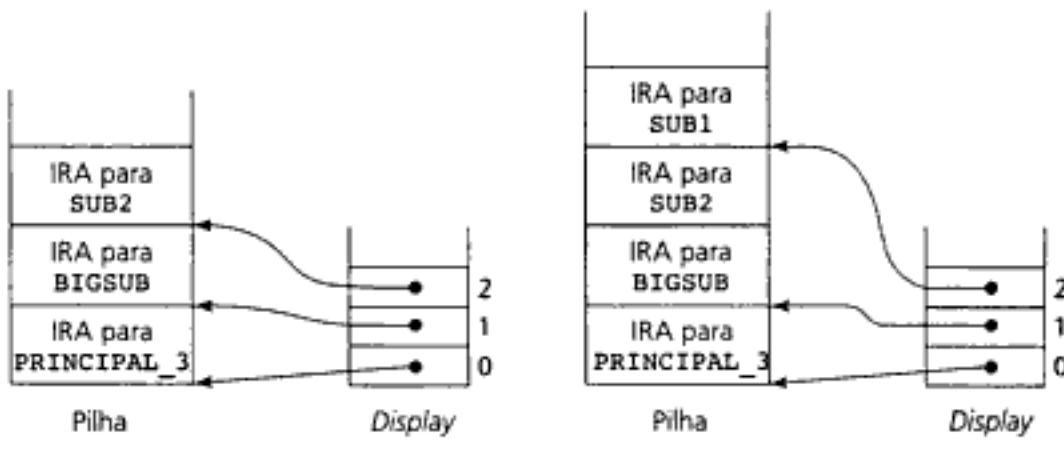
1.  $Qsd = Psd$
2.  $Qsd < Psd$
3.  $Qsd > Psd$

Usamos o programa seguinte, uma versão esquemática do programa de nosso exemplo anterior, **PRINCIPAL\_2**, para examinar os três casos:

```
program PRINCIPAL_3;
procedure BIGSUB;
procedure SUB1;
...
end; { SUB1 }
procedure SUB2;
procedure SUB3;
...
end; { SUB3 }
...
end; { SUB2 }
...
end; { BIGSUB }
end. { PRINCIPAL_3 }
```

O primeiro caso ocorreria se **SUB2** chamasse **SUB1**, porque ambos estão em um nível de profundidade 2. A pilha e o *display* para a situação imediatamente anterior e imediatamente posterior à chamada são mostrados na Figura 10.10.

A chamada, como sempre, exige que a nova instância do registro de ativação de **SUB1** seja adicionada à pilha. O novo ambiente de referenciamento inclui somente **SUB1**, **BIGSUB** e **PRINCIPAL\_3**. Uma vez que os valores de profundidade estática de **SUB1** e **SUB2** são iguais, seus vínculos de *display* devem ocupar a mesma posição nele; ou seja, seus deslocamentos de *display* devem ser iguais. Uma vez que seus vínculos de *display* são sempre salvos na nova instância do registro de ativação, isso não é um problema. Nesse caso, entretanto, a entrada



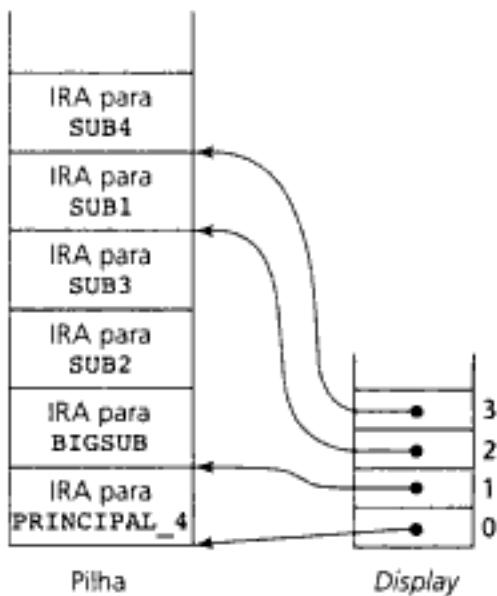
(a) **PRINCIPAL\_3** chama **BIGSUB**; **BIGSUB** chama **SUB2**    (b) **SUB2** chama **SUB1**

IRA = instância do registro de ativação

**FIGURA 10.10** Modificação do *display* para chamadores e chamados com valor de profundidade igual ( $Qsd = Psd$ ).

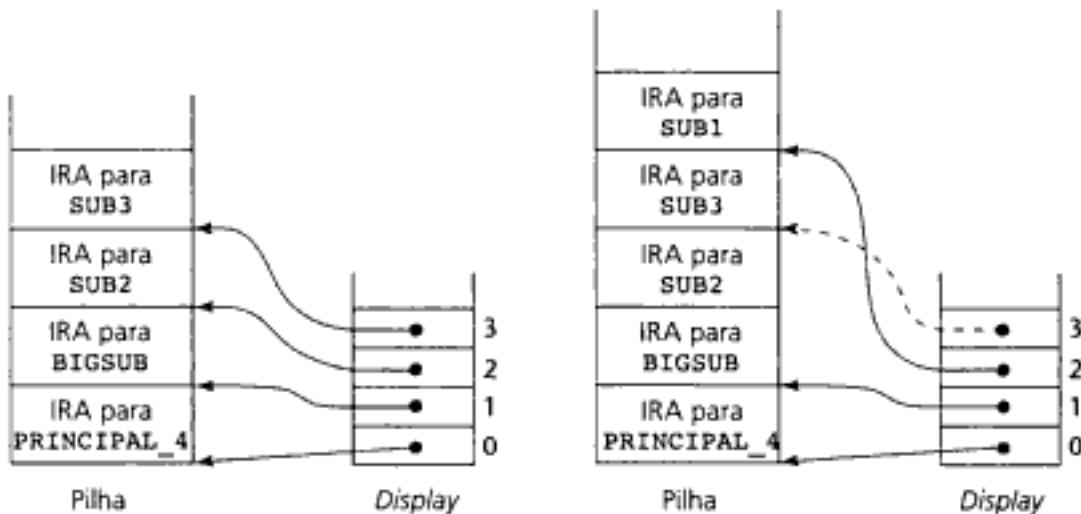
Hidden page

Hidden page



IRA = instância do registro de ativação

**FIGURA 10.13** A pilha e o display depois que `SUB1` chama `SUB4` no programa `PRINCIPAL_4` ( $Qsd < Psd$ ).



IRA = instância do registro de ativação

**FIGURA 10.14** Modificação do display para chamadores com valores de profundidade maiores do que seus chamados ( $Qsd > Psd$ ).

Porém, somente um deve ser removido de fato; o de `SUB2`. O ponteiro para `SUB3` pode permanecer no `display`. As referências a variáveis em `SUB1` não usarão o ponteiro de `display` `SUB3` porque as variáveis `SUB3` não são visíveis a `SUB1`; assim, é seguro deixar o ponteiro no `display`. O compilador não gerará um código que acessará as entradas do `display` acima da entrada para o subprograma ativo atual.

O `display` pode ser armazenado como uma matriz estática na memória. O tamanho máximo do `display`, a profundidade estática máxima de qualquer subprograma no programa, pode ser determinada pelo compilador. Armazenar o `display` na memória funciona razoavelmente bem, contanto que a máquina ofereça endereçamento indireto por meio de localizações da memória. Os acessos a não-locais, então, custam um ciclo de memória a mais do que os locais, que podem usar endereçamento direto. Uma alternativa é colocar os `displays` em registradores, supondo que a máquina tenha um número suficiente sobressalente. Nesse caso, os acessos não exigem o ciclo de memória extra.

Agora, podemos comparar os métodos de encadeamento estático e de `display`. Em uma implementação como a que acabamos de descrever, pode parecer que as referências a variáveis locais seriam mais lentas com um `display` do que com encadeamentos estáticos se este não for armazenado em registradores, porque todas as referências devem passar por entradas de `display`, um processo que acrescenta um nível de indireção. Esta pode ser evitada, entretanto, de modo que o custo para acessar locais seja o mesmo para os dois métodos.

As referências a não-locais somente um nível estático distantes tomam o mesmo tempo para os dois métodos mas, se elas estiverem mais de um nível estático distantes, serão mais rápidas com um `display` porque nenhum encadeamento precisará ser seguido. A tarefa de estimar o tempo necessário para uma referência a não-local é banal quando um `display` é usado — o tempo é igual para todas as variáveis não-locais. Para código crítico quanto ao tempo, essa é uma vantagem dos `displays` sobre os encadeamentos estáticos.

A manutenção em uma chamada a subprograma é mais rápida com encadeamentos estáticos, a menos que o procedimento chamado esteja mais do que alguns níveis estáticos distante. Nesse caso, é necessário um tempo extra para seguir o encadeamento estático do chamador até a instância do registro de ativação do declarante do subprograma chamado. A manutenção de um retorno a subprograma tem um custo fixo em ambos os métodos, com o encadeamento estático sempre sendo ligeiramente mais rápido. O `display` deve ser restaurado quando o subprograma encerrar-se.

Resumindo, os `displays` são uma opção melhor se houver muitas referências a variáveis não-locais distantes. O encadeamento estático é melhor se houver poucas referências a variáveis não-locais distantes, uma situação mais comum. Experiências indicam que os níveis de aninhamento raramente ultrapassam três, na prática.

## 10.4 Blocos

Lembre-se do Capítulo 5 de que diversas linguagens, inclusive a Ada, o C, o C++ e o Java, oferecem escopos locais especificados pelo usuário para variáveis chamados blocos. Como um exemplo de bloco, considere o seguinte segmento de código C:

```
{ int temp;
temp = lista[superior];
lista[superior] = lista[inferior];
```

Hidden page



**FIGURA 10.15** Armazenamento das variáveis de bloco quando blocos não são tratados como procedimentos sem parâmetros.

## 10.5 Implementando o Escopo Dinâmico

Há, pelo menos, duas maneiras distintas pelas quais referências a não-locais em uma linguagem de escopo dinâmico podem ser implementadas: acesso profundo e acesso raso. Note que o acesso profundo e o acesso raso não são conceitos relacionados à vinculação profunda e à rasa. A principal diferença entre vinculação e acesso, aqui, é que as vinculações profunda e rasa resultam em diferentes semânticas; os acessos desta natureza não.

### 10.5.1 Acesso Profundo

Quando um programa em uma linguagem de escopo dinâmico refere-se a uma variável não-local, a referência pode ser resolvida pesquisando-se as declarações em outros subprogramas ativos atualmente, iniciando pelo mais recentemente ativado. Esse conceito é bas-

tante semelhante ao de acessar variáveis não-locais em uma linguagem de escopo estático, exceto que o encadeamento dinâmico — em vez do estático — é seguido. O encadeamento dinâmico vincula todas as instâncias do registro de ativação dos subprogramas na ordem inversa em que eles foram ativados. Portanto, o encadeamento dinâmico é exatamente o necessário para referenciar variáveis não-locais em uma linguagem de escopo dinâmico. Tal método é chamado de **acesso profundo** porque o acesso pode exigir buscas profundas na pilha.

Considere o seguinte exemplo de programa:

```

procedure C;
  integer x, z;
begin
  x := u + v;
  ...
end;
procedure B;
  integer w, x;
begin
  ...
end;
procedure A;
  integer v, w;
begin
  ...
end;
program PRINCIPAL_6;
  integer v, u;
begin
  ...
end;

```

Esse programa foi escrito em uma sintaxe muito assemelhada à do ALGOL, mas não pretende estar em qualquer linguagem particular. Suponhamos que a seguinte seqüência de chamadas a procedimento ocorra:

```

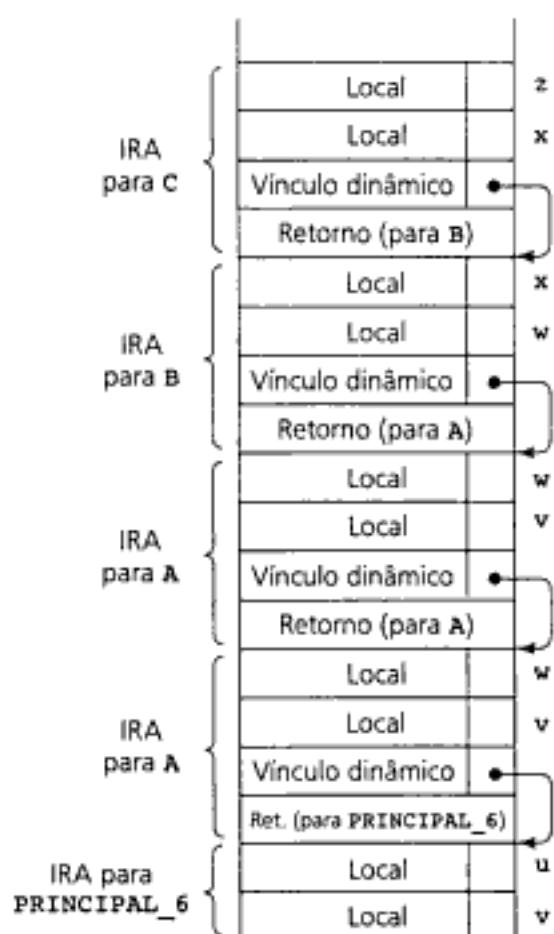
PRINCIPAL_6 chama A
A chama A
A chama B
B chama C

```

A Figura 10.16 mostra a pilha durante a execução do procedimento C depois dessa seqüência de chamada. Note que as instâncias do registro de ativação não têm vínculos estáticos, que não serviriam a nenhum propósito em uma linguagem de escopo dinâmico.

Considere as referências às variáveis x, u e v no procedimento C. A referência a x é encontrada na instância do registro de ativação para C. A referência a u é encontrada pesquisando-se todas as instâncias do registro de ativação na pilha, porque a única variável existente com esse nome está em PRINCIPAL\_6. Isso envolve seguir quatro vínculos dinâmicos e examinar 10 nomes de variáveis. A referência a v é encontrada na instância do registro de ativação mais recente (mais próxima no encadeamento dinâmico) para o procedimento A.

Há duas importantes diferenças entre o método de acesso profundo a não-local em uma linguagem de escopo dinâmico e o método de encadeamento estático para linguagens



IRA = instância do registro de ativação

**FIGURA 10.16** Conteúdo da pilha para um programa de escopo dinâmico.

de escopo estático. No primeiro caso, não existe nenhuma maneira de determinar, em tempo de compilação, o tamanho do encadeamento que deve ser pesquisado. Toda instância do registro de ativação no encadeamento deve ser pesquisada até que a primeira instância da variável seja encontrada. Eis uma razão pela qual as linguagens de escopo dinâmico normalmente são mais lentas do que as de escopo estático. Segundo, os registros de ativação devem armazenar os nomes das variáveis para o processo de busca, ao passo que nas implementações de linguagens de escopo estático, somente os valores são necessários. (Nomes não são exigidos para o escopo estático porque todas as variáveis são representadas pelos pares deslocamento no encadeamento/deslocamento local.)

### 10.5.2 Acesso Raso

O acesso raso é um método de implementação alternativo, não uma semântica alternativa. Conforme afirmamos acima, as semânticas dos acessos profundo e raso são idênticas. No acesso raso, as variáveis declaradas em subprogramas não são armazenadas nos registros de ativação deles. Uma vez que com o escopo dinâmico há, no máximo, uma versão visível de uma variável de qualquer nome específico em determinado momento, uma abordagem

muito diferente pode ser feita. Uma variação do acesso raso é ter uma pilha separada para cada nome de variável em um programa completo. Todas as vezes que uma nova variável com um nome particular é criada por uma declaração no início de uma ativação de subprograma, ela recebe uma célula da pilha para seu nome. Toda referência ao nome é também à variável no topo da pilha associada com aquele, porque a do topo é a mais recentemente criada. Quando um subprograma é finalizado, o tempo de vida de suas variáveis locais encerra-se e as pilhas para esses nomes de variáveis são destacadas. Esse método permite referências muito rápidas a variáveis, mas a manutenção de pilhas nas entradas e saídas de subprogramas é custosa.

A Figura 10.17 mostra as pilhas de variáveis para o exemplo de programa dado na mesma situação mostrada com a pilha da Figura 10.16.

Outra opção para implementar o acesso raso é usar uma tabela central com uma localização para cada diferente nome de variável em um programa. Juntamente com cada entrada é mantido um bit chamado **ativo**, que indica se o nome tem uma vinculação atual ou uma associação a variável. Qualquer acesso a qualquer variável pode ser, então, a um deslocamento estático na tabela central, de modo que o acesso pode ser rápido. As implementações SNOBOL usam a técnica de implementação de tabela central.

A manutenção de uma tabela central é direta. Uma chamada a subprograma exige que todas as suas variáveis locais sejam logicamente colocadas na tabela central. Se a posição da nova variável na tabela central já estiver ativa — ou seja, se, dentro dela, existir uma variável cujo tempo de vida ainda não terminou (o que é indicado pelo bit ativo) — esse valor deverá ser salvo em algum lugar durante o tempo de vida da nova variável. Sempre que uma variável inicia o seu tempo de vida, o bit ativo em sua posição da tabela central deverá ser ajustado.

Tem havido diversas variações no projeto da tabela central e na maneira pela qual os valores são armazenados quando temporariamente substituídos. Uma variação é possuir uma pilha “oculta” em que todos os objetos salvos são armazenados. Uma vez que as chamadas e os retornos a subprogramas e, portanto, os tempos de vida das variáveis locais, são aninhados, isso funciona bem.

A segunda variação talvez seja a mais limpa e a menos dispendiosa de ser implementada. Uma tabela central de células únicas é usada, armazenando somente a versão atual de cada nome de variável. As variáveis substituídas são armazenadas no registro de ativação do subprograma que criou a variável substituta. Esse é um mecanismo de pilha, mas que já existe, de modo que a nova sobretaxa é mínima.

A escolha entre acesso raso e profundo a variáveis não-locais depende das freqüências relativas das chamadas a subprograma e a referências não-locais. O método de acesso

		A			
		A	C		B
PRINCIPAL_6	PRINCIPAL_6		B	C	A
u	v	x	z	w	

(Os nomes nas células da pilha indicam as unidades do programa da declaração de variável)

**FIGURA 10.17** Um método de uso do acesso raso para implementar escopo dinâmico.

profundo proporciona ligação rápida de subprogramas, mas as referências a não-locais, especialmente às distantes (em termos do encadeamento de chamadas), são custosas. O método de acesso raso proporciona referências muito mais rápidas a não-locais, especialmente às distantes, mas é mais custoso em termos de ligação de subprogramas.

## 10.6 Implementando Parâmetros que São Nomes de Subprograma

Os parâmetros que são nomes de subprograma foram discutidos extensamente no Capítulo 9. Lembre-se de que as linguagens de escopo estático usam um método chamado vinculação profunda para associar um ambiente de referenciamento com a ativação de um subprograma passado como parâmetro. Investigaremos, agora, como a vinculação profunda pode ser implementada usando-se os métodos de encadeamento estático e de *display*.

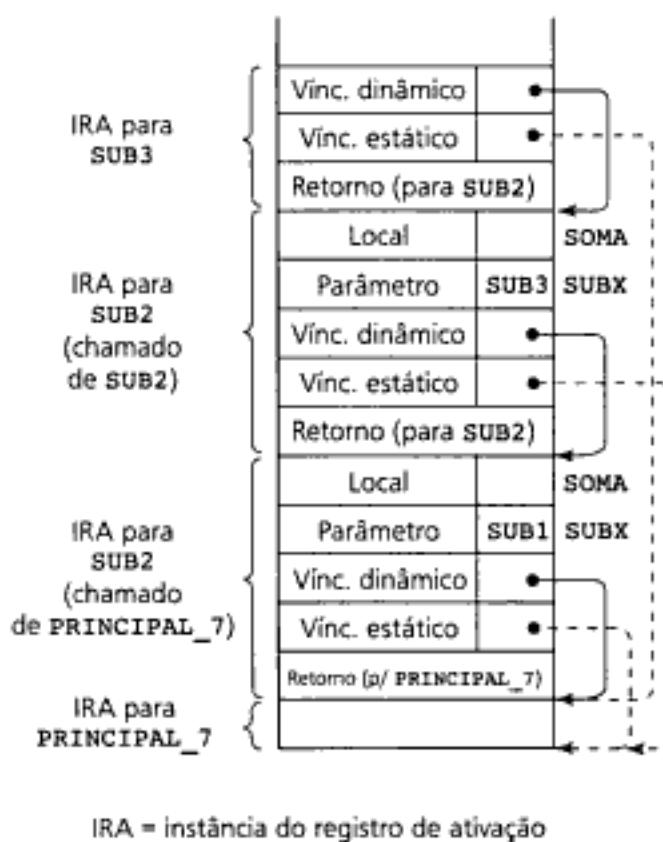
### 10.6.1 Encadeamento Estático

Suponhamos que seja usado o encadeamento estático na implementação. Um subprograma que passa um nome de subprograma como parâmetro deve ter em seu ancestral estático a unidade na qual aquele foi declarado; se não, o nome a ser passado não será visível, e o compilador detectará isso como um erro de sintaxe. Assim, para chamadas sintaticamente corretas, o compilador pode simplesmente passar o vínculo ao pai-estático do subprograma a ser passado, juntamente com o nome deste. A instância do registro de ativação do subprograma passado é, então, inicializada com esse vínculo em seu campo de vínculo estático, em vez de um derivado da maneira usual. A finalização do subprograma passado não exige nenhuma ação diferente das necessárias para quaisquer outras ativações de subprogramas.

### 10.6.2 Displays

Agora, suponhamos que seja usado um *display*, em vez de um encadeamento estático. Lembre-se de que afirmamos especificamente que o processo de manutenção de *displays* descrito na Seção 10.3.4.2 era correto somente quando não havia nenhum parâmetro de nome de subprograma e nenhum parâmetro passado por nome. A manutenção de *displays* para chamadas a subprogramas, sem eles, exige somente a substituição de um único ponteiro de *display*. Quando a chamada é a um subprograma passado como um parâmetro, ponteiros para todos os ancestrais estáticos devem ser colocados no *display*, exigindo que esse número de ponteiros antigos de *display* seja salvo. Uma vez que o ambiente estático de uma ativação de subprograma passado como parâmetro pode ter pouca relação com o ambiente estático daquele no qual ele é chamado, em muitos casos, diversos ponteiros de *display* devem ser substituídos. Em algumas implementações, o *display* inteiro existente é salvo para cada chamada a um subprograma passado como um parâmetro, muitas vezes, na instância do registro de ativação do subprograma em execução. Quando o subprograma que foi baseado é finalizado, o *display* salvo completo substitui o usado para a execução daquele.

Hidden page



**FIGURA 10.18** Conteúdo da pilha para o exemplo de programa PRINCIPAL\_7, com um parâmetro que é um subprograma (SUB1 foi chamado, mas concluiu sua execução).

## RESUMO

A semântica da ligação de subprograma exige muitas ações na implementação. No caso do FORTRAN 77, elas são relativamente simples pelas seguintes razões: a falta de referências não-locais (a não ser pelo COMMON); o fato das variáveis locais usualmente serem estáticas; e a ausência de recursão. Nas linguagens assemelhadas ao ALGOL, a ligação de subprograma é bem mais complexa. Isso decorre das exigências de acessos a não-locais pelo escopo estático, pelas variáveis locais dinâmicas na pilha e pela recursão.

Os subprogramas em linguagens assemelhadas ao ALGOL têm dois componentes: o código real, que é estático, e o registro de ativação, que é dinâmico na pilha. As instâncias do registro de ativação contêm os parâmetros formais e variáveis locais, entre outras coisas.

Os encadeamentos estáticos e displays são os dois métodos principais de implementação de acessos a variáveis não-locais em linguagens de escopo estático. Em ambos os métodos, caminhos de acesso podem ser estaticamente estabelecidos para variáveis em todos os escopos ancestrais estáticos.

O acesso a variáveis não-locais em uma linguagem de escopo dinâmico pode ser implementado por meio do uso do encadeamento dinâmico ou por algum método de tabela de variáveis central. Os encadeamentos dinâmicos proporcionam acessos lentos, mas chamadas e retornos rápidos. Os métodos de tabela central proporcionam acessos rápidos, mas chamadas e retornos lentos.

Parâmetros que são subprogramas oferecem um serviço útil, mas, às vezes, são difíceis de entender. A opacidade situa-se no ambiente de referenciamento disponível quando um subprograma passado como parâmetro está em execução. Subprogramas passados como parâmetros podem ser implementados com encadeamentos estáticos ou com *displays*.

## **NOTAS BIBLIOGRÁFICAS**

A implementação tanto do escopo estático como do dinâmico é abordada em Pratt e Zelkowitz (2001) e Ghezzi e Jazayeri (1997), mas discutida mais a fundo em livros sobre projeto de compiladores, como, por exemplo, Fischer e LeBlanc (1991).

## **QUESTÕES DE REVISÃO**

1. Quais são as quatro razões pelas quais implementar subprogramas em linguagens assemelhadas ao ALGOL é mais difícil do que implementar subprogramas em uma linguagem como o FORTRAN 77?
2. Qual é a diferença entre um registro de ativação e uma instância do registro de ativação?
3. Por que o endereço de retorno, o vínculo estático, o vínculo dinâmico e os parâmetros são colocados na parte inferior do registro de ativação?
4. Quais são os dois passos para localizar uma variável não-local em uma linguagem de escopo estático, independentemente de qual método é usado?
5. Defina encadeamento estático, profundidade estática, profundidade de aninhamento e deslocamento de encadeamento.
6. Explique como uma referência a uma variável não-local é localizada quando encadeamentos estáticos são usados.
7. Quais são os dois problemas potenciais com o método de encadeamento estático?
8. O que é um *display*?
9. Explique como uma referência a uma variável não-local é localizada quando um *display* é usado.
10. Como as referências a uma variável são representadas no método de encadeamento estático? Como elas são representadas no método de *display*?
11. Quais mudanças no *display* são necessárias quando um subprograma é chamado (suponhamos que não haja nenhum parâmetro passado por nome e nenhum parâmetro que seja subprograma)?
12. Compare a eficiência dos métodos de encadeamento estático e *display* para acessos locais, para acessos não-locais, para retornos e para globais.
13. Explique os dois métodos para implementar blocos.
14. Descreva o método de acesso profundo para implementar o escopo dinâmico.
15. Descreva o método de acesso raso para implementar o escopo dinâmico.
16. Quais são as duas diferenças entre o método de acesso profundo para acesso não-local em linguagens de escopo dinâmico e o de encadeamento estático para linguagens de escopo estático?
17. Compare a eficiência do método de acesso profundo com o do método de acesso raso, tanto em termos de chamadas como de acessos a não-locais.
18. Descreva um método de implementar parâmetros que são subprogramas quando é usada a técnica de implementação de encadeamento estático.
19. Descreva um método de implementar parâmetros que são subprogramas quando é usada a técnica de implementação de *display*.
20. Em uma linguagem que permite parâmetros que são nomes de subprograma, a instância do registro de ativação correta de um pai estático é sempre a instância mais próxima do pai que está atualmente na pilha?

**PROBLEMAS**

1. Escreva um algoritmo para executar a manutenção do *display* exigida depois da entrada em um subprograma em uma linguagem que use escopo estático e que também permita nomes de subprograma como parâmetros.
2. Escreva um algoritmo para executar a manutenção do *display* exigida depois da saída de um subprograma em uma linguagem que use escopo estático e que também permita nomes de subprograma como parâmetros.
3. Mostre a pilha com todas as instâncias do registro de ativação, incluindo encadeamentos estáticos e dinâmicos, quando a execução atingir a posição 1 no programa esquemático. Suponha que BIGSUB está no nível 1.

```

procedure BIGSUB;
  procedure A;
    procedure B;
      begin { B }
      ...
      end; { B }
    procedure C;
      begin { C }
      ...
      end; { C }
    begin { A }
    ...
    C;
    ...
    end; { A }
  begin { BIGSUB }
  ...
  A;
  ...
end; { BIGSUB }

```

4. Para o programa esquemático no Problema 3, mostre o *display* que estaria ativo na posição 1, juntamente com as instâncias do registro de ativação na pilha.
5. Mostre a pilha com todas as instâncias do registro de ativação, incluindo encadeamentos estáticos e dinâmicos, quando a execução atingir a posição 1 no programa esquemático seguinte. Suponha que BIGSUB esteja no nível 1.

```

procedure BIGSUB;
  procedure C; forward;
  procedure A(flag: boolean);
    procedure B;
    ...
    A(false)
    end; { B }
  begin { A }
  if flag
    then B
    else C
  ...
end; { A }
procedure C;
  procedure D;

```

Hidden page

## Capítulo 11

# Tipos de Dados Abstratos



### Ole-Johan Dahl

Ole-Johan Dahl e Kristen Nygaard, ambos então do Norwegian Computing Center, estavam primeiramente interessados na simulação por computador. Impulsionados por suas necessidades de simulação, eles projetaram e implementaram as linguagens de simulação SIMULA, em 1964, e a SIMULA 67, em 1967.

- 11.1** O Conceito de Abstração
- 11.2** Encapsulamento
- 11.3** Introdução à Abstração de Dados
- 11.4** Questões de Projeto
- 11.5** Exemplos de Linguagens
- 11.6** Tipos de Dados Abstratos Parametrizados

Neste capítulo, exploraremos o suporte de linguagens de programação para abstração de dados, desde suas origens na SIMULA 67 até sua forma nas linguagens contemporâneas. Entre as novas idéias em termos de metodologias de programação e de projeto de linguagens nos últimos 35 anos, a abstração de dados é uma das mais profundas.

Iniciamos discutindo o conceito geral de abstração em programação e em linguagens de programação. Em seguida, apresentamos e discutimos o encapsulamento nas linguagens de programação. A abstração de dados será, depois, definida e ilustrada com um exemplo. Isso será seguido de uma breve descrição do suporte parcial para abstração de dados na SIMULA 67. Um amplo suporte lingüístico para abstração de dados será discutido em termos de duas linguagens específicas: a Ada e o C++. Uma implementação do mesmo exemplo de abstração de dados será mostrada em cada uma dessas linguagens. Isso iluminará as similaridades e as diferenças no projeto das facilidades de linguagem que suportam a abstração de dados. O suporte do Java para abstração de dados será brevemente discutido como uma alternativa ao C++. Por fim, as capacidades da Ada e do C++ para construir tipos de dados abstratos parametrizados serão discutidas.

Note que, neste livro, os termos *abstração de dados* e *tipos de dados abstratos* referem-se ao mesmo conceito.

## 11.1 O Conceito de Abstração

Uma abstração é uma visualização ou uma representação de uma entidade que inclui somente os atributos de importância em um contexto particular. Ela permite que se colete instâncias de entidades em grupos cujos atributos comuns das mesmas não precisam ser considerados. Estes atributos comuns são abstraídos. Dentro dos grupos, somente os atributos que distinguem os elementos individuais precisam ser considerados. Isso resulta em uma significativa simplificação dos elementos do grupo. Visualizações menos abstratas dessas entidades devem ser consideradas quando é necessário ver um nível mais elevado de detalhe. A abstração é uma arma contra a complexidade da programação; seu propósito é simplificar o processo de programação. Ela é uma arma eficiente porque permite que os programadores concentrem-se nos atributos essenciais e ignorem os atributos subordinados.

Os dois tipos fundamentais de abstração nas linguagens de programação contemporâneas são a de processo e a de dados.

O conceito de **abstração de processo** está entre os mais antigos no projeto de linguagens de programação. Até mesmo a Plankalkül suportou-a. Todos os subprogramas são abstrações de processo porque oferecem uma maneira do programa especificar que algum processo deve ser feito, sem oferecer os detalhes de como será feito (pelo menos, no programa que faz a chamada). Por exemplo, quando um programa precisa classificar um vetor de objetos de dados numéricos de algum tipo, normalmente usa um subprograma para o processo de classificação. No ponto em que o processo de classificação é necessário, uma instrução como

```
ordena_int(lista, comp_lista)
```

é colocada no programa. Essa chamada é uma abstração do processo de classificação real, cujo algoritmo não é especificado. A chamada é independente do algoritmo implementado no subprograma chamado.

No caso do subprograma `ordena_int`, os únicos atributos essenciais são o nome do vetor a ser classificado, o tipo de seus elementos, seu tamanho e o fato de que a chamada a `ordena_int` resultará nele ser classificado. O algoritmo particular que `ordena_int` implementa é um atributo não essencial para o usuário.

A abstração de processo é crucial para a programação. A capacidade de abstrair muitos dos detalhes dos algoritmos em subprogramas torna possível construir, ler e entender programas grandes. Lembre-se de que para ser considerado um programa grande agora, ele deve ter pelo menos várias centenas de milhares de linhas de código.

Todos os subprogramas, inclusive concorrentes (discutidos no Capítulo 13) e os manipuladores de exceções (discutidos no Capítulo 14), são abstrações de processo.

A evolução da abstração de dados seguiu necessariamente à da abstração de processo, porque uma parte integrante e central de toda abstração de dados são suas operações, definidas como abstrações de processo.

## 11.2 Encapsulamento

Preliminarmente à apresentação dos tipos de dados abstratos, devemos discutir o encapsulamento, um precursor e um mecanismo de suporte para estes.

Quando o tamanho de um programa estende-se para além de alguns milhares de linhas, dois problemas práticos aparecem. Do ponto de vista do programador, fazer com que esse programa apareça como uma única coleção de subprogramas não impõe um nível de organização adequado ao programa para mantê-lo intelectualmente administrável. Uma solução é organizá-lo em recipientes (*containers*) sintáticos que incluem grupos de subprogramas e de dados logicamente relacionados. Esses recipientes sintáticos, muitas vezes, são chamados **módulos**, e o processo de projetá-los é chamado **modularização**. O segundo problema prático em relação a programas maiores é a recompilação. No caso de um pequeno, recompilá-lo em sua extensão depois de cada modificação não é custoso. Mas quando os programas vão além de alguns milhares de linhas, o custo da recompilação deixa de ser insignificante. Assim, há uma necessidade evidente de encontrar maneiras de evitar a recompilação das partes de um programa não-afetadas por uma mudança. Isso pode ser obtido organizando-se os programas em coleções de subprogramas e de dados, cada uma das quais pode ser compilada sem a recompilação do restante do programa. Essa coleção é chamada de **unidade de compilação**.

O encapsulamento é um agrupamento de subprogramas e dos dados que eles manipulam. O encapsulamento, separada ou independentemente compilável, constitui um sistema abstruído e uma organização lógica para uma coleção de computações relacionadas. Portanto, o encapsulamento resolve ambos os problemas práticos descritos acima.

Os encapsulamentos, muitas vezes, são colocados em bibliotecas e postos à disposição para serem reutilizados em programas que não para os quais eles foram escritos.

Pessoas têm escrito programas com mais de alguns milhares de linhas ao longo dos últimos 40 anos, de modo que as técnicas para oferecer encapsulamento vêm sendo desenvolvidas há algum tempo.

Em muitas das linguagens assemelhadas ao ALGOL, os programas podem ser organizados aninhando-se definições de subprogramas dentro de subprogramas logicamente maiores que os usam. Conforme discutimos no Capítulo 5, tal método de organizar programas, que usa escopo estático, está longe de ser o ideal. Além disso, em algumas linguagens os

subprogramas não são unidades de compilação. Portanto, eles não criam boas construções de encapsulamento.

No C, uma coleção de funções e de definições de dados relacionadas pode ser colocada em um arquivo, que pode ser, então, compilado independentemente. Não obstante os sistemas de compilação C agora verifiquem a correção das interfaces de funções apropriadamente definidas, eles ainda não verificam o tipo de definições de dados de diferentes arquivos (por "apropriadamente definidas", queremos dizer que não são usados cabeçalhos de função anteriores ao ANSI C). Assim, os arquivos C também não criam encapsulamentos seguros.

Muitas linguagens contemporâneas, inclusive o FORTRAN 90 e a Ada, oferecem a capacidade de reunir coleções de subprogramas, de tipos e de dados em unidades que podem ser compiladas separadamente, significando que suas informações de interface são salvas pelo compilador e usadas para verificação de tipo de interface quando usadas por outra unidade. Essas linguagens também incluem mecanismos de controle de acesso para as entidades nessas unidades. Isso permite que a unidade tenha alguns nomes de tipo visíveis a unidades externas, enquanto tem a representação desses tipos visível somente a outras entidades da unidade. Estas unidades fazem encapsulamentos perfeitos. Elas não somente suportam uma organização de programa concisa e lógica, mas também tornam-na evidente para os leitores do programa.

As características das facilidades de encapsulamento na SIMULA 67, na Ada e no C++ serão discutidas em conjunto com os tipos de dados abstratos na Seção 11.5.

## 11.3 Introdução à Abstração de Dados

Um tipo de dado abstrato, colocando de maneira simples, é um encapsulamento que inclui somente a representação de dados de um tipo específico de dado e os subprogramas que fornecem as operações para esse tipo. Por meio do controle de acesso, detalhes desnecessários do tipo podem ser ocultos das unidades fora do encapsulamento que o usam. As unidades de programa que usam dados abstratos podem declarar variáveis desse tipo, não obstante a representação real estar oculta delas. Uma instância de um tipo de dado abstrato é chamada de **objeto**.

Uma das motivações para a abstração de dados é similar à da abstração de processo. Ela é uma arma contra a complexidade, um meio de tornar programas grandes e/ou complicados mais manejáveis. Outras motivações e vantagens dos tipos de dados abstratos serão discutidas posteriormente nesta seção. Da mesma forma que a presença da abstração de processo permite uma metodologia de projeto de programa diferente, a disponibilidade de abstração de dados também a permite.

A programação orientada a objeto, a qual será descrita no Capítulo 12, é um resultado do uso da abstração de dados no desenvolvimento de software, e ela é um de seus mais importantes componentes.

### 11.3.1 A Vírgula-Flutuante como um Tipo de Dado Abstrato

O conceito de tipos de dados abstratos, pelo menos em termos de tipos embutidos, não é um desenvolvimento recente. Todos eles, até mesmo os do FORTRAN I, são tipos de dados

abstratos, não obstante raramente serem chamados assim. Por exemplo, considere um tipo vírgula-flutuante. A maioria das linguagens inclui pelo menos uma implementação desse tipo, o que constitui um meio de criar variáveis para dados reais e também fornece um conjunto de operações aritméticas para manipular objetos do tipo.

Os tipos vírgula-flutuante em linguagens de alto nível empregam um conceito-chave da abstração de dados: ocultação de informação. O formato real do valor de dados em uma célula de memória de vírgula-flutuante é oculto do usuário. As únicas operações disponíveis são aquelas oferecidas pela linguagem. O usuário não tem permissão para criar novas operações sobre os dados do tipo, exceto aquelas que podem ser construídas usando as operações incorporadas. O usuário não pode manipular diretamente as partes da representação real de objetos de vírgula-flutuante porque essa representação está oculta. É esse recurso que permite a portabilidade do programa entre implementações de uma linguagem particular, não obstante elas poderem usar diferentes representações de valores de números reais.

### 11.3.2 Tipos de Dados Abstratos Definidos pelo Usuário

Um tipo de dado abstrato definido pelo usuário apresenta as mesmas características oferecidas pelos tipos vírgula-flutuante: (1) uma definição de tipo que permite que as unidades de programa declarem suas variáveis, mas oculta a sua representação, e (2) um conjunto de operações para manipular objetos do tipo.

Definimos agora formalmente um tipo de dado abstrato no contexto dos tipos definidos pelo usuário. Um **tipo de dado abstrato** é um tipo de dado que satisfaz as duas condições seguintes:

- A representação ou a definição do tipo e as operações sobre objetos do tipo estão contidas em uma única unidade sintática. Além disso, outras unidades de programa podem ter permissão para criar variáveis do tipo definido.
- A representação de objetos do tipo não é visível pelas unidades de programa que usam o tipo, de modo que as únicas operações diretas possíveis sobre esses objetos são aquelas oferecidas na definição do tipo.

As unidades de programa que usam um tipo de dado abstrato específico são chamadas **clientes** desse tipo.

As principais vantagens de empacotar a representação e as operações em uma única unidade sintática são as mesmas que as do encapsulamento. Constitui um método de organizar um programa em unidades lógicas que podem ser compiladas separadamente. Além disso, permite que modificações nas representações ou operações do tipo sejam feitas em uma única área do programa. Há diversas vantagens em ocultar detalhes da representação e, assim, seu código não pode depender dessa representação. Isso resulta em representações que podem ser modificadas a qualquer hora sem exigir mudanças nos clientes. A interface para a abstração representa alguns, mas não todos, os seus atributos.

Outro benefício distinto e importante da ocultação de informação é o aumento da confiabilidade. Os clientes não podem mudar as representações subjacentes de objetos diretamente, seja intencional ou acidentalmente, aumentando assim a integridade desses objetos. Estes podem ser modificados somente pelas operações fornecidas. É difícil exagerar na importância de ocultar os detalhes de representação de um tipo de dado abstrato.

Hidden page

## 11.4 Questões de Projeto

Uma facilidade para definir tipos de dados abstratos em uma linguagem deve oferecer uma unidade sintática que possa encapsular a definição do tipo e as de subprograma das operações de abstração. Deve ser possível tornar o nome do tipo e os cabeçalhos de subprograma visíveis aos clientes da abstração. Isso permite que os clientes declarem variáveis do tipo abstrato e manipulem seus valores. Embora o nome do tipo precise ter visibilidade externa, sua definição deve ser oculta. Muitas vezes, o mesmo é verdadeiro em relação às definições de subprogramas — os cabeçalhos devem estar visíveis, mas os corpos podem estar ocultos.

Poucas operações incorporadas gerais, supondo que haja alguma, devem ser fornecidas para objetos de tipos de dados abstratos, a não ser aquelas com a definição do tipo. Simplesmente, não há muitas operações que se aplicam a uma ampla variedade de tipos de dados abstratos possíveis. Entre essas, estão as operações de atribuição e as comparações de igualdade e de desigualdade. Se a linguagem não permitir que os usuários sobrecarreguem a atribuição, ela deve estar incorporada. As comparações de igualdade e de desigualdade devem ser predefinidas em alguns casos, mas não em outros. Por exemplo, se o tipo for um ponteiro, a igualdade pode significar equivalência de ponteiros, mas o usuário pode querer que ela signifique igualdade das estruturas apontadas pelos ponteiros.

Algumas operações são exigidas pela maioria dos tipos de dados abstratos, mas, uma vez que elas não são universais, devem ser oferecidas pelo projetista do tipo. Entre elas, estão os iteradores, os construtores e os destruidores. Os iteradores foram discutidos no Capítulo 8. Os construtores são usados para inicializar partes de objetos recém-criados. Os destruidores são usados para reaver armazenamento do monte que pode ser usado por partes de objetos de tipo de dados abstrato.

Conforme afirmamos anteriormente, um tipo de dado abstrato encapsula um único tipo de dados e suas operações. Muitas linguagens contemporâneas, inclusive a Smalltalk (Goldberg e Robson, 1983), o C++ e o Java os suportam diretamente. A alternativa para isso é oferecer uma construção mais generalizada que possa definir qualquer número de entidades, qualquer uma das quais pode ser seletivamente especificada para ser visível fora da unidade que a contém. Essa é a abordagem da Ada. Chamamos essas construções de encapsulamentos que não são tipos de dados abstratos, mas, ao contrário, são generalizações deles. Como tal, elas podem ser usadas para defini-los.

As principais questões de projeto, além do encapsulamento, incluem o seguinte: primeiramente, se as espécies de tipos que podem ser abstratos devem ser restritas. Restringi-las pode ter algumas vantagens. Em especial, se somente ponteiros puderem ser abstratos, uma grande quantidade de recompilação poderá ser evitada no processo de desenvolvimento de software. Por outro lado, alguns consideram isso uma grave restrição por causa de suas desvantagens. Outra questão de projeto é se os tipos de dados abstratos podem ser parametrizados. Por exemplo, se a linguagem suportá-los parametrizados, será possível projetar um tipo de dado abstrato para filas que poderiam armazenar elementos de qualquer tipo escalar. Os tipos de dados abstratos parametrizados serão discutidos na Seção 11.6. Finalmente, há a questão de quais controles de acesso são oferecidos e como esses controles são especificados.

## 11.5 Exemplos de Linguagens

Nesta seção, descreveremos o suporte para abstração de dados oferecidos pela SIMULA 67, pela Ada, pelo C++ e pelo Java.

### 11.5.1 Classes SIMULA 67

As primeiras facilidades de linguagem para suporte direto de abstração de dados, ainda que incompleta segundo nossa definição, apareceu na construção de classes da SIMULA 67.

#### 11.5.1.1 Encapsulamento

Uma definição de classe SIMULA 67 é uma descrição de um tipo. Instâncias ou objetos de uma classe são criados dinamicamente no monte mediante solicitação do programa usuário e podem ser referenciados somente com variáveis ponteiro. Os objetos de classe são, portanto, dinâmicos no monte.

A forma sintática geral de uma definição de classe SIMULA é

```
class nome_da_classe;
begin
    -- declarações de variáveis da classe --
    -- definições de subprogramas da classe --
    -- seção de código da classe --
end nome_da_classe;
```

A seção de código de uma definição de classe é executada somente uma vez, no momento de criação do objeto. Ela serve como construtora da classe e, como tal, é usada para inicialização das variáveis definidas na classe.

A contribuição da SIMULA 67 para a abstração de dados é a capacidade de encapsulamento da construção de classes. Curiosamente, a importância desse aspecto das classes não foi reconhecida até vários anos depois do projeto da SIMULA 67 ter sido concluído. A importância da abstração de dados não foi geralmente entendida até o início da década de 70.

#### 11.5.1.2 Ocultação de Informação

As variáveis declaradas em uma classe SIMULA 67 não são ocultadas dos clientes que criam objetos dela. Tais variáveis podem ser acessadas pelas operações fornecidas pelo subprograma da classe ou diretamente por meio de seus nomes. Isso viola a exigência de ocultação de informação da definição de um tipo de dado abstrato, porque múltiplos caminhos de acesso às entidades da classe são possíveis. O impacto disso é que uma classe SIMULA 67 é bem menos confiável do que um tipo de dado abstrato verdadeiro. Além disso, uma vez que os clientes da classe podem ser projetados para depender das definições de variáveis da classe, modificações nas suas definições podem exigir modificações nos clientes. Isso dificulta a manutenção desses programas.

### 11.5.1.3 Avaliação

A construção de classes SIMULA 67 oferece encapsulamento, mas não ocultação de informação, que permite que os detalhes de representação sejam ocultados dos clientes da classe.

A SIMULA 67 foi revolucionária em seu desenvolvimento da construção de classes. Porém, uma vez que ela jamais desfrutou de um uso generalizado, nós a incluímos aqui somente em função de seu interesse histórico. Agora, voltamos nossa atenção para as duas linguagens contemporâneas que oferecem um suporte completo para abstração de dados: a Ada e o C++.

## 11.5.2 Tipos de Dados Abstratos na Ada

A Ada oferece facilidades de encapsulamento que podem ser usadas para simular tipos de dados abstratos, incluindo a capacidade de ocultar suas representações.

### 11.5.2.1 Encapsulamento

As construções de encapsulamento em Ada são chamadas de **pacotes**, que podem ter duas partes, cada uma das quais também chamada de pacote. São chamadas de **pacote de especificação**, que especificam a interface do encapsulamento, e de **pacote de corpo**, que fornecem a implementação das entidades nomeadas na especificação. Nem todos os pacotes têm uma parte de corpo (pacotes que encapsulam somente dados e constantes não têm ou não precisam de corpos).

Um pacote de especificação e seu pacote de corpo associado compartilham do mesmo nome. A palavra reservada **body** no seu cabeçalho identifica-o como sendo um pacote de corpo. Os pacotes de especificação e de corpo podem ser compilados separadamente, desde que o primeiro seja compilado antes.

### 11.5.2.2 Ocultação de Informação

Não há nenhuma restrição quanto às espécies de tipos que podem ser definidos e exportados de um pacote de especificação. O usuário pode optar por tornar uma entidade inteiramente visível aos clientes ou oferecer somente as informações de interface. Isso é feito oferecendo-se duas seções do pacote de especificação — uma na qual as entidades são visíveis aos clientes e uma que oculta seu conteúdo. Por exemplo, se um tipo precisar ser exportado, mas tiver sua representação oculta, uma declaração abreviada aparecerá na parte visível da especificação, fornecendo somente o nome do tipo e o fato de que sua representação está oculta. Esta aparece em uma parte da especificação chamada **privada**, introduzida pela palavra reservada **private**. A cláusula **private** está sempre no final da especificação.

Suponhamos que um tipo chamado **TIPO\_VERTICE** deva ser exportado por um pacote, mas sua representação deva estar oculta. **TIPO\_VERTICE** é declarado na parte visível do pacote de especificação sem seus detalhes de representação, como em

```
type TIPO_VERTICE is private;
```

Na cláusula **private**, a declaração de **TIPO\_VERTICE** é repetida, mas, dessa vez, com a definição de tipo completa, como em

```

package TIPO_LISTA_ENCADEADA is
    type TIPO_VERTICE is private;
    ...
private
    type TIPO_VERTICE;
    type PTR is access TIPO_VERTICE;
    type TIPO_VERTICE is
        record
            INFO : INTEGER;
            ELO : PTR;
        end record;
end TIPO_LISTA_ENCADEADA;

```

Se nenhuma das entidades de um pacote precisar ser oculta, não haverá nenhum propósito ou necessidade da parte privada da especificação.

A razão pela qual a representação de um tipo aparece no pacote de especificação não tem absolutamente nada a ver com as questões de compilação. Um cliente somente pode ver o pacote de especificação (não o de corpo), mas o compilador deve ser capaz de alocar objetos do tipo exportado quando compilar o cliente. Além disso, o cliente é compilável somente quando o pacote de especificação para o tipo de dado abstrato estiver presente. Por conseguinte, o compilador deve ser capaz de determinar o tamanho de um objeto a partir do pacote de especificação. Assim, a representação do tipo deve estar visível ao compilador, mas não ao código do cliente. Essa é exatamente a situação especificada pela cláusula **private** em um pacote de especificação.

Os tipos declarados como privados são chamados de **tipos privados**. Eles têm operações incorporadas para atribuição e para comparações de igualdade e de desigualdade. Qualquer outra operação deve ser declarada no pacote de especificação que definiu o tipo.

Uma alternativa aos tipos privados é uma forma mais restrita: os **tipos privados limitados**. Eles são descritos na seção privada de um pacote de especificação, como acontece com os privados. A única diferença sintática é que os limitados são declarados como **limited private** na parte visível da especificação do pacote. Objetos de um tipo declarado como privado limitado não têm nenhuma operação incorporada. Isso é útil quando as operações predefinidas usuais de atribuição e de comparação não são significativas ou úteis. Por exemplo, a atribuição e a comparação raramente são usadas para pilhas. Se operações de atribuição ou comparações de igualdade forem necessárias, mas as versões incorporadas não são úteis, essas operações deverão ser fornecidas pelo pacote de especificação. A operação de atribuição deve estar na forma de um procedimento normal, ao passo que os operadores de igualdade e de desigualdade podem ser fornecidos sobrepondo-se esses operadores para o novo tipo.

### 11.5.2.3 Um Exemplo

O exemplo seguinte é o pacote de especificação de um tipo de dado abstrato de pilha:

```

package PACOTEPILHA is
    -- As entidades visíveis, ou interface pública
    type TIPOPILHA is limited private;
    TAM_MAX : constant := 100;
    function VAZIA(STK : in TIPOPILHA) return BOOLEAN;
    procedure EMPILHA(STK : in out TIPOPILHA;
                      ELEMENTO : in INTEGER);

```

Hidden page

A primeira linha do código desse pacote de corpo contém duas instruções: uma **with** e uma **use**. A instrução **with** importa pacotes externos, neste caso, **TEXT\_IO**, que oferece funções para entrada e saída de texto. A instrução **use** elimina a necessidade de qualificação explícita das referências a entidades do pacote nomeado. Isso permite o uso do procedimento **PUT\_LINE** de **TEXT\_IO** sem qualificação explícita (o código teria de usar **TEXT\_IO.PUT\_LINE()** se a instrução **use** não fosse incluída).

O pacote de corpo deve ter as definições de subprograma com cabeçalhos que coincidam com os cabeçalhos de subprograma no pacote de especificação associado. Este último promete que esses subprogramas serão definidos no pacote de corpo associado.

O procedimento seguinte, **USA\_PILHAS**, é um cliente do pacote **PACOTEPIHLA**. Ele ilustra como o pacote poderia ser usado:

```

with PACOTEPIHLA, TEXT_IO;
use PACOTEPIHLA, TEXT_IO;
procedure USA_PILHAS is
    TOPONE : INTEGER;
    PILHA : TIPOPILHA; -- Cria o objeto TIPOPILHA
    begin
        EMPILHA(PILHA, 42);
        EMPILHA(PILHA, 17);
        TOPONE := TOPO(PILHA);
        DESEMPILHA(PILHA);
        ...
    end USA_PILHAS;

```

### 11.5.3 Tipos de Dados Abstratos em C++

O C++ foi criado pela adição de recursos ao C. As primeiras adições importantes foram para suportar a programação orientada a objeto. Uma vez que um dos principais componentes da programação orientada a objeto é o tipo de dados abstratos, o C++ evidentemente deve suportá-lo.

Enquanto a Ada oferece um encapsulamento que pode ser usado para simular tipos de dados abstratos, o C++ oferece a classe, a qual os suporta mais diretamente. As classes C++ são tipos, os pacotes Ada não o são. Os pacotes são importados, permitindo que a unidade importadora declare variáveis de qualquer tipo definido no pacote ou no módulo. Em um programa C++, as variáveis são declaradas como instâncias do tipo classe. Assim, as classes são mais assemelhadas aos tipos incorporados do que os pacotes ou os módulos. Uma unidade de programa que ganha visibilidade a 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 dessa classe, mas somente por meio da instância da classe.

#### 11.5.3.1 Encapsulamento

As classes do C++ baseiam-se nas da SIMULA 67 e são uma extensão dos tipos **struct** do C. Herdada da linguagem SIMULA 67, uma classe C++ é uma descrição de um tipo de dado.

Os dados definidos em uma classe são chamados **membros de dados**; as funções definidas em uma classe são chamadas **funções-membro**. Todas as instâncias de uma classe compartilham de um único conjunto de funções-membro, mas cada instância obtém seu próprio conjunto de membros de dados da classe. Não obstante as instâncias de classe também poderem ser estáticas e dinâmicas no monte, consideramos aqui somente as classes dinâmicas na pilha. Suas instâncias são sempre criadas pela elaboração de uma declaração de objeto. Além disso, o tempo de vida dessa instância de classe encerra-se quando o final do escopo de sua declaração é atingido. As classes podem ter membros de dados dinâmicos no monte, de modo que, não obstante uma instância de classe ser dinâmica na pilha, ela pode incluir membros de dados dinâmicos no monte e alocados no monte. O C++ oferece os operadores **new** e **delete** para gerenciar o monte.

Uma função-membro de uma classe pode ser definida de duas maneiras distintas: a definição completa pode aparecer na classe ou somente seu cabeçalho. Quando tanto o cabeçalho como o corpo de uma função-membro aparecem na definição da classe, a função-membro é implicitamente *inlined*<sup>11</sup>. Lembre-se que isso significa que seu código é colocado no código do chamador, em vez de exigir o processo de ligação de chamada e retorno usual. Se somente o cabeçalho de uma função-membro aparecer na definição de classe, sua definição completa aparecerá fora daquela e será compilada separadamente. É uma boa idéia ter pequenas funções-membro *inlined*, porque elas não ocuparão muito espaço no cliente, e o tempo de ligação será poupano.

#### 11.5.3.2 Ocultação de Informação

Uma classe C++ pode conter tanto entidades ocultas como visíveis. As que devem ser ocultas são colocadas em uma cláusula **private**, enquanto as visíveis ou públicas são escritas em uma cláusula **public**. Esta última, portanto, descreve a interface com objetos da classe. Há também uma terceira categoria de visibilidade, **protected**, a qual será discutida no contexto da herança no Capítulo 12.

O C++ permite que o usuário inclua funções **constructor** em definições de classe, as quais são usadas para inicializar os membros de dados de objetos recém-criados. Um construtor também pode alocar os membros de dados dinâmicos no monte do novo objeto. Os construtores são implicitamente chamados quando um objeto do tipo classe é criado. Um construtor tem o mesmo nome da classe da qual faz parte, o que é estranho, mas inofensivo. Pode haver mais de um construtor para uma classe; neste caso, evidentemente, eles são sobrecarregados. Logicamente, cada um deve ter um perfil paramétrico único.

Uma classe C++ também pode incluir uma função **destructor**, implicitamente chamada quando se encerra o tempo de vida de uma instância da classe. Todos os objetos dinâmicos no monte existem até que sejam explicitamente desalocados com o operador **delete**. Conforme afirmamos acima, as instâncias de classe dinâmicas na pilha podem conter membros de dados dinâmicos no monte. A função destrutora pode incluir um operador **delete** nos membros dinâmicos no monte para desalocar o espaço deles no monte. Destrutores freqüentemente são usados como um auxílio para depuração, em cujo caso, simplesmente, exibem ou imprimem os valores de alguns ou de todos os membros de dados

<sup>11</sup>N. de T. *Inlined*: de *inline* (o mesmo que "unfold", ou "desdobra"). Substituir uma chamada a função por uma instância do corpo da função.

Hidden page

Hidden page

Além das funções, classes inteiras podem ser definidas como amigas; então, todos os membros privados da classe são visíveis a todos os membros da classe amiga.

#### 11.5.3.5 Uma Linguagem Relacionada: Java

O suporte do Java para tipos de dados abstratos é muito similar ao do C++. Entretanto, há algumas diferenças importantes. Todos os tipos de dados definidos pelo usuário em Java são classes, e todos os objetos são alocados no monte e acessados por meio de variáveis de referência. Outra diferença entre o suporte para tipos de dados abstratos do Java e os do C++ é que os subprogramas (métodos) em Java somente podem ser definidos em classes. Assim, não se pode ter apenas cabeçalhos de função em classes Java.

Em vez de ter cláusulas privadas e públicas em suas definições de classe, no Java, **private** e **public** são modificadores que podem ser anexados às definições de método e de variáveis.

Enquanto o C++ depende das classes como sua única construção de encapsulamento, o Java inclui uma segunda em um nível acima delas: os pacotes. Esses podem conter mais de uma definição de classe, e as classes de um pacote são amigas parciais umas das outras. Parcial, aqui, significa que as entidades definidas em uma classe de um pacote, públicas ou protegidas (veja o Capítulo 12) não têm nenhum especificador de acesso, são visíveis a todas as outras classes do pacote. Diz-se que as entidades sem modificadores de acesso têm **escopo de pacote**, porque são visíveis em todo ele. O Java, portanto, tem menos necessidade de declarações amigas explícitas e não inclui as funções amigas ou as classes amigas do C++. Os pacotes, os quais, muitas vezes, contêm bibliotecas, podem ser definidos em hierarquias. As bibliotecas de classes-padrão do Java são definidas em uma hierarquia de pacotes. O escopo de pacote será discutido adicionalmente no Capítulo 12.

O exemplo seguinte é uma definição de classe Java para nosso exemplo de pilha:

```
import java.io.*;
class Classe_pilha {
    private int [] ref_pilha;
    private int tam_max,
                indice_topo;
    public Classe_pilha() { // Um construtor
        ref_pilha = new int [100];
        tam_max = 99;
        indice_topo = -1;
    }
    public void empilha(int numero) {
        if (indice_topo == tam_max)
            System.out.println("Erro no empilhamento-a pilha esta cheia");
        else ref_pilha[++indice_topo] = numero;
    }
    public void desempilha() {
        if (indice_topo == -1)
            System.out.println("Erro no desempilhamento-a pilha esta vazia");
        else --indice_topo;
    }
}
```

```
    public int topo() {return (ref_pilha[indice_topo]);}
    public boolean vazia() {return (indice_topo == -1);}
}
```

Um exemplo de classe que usa `Classe_pilha` é o seguinte:

```
public class Tst_Pilha {
    public static void main (String[] args) {
        Classe_pilha minhaPilha = new Classe_pilha();
        minhaPilha.empilha(42);
        minhaPilha.empilha(29);
        System.out.println("29 é: " + minhaPilha.topo());
        minhaPilha.desempilha();
        System.out.println("42 é: " + minhaPilha.topo());
        minhaPilha.desempilha();
        minhaPilha.desempilha(); // Produz uma mensagem de erro
    }
}
```

Uma pilha é um exemplo bobo para o Java porque a sua biblioteca inclui uma definição de classe para pilhas. Porém, nossa definição de classe para pilhas permite-nos comparar uma implementação Java com a implementação C++ da Seção 11.5.3. Uma diferença evidente é a falta de um destrutor na versão Java, eliminado pela coleta de lixo implícita do Java. Outra diferença significativa é o uso de uma variável de referência, em vez de um ponteiro, para referir-se a objetos da pilha.

## 11.6 Tipos de Dados Abstratos Parametrizados

Muitas vezes, é conveniente parametrizar tipos de dados abstratos. Por exemplo, devemos ser capazes de projetar um tipo de dado abstrato de pilha que possa armazenar quaisquer elementos de tipo escalar em vez de ser necessário escrever uma abstração de pilha separada para cada escalar diferente. Nas duas seções seguintes, as capacidades da Ada e do C++ de construir tipos de dados abstratos parametrizados serão discutidas.

### 11.6.1 Ada

Os procedimentos genéricos da linguagem Ada foram discutidos no Capítulo 9. Os pacotes também podem ser genéricos, de maneira que também podemos construir tipos de dados abstratos genéricos ou parametrizados.

O exemplo de tipo de dados abstrato de pilha em Ada mostrado na Seção 11.5.2 padece de duas restrições: (1) as pilhas de seu tipo podem armazenar somente elementos de tipo inteiro, e (2) elas podem ter somente até 100 elementos. Ambas as restrições podem ser eliminadas usando-se um pacote genérico, que pode ser instanciado para outros tipos de elemento e para qualquer tamanho desejável (essa é uma instanciação genérica, muito diferente da de uma classe para criar um objeto). O pacote de especificação seguinte descreve a interface de um tipo de dado abstrato de pilha genérico com tais recursos:

```

generic
    TAM_MAX : POSITIVE; -- Um parâmetro genérico para o tamanho
                          da pilha
    type TIPO_ELEMENTO is private; -- Um parâmetro genérico para
                                    -- o tipo de elemento
package PILHA_GENERICA is
    -- As entidades visíveis, ou interface pública
    type TIPOPILHA is limited private;
    function VAZIA(STK : in TIPOPILHA) return BOOLEAN;
    procedure EMPILHA(STK : in out TIPOPILHA;
                      ELEMENTO : in TIPO_ELEMENTO);
    procedure DESEMPILHA(STK : in out TIPOPILHA);
    function TOPO(STK : in TIPOPILHA) return TIPO_ELEMENTO;
    -- A parte oculta
private
    type TIPO_LISTA is array (1.. TAM_MAX) of TIPO_ELEMENTO;
    type TIPOPILHA is
        record
            LISTA : TIPO_LISTA;
            TOPSUB : INTEGER range 0..TAM_MAX :=0;
        end record;
    end PILHA_GENERICA;

```

O pacote de corpo para `PILHA_GENERICA` é o mesmo para `PACOTEPILHA` da seção anterior, exceto que o tipo do parâmetro formal `ELEMENTO` em `EMPILHA` e `TOPO` é `TIPO_ELEMENTO` em vez de `INTEGER`.

A instrução seguinte instancia `PILHA_GENERICA` para uma pilha de 100 elementos do tipo `INTEGER`:

```
package PILHA_INTEIRA is new PILHA_GENERICA(100, INTEGER);
```

Também seria possível criar um tipo de dado abstrato para uma pilha de tamanho 500 para elementos reais, como em

```
package PILHA_REAL is new PILHA_GENERICA(500, FLOAT);
```

Essas instanciações criam duas versões de código-fonte de `PILHA_GENERICA` diferentes durante a compilação.

### 11.6.2 C++

O C++ também suporta tipos de dados abstratos parametrizados ou genéricos. Para tornarmos o exemplo da classe de pilha C++ da Seção 11.5.3 genérico no tamanho de pilha, somente a função construtora precisa ser mudada, como em

```
pilha(int tamanho) {
    ptr_pilha = new int [tamanho];
    tam_max = tamanho - 1;
    top = -1;
}
```

A declaração para o objeto pilha agora pode aparecer como

Hidden page

Hidden page

## ***PROBLEMAS***

---

1. Projete o exemplo de tipo abstrato de pilha em Pascal, supondo que a definição de pilha, as suas operações e o código que a usa estejam todos no mesmo programa.
2. Qual parte ou partes críticas da definição de um tipo de dado abstrato estão faltando em uma implementação Pascal do tipo pilha, como, por exemplo, a do Problema 1?
3. Projete o exemplo de tipo abstrato de pilha em FORTRAN 77, usando um único subprograma com múltiplas entradas para a definição de tipo e as operações.
4. Como a implementação FORTRAN do Problema 3 compara-se com a implementação Ada deste capítulo em termos de confiabilidade e de flexibilidade?
5. Modifique a classe C++ do tipo abstrato de pilha para usar uma representação de lista encadeada e teste-a com o mesmo código que aparece neste capítulo.
6. Alguns engenheiros de software acreditam que todas as entidades importadas devem ser qualificadas pelo nome da unidade de programa exportadora. Você concorda? Sustente sua resposta.
7. Projete um tipo de dado abstrato para uma abstração de matriz em uma linguagem que você conheça, incluindo operações de adição, de subtração e de multiplicação de matrizes.
8. Projete um tipo de dado abstrato de fila em uma linguagem que você conheça, incluindo operações para enfileirar, desenfileirar e testar se a fila está vazia.
9. Suponhamos que alguém tenha projetado um tipo de dado abstrato de pilha no qual a função, topo, retorne um caminho (ou ponteiro) de acesso 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.
10. Escreva um tipo de dado abstrato para números complexos, incluindo operações de adição, de subtração, de multiplicação, de divisão, de extração de cada uma das partes de um número complexo e de construção de um número complexo a partir de duas constantes, variáveis ou expressões reais. Use a Ada, o C++ ou o Java.
11. Escreva um tipo de dado abstrato para filas cujos elementos armazenem nomes de 10 caracteres. Os elementos da fila devem ser alocados dinamicamente no monte. As operações da fila são enfileirar, desenfileirar e testar se a fila está vazia. Use a Ada, o C++ ou o Java.

Hidden page

## Capítulo 12

# Suporte para Programação Orientada a Objeto



**Adele Goldberg**

Adele Goldberg passou 14 anos no Palo Alto Research Center da Xerox, chefiando a equipe de projeto da Smalltalk e sua implementação. Ela foi fundamental não somente no desenvolvimento da Smalltalk, mas também no desenvolvimento do paradigma de interfaces do usuário baseadas em janelas e ícones.

- 12.1** Introdução
- 12.2** Programação Orientada a Objeto
- 12.3** Questões de Projeto Referentes às Linguagens Orientadas a Objeto
- 12.4** Visão Geral da Smalltalk
- 12.5** Introdução à Linguagem Smalltalk
- 12.6** Exemplo de Programas Smalltalk
- 12.7** Recursos em Grande Escala da Smalltalk
- 12.8** Avaliação da Smalltalk
- 12.9** Suporte para Programação Orientada a Objeto em C++
- 12.10** Suporte para Programação Orientada a Objeto em Java
- 12.11** Suporte para Programação Orientada a Objeto em Ada 95
- 12.12** Suporte para Programação Orientada a Objeto em Eiffel
- 12.13** O modelo de Objetos da JavaScript
- 12.14** Implementação de Construções Orientadas a Objeto

Este capítulo inicia-se com uma introdução à programação orientada a objeto. Segue-se uma discussão das principais questões de projeto referentes à herança e à vinculação dinâmica. Depois, apresentamos uma visão geral da Smalltalk, seguida de uma descrição detalhada de um subconjunto dela, a qual é ilustrada com dois programas Smalltalk completos. Seguem-se breves descrições do suporte para programação orientada a objeto em C++, em Java, em Ada 95 e em Eiffel.

## 12.1 Introdução

As linguagens que suportam programação orientada a objeto agora estão firmemente entranhadas em uma posição de destaque. Do COBOL ao LISP, incluindo virtualmente todas as linguagens entre um e outro, foram criados dialetos que suportam programação orientada a objeto. Entre elas estão o C++, a Ada 95 e a CLOS, uma versão orientada a objeto do LISP (Bobrow et al., 1988). O C++ e a Ada 95 suportam programação orientada a procedimentos e a dados, além da programação orientada a objeto. A CLOS também suporta programação funcional. Algumas das linguagens mais recentes projetadas para suportar programação orientada a objeto não suportam outros paradigmas, mas ainda empregam algumas das estruturas básicas e têm a aparência das linguagens imperativas mais antigas. Entre elas estão a Eiffel e o Java. Por fim, há uma linguagem puramente orientada a objeto bastante não-convencional: a Smalltalk. Ela foi a primeira a oferecer suporte completo para programação orientada a objeto. O suporte específico varia amplamente entre as linguagens, e esse é um tópico importante deste capítulo.

Este capítulo recorre fortemente ao Capítulo 11. Ele é, de fato, uma continuação deste último. Isso reflete a realidade de que a programação orientada a objeto é essencialmente uma aplicação do princípio da abstração de tipos de dados. Especificamente, na programação orientada a objeto, os atributos comuns de uma coleção de tipos de dados abstratos similares são fatorados e colocados em um novo tipo. Os membros da coleção herdam as partes comuns deste. Isso é herança, a qual é o centro da programação orientada a objeto e das linguagens que a suportam.

## 12.2 Programação Orientada a Objeto

### 12.2.1 Introdução

O conceito de **programação orientada a objeto** tem suas raízes na SIMULA 67, mas não foi amplamente desenvolvido até que a evolução da Smalltalk resultasse na produção de sua versão 80 (em 1980, é claro). De fato, alguns consideram a Smalltalk a única linguagem puramente orientada a objeto. Uma linguagem com esta característica deve oferecer três recursos chaves: tipos de dados abstratos, herança e um tipo particular de vinculação dinâmica.

A programação orientada por procedimentos, o paradigma de desenvolvimento de software mais popular na década de 70, concentra-se em subprogramas e em bibliotecas de subprogramas. Dados são enviados a subprogramas para computações. Por exemplo, um

vetor de valores inteiros que precisa ser classificado é enviado como um parâmetro a um subprograma que o classifica.

A programação orientada a dados concentra-se em tipos de dados abstratos, discutidos detalhadamente no Capítulo 11. Nesse paradigma, a computação de um objeto-dados é especificada chamando-se subprogramas associados com o objeto-dados. Se um objeto vetor precisar ser classificado, a operação será definida no tipo de dados abstrato para o vetor. O processo de classificação é ativado chamando-se essa operação no objeto vetor específico. O paradigma da programação orientada a dados foi popular na década de 80, e é bem servido pelas facilidades de abstração de dados do Modula-2, da Ada e de diversas linguagens mais recentes. As linguagens que suportam programação orientada a dados freqüentemente são conhecidas como **baseadas em objetos**.

### 12.2.2 Herança

De meados até o final da década de 80, tornou-se patente a muitos desenvolvedores de software que uma das melhores oportunidades para uma maior produtividade em suas profissões era a reutilização de software. Os tipos de dados abstratos, com seu encapsulamento e com seus controles de acesso, evidentemente eram as unidades a serem reutilizadas. O problema com a reutilização dos tipos de dados abstratos é que, em quase todos os casos, os recursos e as capacidades do tipo existente não são muito certos para o novo emprego. O antigo exige, no mínimo, algumas pequenas modificações, que podem ser difíceis, porque exigem que a pessoa que faz a modificação entenda parte (se não todo) do código existente. Além disso, em muitos casos, as modificações exigem mudanças em todos os programas-clientes.

Um segundo problema com a programação orientada a dados é que todas as definições de tipos de dados abstratos são independentes e estão no mesmo nível. Isso, freqüentemente, torna impossível estruturar um programa para ajustar-se ao espaço do problema alvo. Em muitos casos, o problema subjacente tem categorias de objetos relacionadas, tanto como parentes (sendo similares uns aos outros) como pais e filhos (tendo algum tipo de relação subordinada).

A herança oferece uma solução tanto para o problema da modificação apresentado pela reutilização de tipos de dados abstratos como pelo problema de organização do programa. Se um novo tipo de dado abstrato puder herdar os dados e a funcionalidade de algum tipo existente, e se também for permitido modificar algumas dessas entidades e adicionar novas, a reutilização será grandemente facilitada sem exigir mudanças no tipo de dado abstrato reusado. Os programadores podem pegar um tipo de dado abstrato existente e moldá-lo para que se ajuste à nova exigência do problema. Por exemplo, suponhamos que um programa já tenha um tipo de dado abstrato para vetores de inteiros que inclua uma operação de classificação. Depois de algum período de uso, o programa é atualizado e exige um tipo de dados abstrato para vetores de inteiros com a operação de classificação, mas também precisa de uma operação para computar a mediana dos elementos dos objetos-vetor. Uma vez que a estrutura de vetor é oculta em um tipo de dado abstrato, sem herança, ele deve ser modificado para adicionar a nova operação nessa estrutura. Com herança, não há necessidade de modificar o tipo existente; é possível definir uma subclasse dele que retenha as operações de classificação, mas adicione outra para a computação da mediana.

Os tipos de dados abstratos em linguagens orientadas a objeto, seguindo a linha da SIMULA 67, usualmente são chamados de classes. Como acontece com as instâncias dos tipos de dados abstratos, as instâncias de classe são chamadas **objetos**. Uma classe defini-

da pela herança de outra é uma **classe derivada** ou uma **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 **métodos**. As chamadas a métodos freqüentemente têm nome de **mensagens**. A coleção inteira de métodos de um objeto é chamada de **protocolo de mensagem** ou de **interface de mensagem** do objeto. Uma mensagem deve ter, no mínimo, duas partes: o objeto específico ao qual ela está sendo enviada e um nome de um método que defina a ação solicitada naquele. Assim, as computações em um programa orientado a objeto são especificadas pelas mensagens enviadas de objetos a outros objetos.

No caso mais simples, uma classe herda todas as entidades (variáveis e métodos) de sua classe-pai. Isso pode ser complicado pelos controles de acesso nas entidades de uma classe-pai. Por exemplo, conforme vimos nas definições de tipo de dados abstratos no Capítulo 11, algumas das entidades são classificadas como públicas e outras como privadas. Esses controles de acesso permitem que o projetista do programa oculte parte do tipo de dado abstrato dos clientes. Esses mesmos controles normalmente estão presentes nas classes das linguagens orientadas a objeto. As classes derivadas são outro tipo de cliente para o qual o acesso pode ser concedido ou recusado. Para levar isso em conta, algumas linguagens orientadas a objeto incluem uma terceira categoria de controle de acesso, freqüentemente chamada de **protegida (protected)**, usada para fornecer acesso a classes derivadas e recusá-lo a outras classes.

Além de herdar entidades de sua classe-pai, uma classe derivada pode acrescentar novas entidades e modificar métodos herdados. Um método modificado tem o mesmo nome, e freqüentemente o mesmo protocolo do método do qual ele é uma modificação. Diz-se que o novo método **sobrepõe-se (override)** à versão herdada, o que é então chamado de método **sobreposto (overridden)**. O propósito mais comum de um método de sobreposição é fornecer uma operação específica a objetos da classe derivada, mas que não seja apropriada para objetos da classe-pai. Por exemplo, considere uma hierarquia de classes em que a classe-raiz descreve as características arquitetônicas gerais de catedrais góticas francesas. Esta classe-raiz, `French_Gothic`, tem um método para desenhar a fachada de uma catedral gótica francesa genérica. Em seguida, suponhamos que a classe `French_gothic` tenha três derivadas, `Reims`, `Amien` e `Chartres`, cada uma das quais inclui um método para desenhar sua fachada particular. Tais versões de desenho devem sobrepor-se ao método `desenha` herdado da classe-pai.

Classes podem ter dois tipos de métodos e dois tipos de variáveis. Os métodos e as variáveis mais comumente usados são chamados de **instância**. Todo objeto de uma classe tem seu próprio conjunto de variáveis de instância, as quais armazenam o estado do objeto. A única diferença entre dois objetos da mesma classe é o estado de suas variáveis de instância. Os métodos de instância operam somente nos objetos da classe. As **variáveis de classe** pertencem a esta, não ao seu objeto, de maneira que há somente uma cópia da classe. Os **métodos de classe** podem executar operações nela, e possivelmente também nos objetos da mesma. Na maior parte do restante deste capítulo, ignoraremos os métodos de classe e as suas variáveis.

Se uma classe criada por meio de herança tiver uma única classe-pai, o processo irá chamar-se **herança simples**. Se a classe tiver mais de uma classe-pai, o processo irá chamar-se **herança múltipla**. Quando um grande número de classes está relacionado pela herança simples, as relações entre elas podem ser mostradas em uma árvore de derivação. As relações de classes em uma herança múltipla podem ser mostradas em um grafo de derivação.

O projeto de programa para um sistema orientado a objeto inicia-se com a definição de uma hierarquia de classes que descreva as relações dos objetos que preencherão o pro-

grama-solução. Quanto melhor essa hierarquia de classes combinar-se com o espaço de problema, mais natural será a solução completa.

Uma desvantagem da herança como meio de aumentar a possibilidade de reutilização é que ela cria uma dependência entre as classes em uma hierarquia de herança. Isso trabalha contra uma das vantagens dos tipos de dados abstratos: o fato deles serem independentes uns dos outros. Obviamente, nem todos os tipos de dados abstratos devem ser completamente independentes. Mas, em geral, a independência dos tipos de dados abstratos é uma de suas características mais fortes. Porém, pode ser difícil, se não impossível, aumentar a capacidade de reutilização de tipos de dados abstratos sem criar dependências entre alguns deles.

### 12.2.3 Polimorfismo e Vinculação Dinâmica

A terceira característica das linguagens de programação orientadas a objeto é um tipo de polimorfismo proporcionado pela vinculação dinâmica de mensagens a definições de método. Isso é suportado permitindo-se que alguém defina variáveis polimórficas do tipo da classe-pai que também são capazes de referenciar objetos de qualquer uma das subclasses da mesma. A classe-pai pode definir um método sobreposto por suas subclasses. As operações definidas por tais métodos são similares, mas devem ser personalizadas para cada classe da hierarquia. Quando esse método é chamado por meio da variável polimórfica, essa chamada é vinculada dinamicamente ao método da classe apropriada. Um propósito dessa vinculação dinâmica é permitir que os sistemas sejam mais facilmente estendidos tanto durante o desenvolvimento quanto durante a manutenção. Os programas podem ser escritos para executar operações em objetos-classe genéricos. Estas operações são genéricas em termos de que podem aplicar-se a objetos de qualquer classe relacionados por meio de derivação da mesma classe básica. Como um exemplo de vinculação dinâmica, considere o exemplo das catedrais. Se um programa que usa `French_Gothic` tiver uma variável polimórfica, `cathedral`, do tipo da `French_Gothic`, essa variável poderia referenciar objetos de `French_Gothic` e também objetos de qualquer uma das classes derivadas. Agora, quando `cathedral` for usada para chamar `desenha` (definido em `French_Gothic` e em todos os seus descendentes), essa chamada será vinculada dinamicamente à versão correta de `desenha`, escolhida pelo tipo ao qual a variável polimórfica está então referenciando.

A vinculação dinâmica por meio de variáveis polimórficas é um conceito poderoso. Suponhamos que nosso exemplo das catedrais seja escrito em C. Os três exemplos de catedrais góticas francesas poderiam ser armazenados em variáveis de um tipo `struct`. Poderia haver uma única função `desenha` que usasse uma instrução `switch` para chamar a função de desenho correta, baseando-se na catedral específica. Porém, nesse tipo de implementação, cabe ao programador chamar a versão apropriada da função de desenho. A manutenção é muito mais fácil com a implementação orientada a objeto. Por exemplo, suponhamos adicionar uma nova catedral à coleção, digamos, `Paris`. Isso nos obrigaría a modificar a construção `switch` na função de desenho geral, juntamente com quaisquer funções similarmente construídas para catedrais. No caso orientado a objeto, entretanto, adicionar uma nova catedral não tem nenhum efeito no código existente.

Em muitos casos, o projeto de uma hierarquia de heranças resulta em uma ou em mais classes que estão tão altas na hierarquia que uma instanciação delas não faz sentido. Por exemplo, suponhamos que houvesse uma classe `building` como classe-pai ou como ancestral da `French_Gothic`. Provavelmente, não faria sentido ter um método `desenha` implementado em `building`. Entretanto, uma vez que todas as suas classes descendentes devem ter esse método implementado, o protocolo (mas não o corpo) desse método é inclu-

Hidden page

em uma linguagem maior cuja estrutura de tipos é confusa para todos, a não ser para usuários especialistas.

Outra alternativa ao uso exclusivo de objetos é ter uma estrutura de tipos ao estilo imperativo para os tipos escalares primitivos. Isso proporciona a velocidade de operações sobre valores primitivos comparáveis àquelas esperadas no modelo imperativo. Infelizmente, a alternativa também leva a complicações na linguagem. Invariavelmente, valores não-objeto devem ser misturados com objetos. Isso cria a necessidade das chamadas classes envoltório (*wrapper*) para os tipos não-objeto, de modo que algumas operações comumente necessárias podem ser enviadas a objetos com valores de tipo não-objeto. Na Seção 12.6, discutiremos um exemplo disso em Java.

### 12.3.2 As Subclasses são Subtipos?

A questão aqui é relativamente simples: a relação “é-uma” sustenta-se entre uma classe-pai e suas derivadas? Uma relação é-uma garantiria que uma variável do tipo derivada pode aparecer em qualquer lugar em que uma variável do tipo classe-pai fosse legal.

Os subtipos da Ada são exemplos dessa forma simples de herança. Por exemplo,

```
subtype SMALL_INT is INTEGER range -100..100;
```

As variáveis do tipo **SMALL\_INT** têm todas as operações das variáveis **INTEGER**, mas podem armazenar somente um subconjunto dos valores possíveis em **INTEGER**. Além disso, toda variável **SMALL\_INT** pode ser usada em qualquer lugar onde é usada uma **INTEGER**. Ou seja, toda variável **SMALL\_INT** é, em certo sentido, uma variável **INTEGER**.

Uma classe derivada é chamada **subtipo** se tiver uma relação é-uma com sua classe-pai. As características de uma subclasse que lhe asseguram o conceito de subtipo são as seguintes: a subclasse pode adicionar somente variáveis e métodos e se sobrepor a métodos herdados de maneira “compatível”. Este último significa, aqui, que o método que sobrepõe pode substituir o sobreposto sem causar erros de tipo. Ter um número idêntico de parâmetros e idênticos tipos de parâmetro e retorno garantiria, é claro, a compatibilidade. Restrições menos severas são possíveis, dependendo das regras de compatibilidade de tipos da linguagem.

Nossa definição de subtipo desautoriza claramente a existência de entidades na classe-pai que não sejam herdadas pela subclasse.

### 12.3.3 Herança de Implementação e de Interface

O conceito de ocultação de informação em um tipo de dados abstrato fornece a interface de suas facilidades com os clientes, mas oculta sua implementação. E as subclasses? Elas devem ver somente a interface de suas classes-pai ou também devem ter acesso garantido aos detalhes de implementação das facilidades desta? Se somente a interface da classe-pai for visível à subclasse, isso se chama **herança de interface**. Se os detalhes de implementação também forem visíveis, irá chamar-se **herança de implementação**.

Essa questão apresenta ao projetista um compromisso de projeto com vantagens e com desvantagens a serem consideradas. Conceder acesso à subclasse para a parte “oculta” da classe-pai a torna dependente desses detalhes. Qualquer mudança na implementação da classe-pai exigirá recompilação da subclasse e, em muitos casos, a mudança exigirá sua modificação. Isso derruba a vantagem da ocultação de informação para os clientes da subclasse. Por outro lado, manter a parte de implementação da classe-pai oculta das subclasses

pode causar ineficiências na execução das instâncias das mesmas. Isso pode ser causado pela diferença de eficiência entre o acesso direto a estruturas de dados e a exigência de acesso por meio de operações definidas na classe-pai. Por exemplo, considere uma classe que define uma pilha e uma subclasse que deve incluir uma operação para retornar o segundo elemento a partir do topo. Se a linguagem usar herança de implementação, tal operação poderá ser definida para simplesmente retornar o elemento na posição do topo da pilha menos um. Porém, se o projetista da linguagem tiver optado pela herança de interface, esse código ficaria mais ou menos assim:

```
int segundo() {
    int temp = topo();
    desempilha();
    int result_temp = topo();
    empilha(temp);
    return result_temp;
}
```

Esse é, evidentemente, um processo mais lento do que o acesso direto ao segundo elemento do topo da pilha. Porém, se a implementação desta for mudada, o método provavelmente precisará ser modificado.

A melhor solução para o projetista da linguagem é fornecer tanto a opção de herança de implementação como a de interface para o projetista de software e deixar que ele decida, considerando caso a caso, qual versão é a melhor.

#### 12.3.4 Verificação de Tipos e Polimorfismo

Na Seção 12.2, o polimorfismo, no universo orientado a objeto, é definido como o uso de um ponteiro (ou uma referência) polimórfico, para acessar um método cujo nome é sobreposto na hierarquia de classes. Isso define o objeto para o qual um dos dois está apontando. A variável polimórfica é do tipo da classe-pai, que, por sua vez, define pelo menos o protocolo de um método sobreposto pelas derivadas. A variável polimórfica pode referenciar objetos da classe-pai e das derivadas, de modo que a classe do objeto para a qual ela aponta nem sempre pode ser estaticamente determinada. A vinculação de mensagens a métodos enviados por variáveis polimórficas deve ser dinâmica. Aqui, a questão é quando a verificação de tipos dessa vinculação desenvolve-se.

Essa questão é importante, porque alinha-se com a natureza fundamental da linguagem de programação. Se o projeto tiver a tipificação forte como uma de suas metas básicas, como acontece com muitas linguagens de programação contemporâneas, a verificação de tipos deve ser feita estaticamente. Isso impõe algumas restrições sérias na relação entre mensagens e métodos polimórficos.

Há duas espécies de verificação de tipos que devem ser feitas entre uma mensagem e um método em uma linguagem fortemente tipificada: 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 ambos precisarem coincidir exatamente, um método de sobreposição deverá ter o mesmo número, os mesmos tipos de parâmetros e o mesmo tipo de retorno que o método sobreposto. Um relaxamento da regra seria permitir compatibilidade de atribuição entre os parâmetros reais e formais e entre o tipo retornado e o esperado pela mensagem.

A alternativa óbvia à verificação de tipos estática é retardar a verificação dos mesmos até que a variável polimórfica seja usada para chamar um método. Como em outras situações, a verificação dinâmica de tipos é mais dispendiosa e retarda a detecção de erros de tipo.

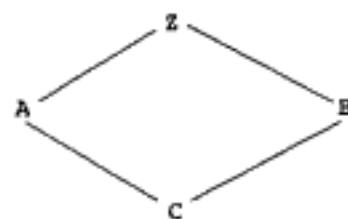
### 12.3.5 Heranças Simples e Múltipla

Outra questão simples é: a linguagem permite herança múltipla (além da simples)? Bem, talvez não seja tão simples. O propósito da primeira é permitir uma nova classe herdar de duas ou mais que descrevem abstrações distintas. Por exemplo, no Java, é comum a escrita de miniaplicativos (*applets*) que incluem animação que é executada, muitas vezes, de maneira concorrente com as outras partes do miniaplicativo. Os miniaplicativos são suportados pela classe `Applet`, e a concorrência é suportada pela classe `Thread`. Seria necessário que um miniaplicativo herdasse de ambas essas classes. Na Seção 12.10.2, discutimos como o Java suporta essa necessidade.

Por que um projetista de linguagem não incluiria a herança múltipla, uma vez que ela é, muitas vezes, útil? As razões situam-se em duas categorias: complexidade e eficiência. A complexidade adicional é ilustrada por diversos problemas. Um evidente é a colisão de nomes. Por exemplo, se uma subclasse chamada `C` herdar tanto da classe `A` como da `B`, e tanto `A` como `B` incluírem uma variável hereditária chamada `soma`, como `C` poderá referir-se às duas `somas` diferentes? Uma variação disso ocorre se tanto `A` como `B` forem derivados de um pai comum, `Z`, em cujo caso é chamado herança **diamante**. Assim, tanto `A` como `B` têm variáveis hereditárias de `Z`; e `C` herda duas versões de cada uma delas (presumindo que elas sejam hereditárias) de `A` e `B`. A herança diamante é mostrada na Figura 12.1.

A questão da eficiência é mais visual do que real. No C++, por exemplo, o suporte à herança múltipla exige apenas uma operação extra para cada método vinculado dinamicamente (Stroustrup, 1994, p. 270). Embora isso seja necessário mesmo que o programa não use herança múltipla, é um pequeno custo adicional.

O uso da herança múltipla pode levar facilmente a organizações complexas de programa. Muitos que tentaram usá-la acharam que projetar as classes para serem usadas como pais múltiplos é difícil. A manutenção de sistemas com herança múltipla pode ser um problema mais sério, porque ela leva a dependências mais complexas entre as classes. Não está claro para alguns que os benefícios da herança múltipla valem o esforço adicional de projetar e de manter um sistema que a use.



**FIGURA 12.1** Um exemplo de herança diamante.

### 12.3.6 Alocação e Desalocação de Objetos

Há duas questões de projeto referentes à alocação e à desalocação de objetos. A primeira delas é o lugar em que os objetos são alocados. Se eles se comportarem como os tipos de dados abstratos, talvez possam ser alocados em qualquer lugar. Isso significa que eles podem ser alocados estaticamente pelo compilador, como objetos dinâmicos na pilha em tempo de execução ou criados explicitamente no monte com um operador ou com uma função, como, por exemplo, `new`. Se todos eles forem alocados no monte, haverá a vantagem de ter-se um método uniforme de criação e de acesso por variáveis ponteiro ou por referência. Isso simplifica a operação de atribuição de objetos, tornando-a, em todos os casos, somente uma mudança de valor de ponteiro ou referência. Isso também permite que as referências a objetos sejam implicitamente desreferenciadas, simplificando a sintaxe de acesso.

A segunda questão refere-se aos casos em que os objetos são alocados no monte. A questão é se a desalocação é implícita, explícita ou ambas. Se ocorrer o primeiro caso, algum método implícito de reivindicação de armazenamento será necessário, como a contagem de referências ou a coleta de lixo, mas suscitará a questão de se ponteiros ou referências penduradas podem ser criados.

### 12.3.7 Vinculação Dinâmica e Estática

Como já discutimos, a vinculação dinâmica de mensagens a métodos em uma hierarquia de heranças é uma parte fundamental da programação orientada a objeto. A questão aqui é se toda vinculação de mensagens a métodos é dinâmica. A alternativa é permitir que o usuário especifique se uma vinculação específica deve ser dinâmica ou estática. A vantagem reside no fato da vinculação estática ser mais rápida. Assim, se uma vinculação não precisar ser dinâmica, por que pagar o preço? Completamos aqui nossa discussão das questões de projeto das linguagens de programação orientadas a objeto.

---

## 12.4 Visão Geral da Smalltalk

A Smalltalk é a linguagem de programação orientada a objeto definitiva. Esta seção apresenta algumas de suas características gerais. A Seção 12.5 descreve os aspectos específicos de um subconjunto da sintaxe e da semântica em pequena escala da Smalltalk. A Seção 12.6 descreve dois exemplos completos de programa em Smalltalk. A Seção 11.7 discute diversos de seus recursos em grande escala, especificamente em termos das questões de projeto descritas na Seção 12.3.

### 12.4.1 Características Gerais

Um programa em Smalltalk é composto inteiramente de objetos, e o conceito destes é verdadeiramente universal. Virtualmente tudo, desde itens tão simples como a constante inteira 2 a um sistema complexo de manipulação de arquivos, é um objeto. Sendo assim, eles são tratados uniformemente. Todos eles têm memória local, herdam capacidade de processamento inerente, capacidade de comunicarem-se com outros objetos e a possibilidade de herdarem métodos e variáveis de instância de ancestrais.

As mensagens podem ser parametrizadas com variáveis que referenciam objetos. As respostas às mensagens têm a forma deles e são usadas para retornar informações solicitadas ou para confirmar se o serviço solicitado foi concluído.

Todos os objetos Smalltalk são alocados no monte e referenciados por variáveis de referência que, por sua vez, são desreferenciadas implicitamente. Não há nenhuma instrução ou operação de desalocação. Toda desalocação é implícita, usando um processo de coleta de lixo para a reivindicação de armazenamento.

Diferentemente das linguagens híbridas como o C++ e a Ada 95, a Smalltalk foi projetada para apenas um paradigma de desenvolvimento de software, orientado a objeto. Além disso, ela não adota nada da aparência das linguagens imperativas. Sua pureza de propósito reflete-se em sua elegância simples e uniformidade de projeto.

#### **12.4.2 O Ambiente Smalltalk**

O ambiente da Smalltalk integra um editor de programa, um compilador, os recursos usuais de um sistema operacional e uma máquina virtual em um único sistema. A interface para este último é gráfica.

Um aspecto importante do ambiente Smalltalk está no fato dele ser quase inteiramente escrito em Smalltalk, e o usuário pode modificá-lo para que se ajuste a suas necessidades particulares. Portanto, a versão fonte do sistema dessa linguagem deve estar disponível ao usuário.

### **12.5 Introdução à Linguagem Smalltalk**

Esta seção apresenta um subconjunto da Smalltalk. Os tópicos discutidos aqui são suficientes para exprimir o sabor da programação em Smalltalk. Os recursos mais importantes não descritos aqui são a grande hierarquia de classes, fornecidas com um sistema Smalltalk que constitui a base para a maioria dos programas dela, e o poderoso ambiente de janelas em que seus programas são desenvolvidos.

#### **12.5.1 Expressões**

Os métodos Smalltalk são construídos a partir de expressões. Uma expressão especifica um objeto, que será o valor da expressão. A Smalltalk tem quatro tipos de expressões: literais, nomes de variáveis, expressões de mensagem e de bloco. As literais, as variáveis e as expressões de mensagem serão discutidas nas três subseções seguintes. As expressões de bloco serão descritas na Seção 12.5.4.1.

##### **12.5.1.1 Literais**

As literais mais comuns são números, cadeias e palavras-chave. Números são objetos literais que representam valores numéricos. Eles são bem diferentes das literais numéricas das linguagens imperativas comuns, as quais agem de uma maneira um tanto similar às constantes nomeadas, porque são associadas com localizações da memória que contêm seus valores. Na Smalltalk, literais numéricas são objetos caracterizados por seu protocolo de

Hidden page

símbolo de uma mensagem unária especifica um objeto receptor; o último, o método desse objeto que deve ser executado. Por exemplo, a mensagem

`primeiroAngulo seno`

envia uma mensagem sem parâmetro ao método `seno` do objeto `primeiroAngulo`. Lembre-se que todos os objetos são referenciados por ponteitos, de modo que `primeiroAngulo` é de fato um ponteiro para um objeto. O método `seno` (provavelmente) retorna um objeto número que é o valor do seno de `primeiroAngulo`.

As mensagens binárias têm um único parâmetro, um objeto, passado ao método especificado do objeto receptor especificado. Entre as mensagens binárias mais comuns estão as destinadas a operações aritméticas, como, por exemplo

`21 + 2`

e

`soma / cont`

Na primeira mensagem, o objeto receptor é o número `21`, para que é enviada a mensagem `+ 2`. Assim, a mensagem `21 + 2` passa o objeto parâmetro `2` para o método `+` do objeto `21`. O código usa o objeto `2` para construir um novo objeto. Nesse caso, `23`. Se o sistema já contiver `23`, o resultado será uma referência a ele em vez de um novo objeto.

Pode parecer estranho considerar `21` um objeto, mas na Smalltalk é perfeitamente natural que números sejam objetos com operações. As mais usuais para objetos inteiros são definidas na classe `Integer`, da qual elas são instâncias.

Na segunda mensagem acima, `/ cont` é enviada ao objeto referenciado por `soma`, o que resulta no objeto referenciado pela variável `cont` ser passado como parâmetro para o método `/` do objeto referenciado por `soma`.

As expressões palavra-chave especificam uma ou mais palavras-chave para organizar a correspondência entre os parâmetros reais da mensagem e os formais do método. Ou seja, as palavras-chave agem afinadas para selecionar o método ao qual a mensagem é dirigida. A intercalação daquelas e de parâmetros em mensagens aumenta a legibilidade. Métodos que aceitam mensagens de palavras-chave não são nomeados. Ao contrário, são identificados pelas mesmas. Considere o seguinte exemplo:

`primeiroVetor at: 1 put: 5`

Essa mensagem envia os objetos `1` e `5` a um método particular, `at:put:`, do objeto `primeiroVetor`. As palavras-chave `at:` e `put:` identificam os parâmetros formais do método para o qual `1` e `5`, respectivamente, devem ser enviados. O método para o qual esta mensagem é enviada inclui as palavras-chave da mensagem. Conforme mencionamos anteriormente, os métodos palavra-chave não têm nomes: ao contrário, eles são identificados por suas palavras-chave. Tal concatenação — nesse caso, `at:put:` — é chamada `seletor`.

As expressões de mensagens podem consistir em qualquer número de combinações dos seus três tipos, como em

`total - 3 * divisor.`

`primeiroVetor at: indice - 1 put: 77`

Para determinar como elas são avaliadas, a precedência e a associatividade de operadores de expressão devem ser conhecidas. As expressões unárias têm a precedência mais alta, seguidas pelas binárias e pelas de palavras-chave. Tanto as expressões unárias como as

binárias associam-se da esquerda para a direita. Note que isso é bem diferente das regras de precedência comumente usadas em linguagens como a Ada e o C++.

As expressões podem ser colocadas entre parênteses para forçar qualquer ordem de avaliação de operadores. A primeira expressão, colocada entre parênteses para ilustrar, mas não alterar sua ordem de avaliação normal, é:

```
(total - 3) * divisor
```

Essa expressão envia o 3 ao método — do objeto `total`. O valor da variável `divisor` é, então, enviado ao método `*` do objeto que resultou da primeira operação.

A expressão

```
primeiroVetor at: indice - 1 put: 77
```

envia 1 ao método — do objeto `indice`. O resultado dessa operação, juntamente com 77, é então enviado ao método `at:put:` do objeto `primeiroVetor`.

As mensagens podem ser dispostas em cascata — o que significa a possibilidade de enviar múltiplas mensagens ao mesmo objeto sem duplicar o nome do objeto receptor — separando-se os grupos seletor-parâmetro, ou mensagens, por ponto-e-vírgulas. As mensagens são enviadas seqüencialmente, como aparecem, da esquerda para a direita. Por exemplo,

```
nossaCaneta home; up; goto: 500@500; down; home
```

é equivalente ao seguinte:

```
nossaCaneta home.  
nossaCaneta up.  
nossaCaneta goto: 500@500.  
nossaCaneta down.  
nossaCaneta home
```

Essa seqüência desenha uma linha na tela, supondo que `nossaCaneta` seja uma instância da classe `Caneta`. Um objeto da classe `Caneta` é ilustrado na Seção 12.6.2.

Note que são usados pontos finais para separar mensagens enviadas a diferentes métodos e aparecem em linhas adjacentes. Isso é semelhante ao uso de ponto-e-vírgulas para separar instruções em programas Pascal.

### 12.5.2 Métodos

A forma sintática geral de um método Smalltalk é

```
padrão_mensagem [ | variáveis temporárias | ] instruções
```

em que os colchetes são metassímbolos indicando que aquilo que eles envolvem é opcional. Uma vez que a Smalltalk não tem declaração de tipo, as variáveis temporárias, quando presentes, somente precisam ser nomeadas em uma lista. Elas existem somente durante a execução do método no qual estão alinhadas. Nenhuma pontuação aparece no final de um método.

O padrão de uma mensagem corresponde ao cabeçalho de uma função em uma linguagem como o C. Os padrões de mensagem, protótipos delas mesmas, podem estar em uma de duas formas básicas. Para mensagens unárias ou binárias, somente o nome do método é incluído. Para as de palavra-chave, as palavras-chave e os nomes dos parâmetros formais formam o padrão da mensagem.

Um valor a ser retornado por um método é indicado precedendo-se a expressão que o descreve com um circunflexo (^). Em muitos casos, essa é a última expressão que aparece no método. Se nenhum valor de retorno for especificado, o próprio objeto receptor será o de retorno.

Um padrão de uma mensagem unária é simplesmente o nome do método. Um exemplo deste é

```
totalCorrente
^(totalVelho + valorNovo)
```

O método, nomeado `totalCorrente`, retorna o valor da expressão

```
totalVelho + valorNovo.
```

Os métodos binários são usados principalmente para operações aritméticas predefinidas, de modo que não serão aqui discutidos.

A forma geral do padrão de mensagem para métodos de palavra-chave é

```
chave_1: parâmetro_1 chave_2: parâmetro_2... chave_n: parâmetro_n
```

Considere o seguinte exemplo de método de palavra-chave, que não especifica um valor a ser retornado:

```
x: xCoord y: yCoord
nossaCaneta up; goto xCoord@yCoord; down.
```

Nesse método, que combina o seletor de mensagens `x:y:`, o objeto `nossaCaneta` é enviado às mensagens `up` `goto` (que usa os dois parâmetros `xCoord` e `yCoord`) e `down`. O padrão de mensagem é simplesmente uma lista dos pares palavra-chave/parâmetro formal do método.

Um exemplo de mensagem para esse método é

```
nossaCaneta x: 300 y: 400
```

Recursos adicionais dos métodos, inclusive variáveis temporárias, serão discutidos na Seção 12.5.6.

### 12.5.3 Instruções de Atribuição

A Smalltalk tem instruções de atribuição semelhantes, pelo menos na aparência, às de linguagens como o C++ e a Ada. Qualquer expressão de mensagem, objeto literal ou nome de variável pode estar no lado direito de uma instrução de atribuição. O lado esquerdo é um nome de variável, e o operador é especificado com uma seta apontando para a esquerda, como em

```
total <- 22.
soma <- total
```

O objeto particular referenciado por uma variável é mudado quando o nome desta aparece no lado esquerdo de uma atribuição. Nesse exemplo, a variável `total` é definida para referir-se ao objeto 22. Depois, a variável `soma` é definida para referir-se ao mesmo. A operação está estreitamente relacionada com a atribuição das variáveis ponteiro em linguagens imperativas típicas.

Lembre-se de que todos os métodos transmitem informações de volta aos remetentes que enviaram as mensagens. Para salvá-los, a expressão de mensagem é colocada no lado

Hidden page

```
somaIndice <- [soma + indice]
soma <- somaIndice value
```

Os blocos são sempre executados no contexto de suas definições, mesmo quando enviados como parâmetros a um objeto diferente. Desse modo, estão semanticamente relacionados com os parâmetros passados pelo nome do ALGOL 60.

Os blocos podem ser imaginados como declarações de procedimento que podem aparecer em qualquer lugar. À semelhança dos procedimentos, os blocos podem ter parâmetros. Os parâmetros de blocos são especificados em uma seção no início do bloco separada do restante do bloco por uma barra vertical (|). As especificações de parâmetro formal exigem que dois-pontos sejam anexados na extremidade esquerda de cada parâmetro. Uma vez que não há nenhum tipo declarado, as especificações incluem somente os nomes dos parâmetros formais, listados sem quaisquer pontuações de separação. Como um exemplo de bloco com parâmetros, considere o seguinte:

```
[ :x :y | soma <- x + 10. total <- soma * y]
```

Os blocos constituem um meio de coletar expressões, de modo que são uma maneira natural de formar estruturas de controle na Smalltalk.

#### 12.5.4.2 Iteração

Os blocos podem conter expressões relacionais, em cujo caso eles retornam um dos objetos booleanos predefinidos, **true** ou **false**. Esses blocos, às vezes, são chamados blocos condicionais. Os dois objetos, **true** e **false**, têm métodos que oferecem algumas das facilidades para construir estruturas de controle.

Laços de pré-teste lógico podem ser formados usando-se o método de palavra-chave **whileTrue:**, oferecido pela classe **block**, para enviar o bloco a ser controlado para um segundo bloco que contém a condição de laço. Tal método é definido para todos os blocos que retornam objetos booleanos. O método **whileTrue:** é definido para enviar **value** ao objeto que contém o método (ou **true** ou **false**), fazendo assim com que seu bloco paramétrico seja executado, como no seguinte:

```
cont <- 1.
soma <- 0.
[cont <= 20] "O bloco que contém a definição de laço"
    whileTrue: [soma <- soma + cont.
        cont <- cont + 1] "O corpo do laço"
```

Esse código realiza uma operação bastante convencional, sendo que o processo pelo qual ele o faz é significativamente diferente daquele usado pelas linguagens imperativas, ainda que possa ter uma aparência pouco convencional.

O controle de laço é realizado da seguinte maneira: o bloco que contém o código para adicionar **cont** a **soma** e incrementar **cont**, o segmento de código cuja execução deve ser controlada, é enviado como o parâmetro ao método **whileTrue:** do bloco condicional [**cont <= 20**]. O método **whileTrue:** envia **value** ao bloco condicional, fazendo assim com que o bloco seja avaliado. O resultado da avaliação é um objeto booleano. Se o resultado for **true**, o método **whileTrue:** fará com que o parâmetro enviado pela mensagem **whileTrue:** seja avaliada. Seu parâmetro é o bloco que contém as expressões da iteração. Depois que elas são avaliadas, o processo é repetido enviando o bloco de expressões a [**cont <= 20**] novamente. A repetição pára quando uma avaliação de [**cont <= 20**] produz **false** como um objeto resultado.

Suponhamos que o bloco de controle no exemplo acima fosse [cont <= 2], em vez de [cont <= 20]. O que apresentamos a seguir é um rastreamento das ações que ocorrem quando o código modificado é executado. Note que → indica uma mensagem que é enviada, e os comentários são delimitados por aspas.

```

cont <- 1
soma <- 0
[soma <- soma + cont. cont <- cont + 1] → [cont <= 2] whileTrue
value → [cont <= 2] "valor enviado por whileTrue"
[cont <= 2] retorna true
whileTrue avalia [soma <- soma + cont. cont <- cont + 1]
    (soma <- 1; cont <- 2) "resultados da avaliação"
[soma <- soma + cont. cont <- cont + 1] → [cont <= 2] whileTrue
value → [cont <= 2] "valor enviado por whileTrue"
[cont <= 2] retorna true
whileTrue avalia [soma <- soma + cont. cont <- cont + 1]
    (soma <- 3; cont <- 3) "resultados da avaliação"
[soma <- soma + cont. cont <- cont + 1] → [cont <= 2] whileTrue
value → [cont <= 2]
[cont <= 2] retorna false

```

Outra estrutura de controle de laço comum é a simples repetição com um controle de contador. Para isso, há um método para números inteiros chamado **timesRepeat:**. Quando **timesRepeat:** é enviado a um inteiro, tendo um bloco como parâmetro, o bloco é executado o número de vezes igual ao valor do número inteiro. Por exemplo

```

xCubo <- 1.
3 timesRepeat: [xCubo <- xCubo * x]

```

computa o cubo de **x** por meio de um processo bastante longo.

Estruturas de controle similares aos laços **do** do FORTRAN podem ser construídas com alguns dos métodos de inteiros. Os dois mais úteis deles são **to:do:** e **to:by:do:**. Primeiro, considere o método **to:do:**. O parâmetro **to:** é uma expressão de número inteiro cujo valor serve como terminal. O parâmetro **do:** é um bloco que deve ser executado pelo método número inteiro. O método **to:do:** gera, internamente, uma seqüência de valores, iniciando-se pela literal inteira para a qual a mensagem é enviada e encerrando-se com o valor de parâmetro **to:**. Por exemplo, considere a seguinte mensagem:

```

1 to: 5 do: [soma <- soma + x]

```

O bloco é executado cinco vezes. Os valores internos produzidos e retornados pelo objeto 1 são 1, 2, 3, 4 e 5.

O bloco que forma um corpo de laço pode ter um parâmetro. Os valores internos criados pela mensagem são implicitamente atribuídos ao parâmetro. Eles são aqueles retornados pelo objeto numérico ao qual toda a mensagem é enviada. Por exemplo, considere a seguinte mensagem:

```

2 to: 10 by: 2 do: [:par | soma <- soma + par]

```

Essa mensagem faz com que o bloco seja executado cinco vezes, mas, no caso do **par**, o parâmetro do bloco, assume os valores internos 2, 4, 6, 8 e 10.

Hidden page

Hidden page

enviada ao resultado da execução do método `=`. Se o resultado for o objeto `true`, como seria para a mensagem `0 factorial`, o valor `1` será retornado ao remetente de `fatorial`. Se o resultado da execução do método `=` for `false`, nenhuma ação será posta em prática porque o método `ifTrue` em `false` é definido para fazer nada.

A expressão booleana seguinte, `self < 0`, envia o parâmetro `0` ao método `<` do número inteiro para o qual `fatorial` foi enviada. O restante da mensagem `fatorial` (`ifTrue:ifFalse: inteira`) é, então, enviada ao resultado da execução do método `<`. Se o objeto resultante for `true`, a mensagem de erro será enviada ao objeto para o qual `fatorial` foi enviada. Se o resultado do `<` for `false`, o bloco da parte `ifFalse:` de `ifTrue:ifFalse:` será executado, e o resultado será retornado ao remetente de `fatorial`.

Para entender a mensagem `ifFalse:`, você deve ter uma idéia clara das regras de precedência de avaliação de mensagens. Nessa expressão, há duas expressões binárias (aqueles com `*` e `-`) e uma expressão unária (`fatorial`). Lembre-se de que as expressões unárias têm precedência sobre as binárias, a menos que estas estejam entre parênteses. Além disso, todas as expressões têm associatividade à esquerda. Agora, a ordem de avaliação é clara: primeiro, `- 1` é enviado a `self`, produzindo um objeto que é uma unidade menor do que o objeto para o qual a mensagem foi enviada. Então, `fatorial` é enviado a esse novo objeto. O resultado final da mensagem, depois de toda recursão, é enviado com `*` para `self`, o objeto original para o qual `fatorial` foi enviado. O resultado dessa mensagem é o valor `fatorial`.

Esse exemplo ilustra a similaridade da recursão em semânticas de linguagem amplamente diferentes.

## 12.6 Exemplo de Programas Smalltalk

### 12.6.1 Um Manipulador\* de Tabela Simples

O exemplo de classe desta seção demonstra que problemas simples de gerenciamento de tabela típicos de linguagens imperativas comuns também podem ser implementados facilmente em Smalltalk. O problema é construir um programa que crie e pesquise uma tabela de nomes de departamento e seus números de código. Devido à falta de tipificação estática, o sistema resultante poderia ser usado por qualquer tabela composta de dois vetores paralelos de dados, no qual as pesquisas se baseiam nos elementos de dados do primeiro vetor.

Um dos recursos interessantes usados no programa é a vinculação dinâmica de faixas de índice a vetores. Dois destes são sempre exatamente do tamanho dos dados armazenados. Cada adição à tabela simplesmente aumenta o seu tamanho. Note que embora isso possa parecer interessante e eficiente quanto ao espaço, é altamente ineficiente em termos de tempo de execução. Cada adição provoca a criação de dois novos vetores e uma cópia do antigo conteúdo deles para os novos vetores — um processo que consome muito tempo.

\*N. de T. Manipulador: tradução literal de `handler` (rotina de tratamento) — Uma sub-rotina de software que realiza uma tarefa em particular. Por exemplo, quando um erro é detectado, uma rotina de tratamento de erro é chamada para recuperar a condição deste.

Uma das omissões do programa é a falta de um método para remover uma entrada

```

class name                                     CodDept
superclass                                    Object
instance variable names                      nomes
                                                codigos

" Métodos de classe"
"Cria uma instância"
    new
        ^ super new

" Métodos da instância"
"Número de entradas da tabela"
    tamanho
        ^ nomes size

"Busca o código de um departamento"
    at: nome | indice |
        indice <- self indexOf: nome.
        indice = 0
        ifTrue: [self error: 'Erro-Nome não está na tabela' ]
        ifFalse: [^codigos at: indice]

"Instala um novo código; cria uma entrada, se necessário"
    at: nome put: codigos | indice |
        indice <- self indexOf: nome.
        indice = 0
        ifTrue: [indice <- self newIndexOf: nome].
        ^ codes at: indice put: code

"Pesquisa o índice de um nome de departamento dado"
    indexOf: nome
        1 to: nomes tamanho do:
            [:indice | (nomes at: indice) = nome ifTrue:
                (^indice)].
        ^ 0

"Cria uma nova entrada com o nome dado e retorna o índice"
    newIndexOf: nome
        self grow.
        nomes at: nomes tamanho put: nome.
        ^ nomes tamanho

"Estenda a tabela em um elemento e coloque o novo nome"
    grow | nomeVelho codVelho|
        nomeVelho <- nomes.
        codVelho <- codigos.
        nomes <- Array new: nomes tamanho + 1.
        codigos <- Array new: codigos tamanho + 1.
        nomes replaceFrom: 1 to: nomeVelho tamanho with: nomeVelho.

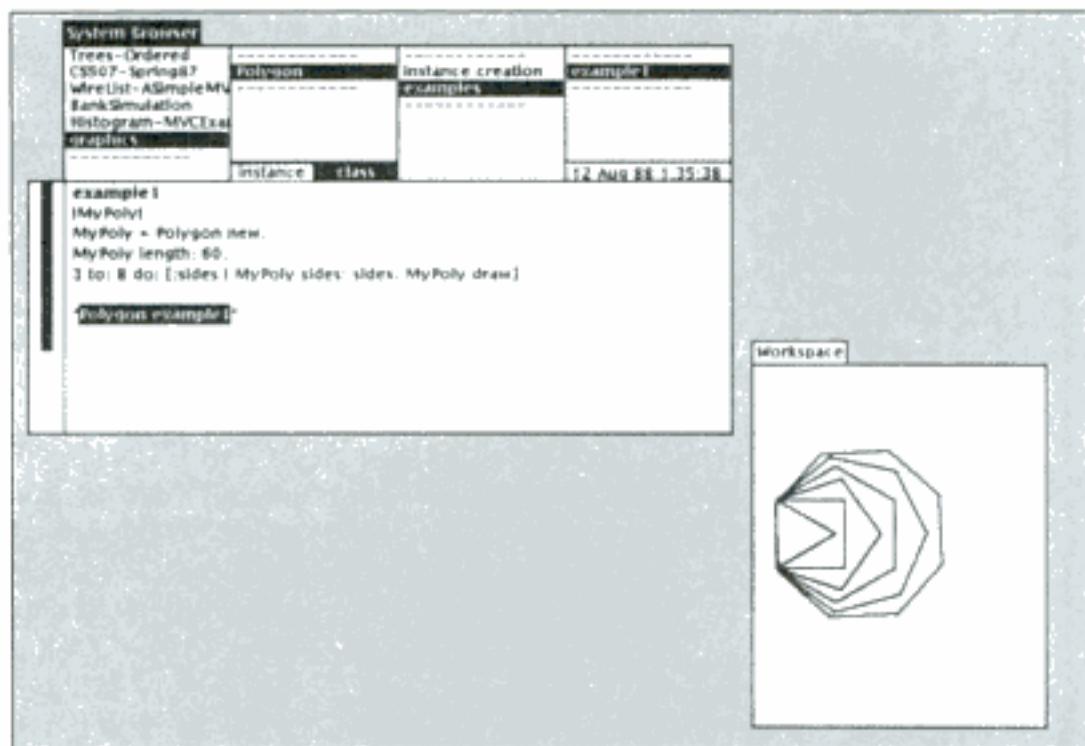
```

Hidden page

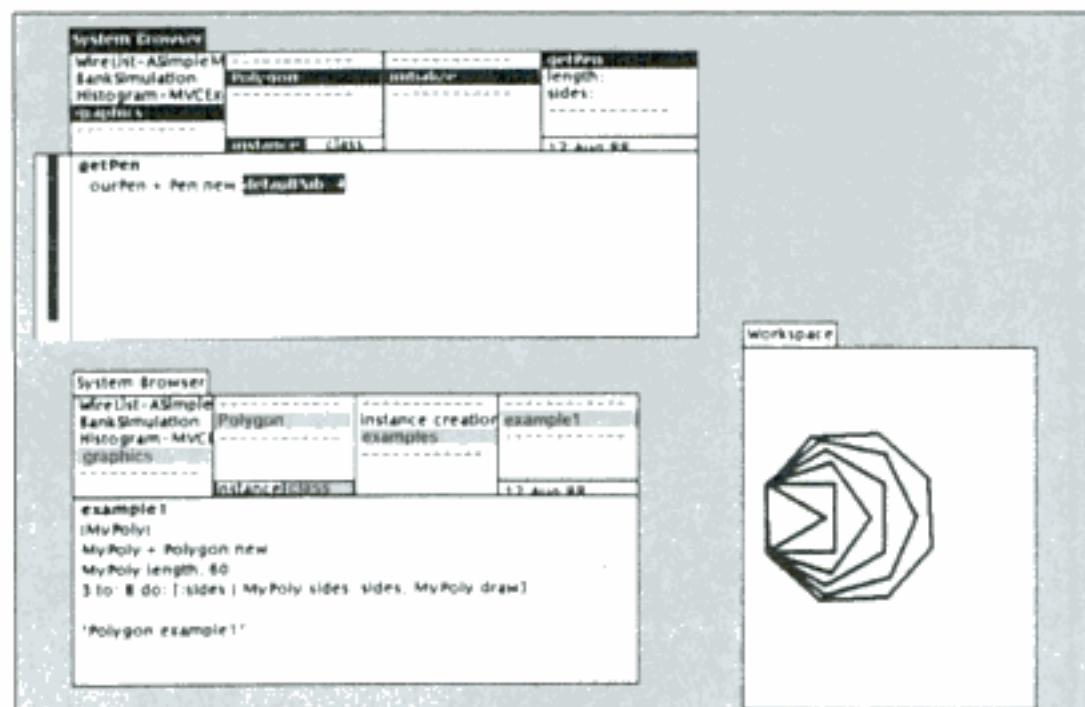
Hidden page

Hidden page

Hidden page



**FIGURA 12.3** Polígonos concêntricos do objeto **Polígono**.



**FIGURA 12.4** Polígonos concêntricos com **nib** definido em 4.

ocorrerá um erro. É importante lembrar que a procura de método é dinâmica — ela se desenvolve quando a mensagem é enviada. A Smalltalk não vincula, em nenhuma circunstância, mensagens a métodos estaticamente.

A única verificação de tipos na Smalltalk é dinâmica, e o único erro de tipo ocorre quando uma mensagem é enviada a um objeto que não tem nenhum método coincidente, ou localmente ou por meio de herança. Esse é um conceito de verificação de tipos diferente da maioria das outras linguagens. A verificação de tipos em Smalltalk tem a meta simples de assegurar que a mensagem coincida com um método.

As variáveis Smalltalk não são tipificadas; qualquer nome pode ser vinculado a qualquer objeto. Como um resultado direto, ela suporta polimorfismo dinâmico. Todo código Smalltalk é genérico em termos de que os tipos das variáveis são irrelevantes, contanto que sejam consistentes. O significado de uma operação (método ou operador) sobre uma variável é determinado pela classe da variável à qual ela está atualmente vinculada.

O ponto dessa discussão é que, contanto que os objetos referenciados em uma expressão tenham métodos para as mensagens da expressão, os tipos dos objetos são irrelevantes. Isso significa que todo código é genérico: nenhum está ligado a um tipo particular.

### 12.7.2 Herança

Uma subclasse Smalltalk herda todas as variáveis e métodos de instância e os métodos da classe de sua superclasse. A subclasse também pode ter suas próprias variáveis de instância, que devem ter nomes diferentes dos das variáveis em suas classes ancestrais. Finalmente, a subclasse pode definir novos métodos e redefinir outros que já existem em uma ancestral. Quando uma subclasse tem um método cujo nome e protocolo são os mesmos que a classe ancestral, o método da subclasse oculta o da classe ancestral. O acesso ao método que está oculto é garantido prefixando-se a mensagem com a pseudovariável `super`. Isso faz com que a procura do método inicie-se na superclasse em vez de localmente.

Uma vez que as entidades de uma classe-pai não podem ser escondidas das subclases, todas subclasses são subtipos. Além disso, toda herança é de implementação.

A Smalltalk suporta herança simples; ela não permite a múltipla.

## 12.8 Avaliação da Smalltalk

---

A Smalltalk é uma linguagem pequena, não obstante o seu sistema ser grande. A sintaxe da linguagem é simples e muito regular. Ela é um bom exemplo do poder que pode ser oferecido por uma linguagem pequena se for construída em torno de um conceito simples, mas poderoso. No caso da Smalltalk, esse conceito é que a programação pode ser feita usando-se somente uma hierarquia de classes construída utilizando-se herança, objetos e passagem de mensagens.

Em comparação com programas em linguagens imperativas compiladas convencionais, os programas Smalltalk equivalentes são significativamente mais lentos. Ainda que seja teoricamente interessante que a indexação de vetores e laços possam ser oferecidas dentro do modelo de passagem de mensagens, a eficiência é um fator importante na avaliação das linguagens de programação. Portanto, a eficiência será evidentemente uma questão a ser considerada na maioria das discussões sobre a aplicabilidade prática da Smalltalk.

A vinculação dinâmica da Smalltalk impede que erros de tipo sejam detectados até a execução. Um programa pode ser escrito e compilado ainda que inclua mensagens a métodos não-existentes. Isso provoca um número de reparos de erros muito maior posteriormente no desenvolvimento do que ocorreria em uma linguagem com tipos estáticos.

A interface do usuário da Smalltalk teve um importante impacto na computação: o uso integrado de janelas, de dispositivos apontadores por meio de mouse, de menus secundários ou suspensos dominam os sistemas contemporâneos.

Talvez o maior impacto da Smalltalk seja o avanço da programação orientada a objeto, que agora é a metodologia de projeto e codificação mais usada.

## 12.9 Suporte para Programação Orientada a Objeto em C++

O Capítulo 2 descreve como o C++ evoluiu a partir do C e da SIMULA 67, com a meta de projeto de dar suporte para a programação orientada a objeto. Como as classes C++ são usadas para suportar tipos de dados abstratos, elas foram discutidas no Capítulo 11. O suporte do C++ para outros aspectos fundamentais da programação orientada a objeto será explorado nesta seção. O conjunto inteiro de detalhes das classes, da herança e da vinculação dinâmica do C++ é grande e complexo. Esta seção discutirá somente os mais importantes entre estes tópicos, especificamente aqueles diretamente relacionados com as questões de projeto discutidas na Seção 12.3.

### 12.9.1 Características Gerais

Uma vez que uma das mais importantes considerações de projeto do C++ era o fato de ter compatibilidade retrógrada<sup>1</sup> quase completa com o C, ele mantém o seu sistema de tipos e adiciona classes a ele. Portanto, o C++ tem tanto os tipos das linguagens imperativas tradicionais como a estrutura de classes de uma linguagem orientada a objeto. Isso o torna híbrido, suportando tanto programação baseada em procedimentos como programação orientada a objeto.

Os objetos do C++ podem ser alocados nos mesmos lugares onde se pode alocar variáveis no C. Eles podem ser alocados estaticamente pelo compilador, dinamicamente na pilha ou no monte usando-se o operador `new`. A desalocação explícita é necessária, usando o operador `delete`. Não há nenhuma reclamação de armazenamento implícito.

Todas as classes C++ incluem pelo menos um método construtor, usado para inicializar os membros de dados do novo objeto. Os métodos construtores são chamados implicitamente quando um objeto é criado. Se qualquer um dos membros de dados for alocado no monte, o construtor fará essa alocação. Se nenhum construtor estiver incluído em uma definição de classe, o compilador incluirá um construtor trivial. O construtor padrão chama o construtor da classe-pai, se houver uma (veja a Seção 12.9.2).

A maioria das definições de classes inclui um método destrutor, implicitamente chamado quando um objeto da classe deixa de existir. O destrutor é usado para excluir os

<sup>1</sup>N. de T. Compatibilidade retrógrada (ou descendente): tradução literal de *backward compatible*. Também chamada *downward compatible*. Refere-se a hardware ou a software compatível com versões anteriores.

membros de dados alocados no monte. Ele também pode ser usado para registrar parte ou todo o estado do objeto imediatamente antes de deixar de existir, usualmente para propósitos de depuração.

Enquanto a Smalltalk não permite que o usuário exerça qualquer controle de acesso sobre as variáveis e sobre os métodos de instância em uma classe, o C++ fornece uma variedade desses controles de acesso. Alguns deles servem para controlar aquilo que o cliente pode e não pode ver na definição de classes. Outros servem para controlar o acesso por subclasses. Os controles de acesso do C++ para entidades de classes serão discutidos na Seção 12.9.2.

### 12.9.2 Herança

Uma classe C++ pode ser derivada de uma classe existente, que é, então, sua classe-pai ou básica. Diferentemente da Smalltalk, uma classe C++ também pode ser independente, sem uma superclasse.

Lembre-se de que os dados definidos em uma definição de classe são chamados membros de dados desta, e que as funções determinadas em uma definição de classe são chamadas funções-membro dela. Alguns ou todos os membros de dados e funções-membro da classe básica podem ser herdados pela classe derivada, que também pode adicionar novos membros de dados e de funções-membro e modificar os herdados. A acessibilidade aos membros das subclasses pode ser diferente daquela dos membros correspondentes da classe básica. É assim que as classes derivadas do C++ podem ser impedidas de serem subtipos.

Lembre-se do Capítulo 11 que os membros de classes podem ser privados, protegidos ou públicos. Os privados são acessíveis somente pelas funções-membro e amigas da classe. Tanto as funções como as classes podem ser declaradas como amigas de uma classe e, assim, terem acesso a seus membros privados. Os públicos são acessíveis por qualquer função, enquanto os membros protegidos são como os privados, exceto em classes derivadas, cujo acesso será descrito abaixo. Classes derivadas podem modificar a acessibilidade a seus membros herdados. A forma sintática de uma classe derivada é

```
class classe_derivada : modo_de_acesso nome_da_classe
    {declarações do membro de dados e função membro};
```

O modo\_de\_acesso pode ser **public** ou **private**. Os membros públicos e protegidos de uma classe básica são também públicos e protegidos, respectivamente, em uma derivada pública. Em uma classe privada derivada, os membros públicos e protegidos da classe básica são privados. Assim, em uma hierarquia de classes, a classe privada derivada interrompe o acesso a todos os membros de todas as classes ancestrais a todas as suas sucessoras, e os protegidos podem ser ou não acessíveis às subclasses subsequentes (depois da primeira). Os membros privados de uma classe básica são herdados por uma derivada, mas não são visíveis aos membros desta e, portanto, não têm nenhuma utilidade lá. Considere o seguinte exemplo:

```
class classe_base {
    private:
        int a;
        float x;
    protected:
        int b;
```

```

        float y;
public:
    int c;
    float z;
};

class subclasse_1 : public classe_base { ... };
class subclasse_2 : private classe_base { ... };

```

Em `subclasse_1`, `b` e `y` são protegidos, e `c` e `z` são públicos. Em `subclasse_2`, `b`, `y`, `c` e `z` são privados. Nenhuma classe derivada de `subclasse_2` pode ter membros com acesso a qualquer membro de `classe_base`. Os membros de classe `a` e `x` em `classe_base` não são acessíveis em `subclasse_1` ou em `subclasse_2`.

Sob a derivação de classe privada, nenhum membro da classe-pai é implicitamente visível para as instâncias da classe derivada. Qualquer membro que deva se tornar visível deve ser reexportado nesta. Essa reexportação, com efeito, exime o membro de estar oculto, não obstante a derivação ser privada. Por exemplo, considere a seguinte definição de classe:

```

class subclasse_3 : private classe_base {
    classe_base :: c;
    ...
}

```

Agora, as instâncias de `subclasse_3` podem acessar `c`. No que se refere a `c`, é como se a derivação fosse pública. Os dois-pontos duplos (`::`) nessa definição de classe são um operador de resolução de escopo. Ele especifica a classe na qual sua entidade seguinte é definida.

Considere o seguinte exemplo de herança C++, na qual uma classe de lista encadeada geral é definida e depois usada para definir duas subclasses úteis:

```

class lista_ligada_simples {
    class vertice {
        friend class lista_ligada_simples;
    private:
        vertice *ligacao;
        int conteudos;
    };
    private:
        vertice *topo;
    public:
        lista_ligada_simples() {topo = 0};
        void insere_no_topo(int);
        void insere_no_fim(int);
        int remove_do_topo();
        int vazio();
    };

```

A classe aninhada, `vertice`, define uma célula da lista encadeada como sendo uma localização de número inteiro e um ponteiro para uma célula. Ela lista `lista_ligada_simples` como uma **amiga**, garantindo assim que os objetos da `lista_ligada_simples` acessem seus dois membros de classe, `ligacao` e `conteudos`. Isso é necessário porque as classes envolventes não têm nenhum direito de acesso especial aos membros de suas aninhadas.

A classe envolvente, `lista_ligada_simples`, tem apenas um único membro de dados, um ponteiro para agir como cabeçalho da lista. Ela contém uma função construtora, que simplesmente fixa `topo` no valor de ponteiro nulo. As quatro funções-membro permitem que vértices sejam inseridos em qualquer extremidade de um objeto-lista, que sejam removidos de uma extremidade deste, e que listas sejam testadas sobre estarem vazias.

As definições seguintes apresentam classes pilha e fila, ambas baseadas na classe `lista_ligada_simples`:

```
class pilha : public lista_ligada_simples {
public:
    pilha() {}
    void empilha(int valor) {
        lista_ligada_simples :: insere_no_topo(int valor);
    }
    int desempilha() {
        return lista_ligada_simples :: remove_do_topo();
    }
};

class fila: public lista_ligada_simples {
public:
    fila() {}
    void enfileira(int valor) {
        lista_ligada_simples :: insere_no_fim(int valor);
    }
    int desenfileira() {
        lista_ligada_simples :: remove_do_topo();
    }
};
```

Note que os objetos tanto da subclasse `pilha` como da `fila` podem acessar a função vazia definida na classe básica, `lista_ligada_simples` (porque ela é uma derivação pública). Ambas subclasses definem funções construtoras que nada fazem. Quando um objeto de uma subclasse é criado, o seu construtor apropriado é implicitamente chamado. Depois, qualquer construtor aplicável da classe básica é chamado. Assim, em nosso exemplo, quando um objeto do tipo `pilha` é criado, o construtor `pilha` é chamado e não faz nada. Depois, o construtor em `lista_ligada_simples` é chamado e faz a inicialização necessária.

As classes `pilha` e `fila` padecem do mesmo problema sério: os objetos de ambas podem acessar todos os membros públicos da classe-pai, `lista_ligada_simples`. Portanto, um objeto `pilha` poderia acessar `insere_no_fim`, destruindo, assim, a integridade de sua pilha. Similarmente, um objeto `fila` poderia acessar `insere_no_topo`. Esses acessos indesejáveis são permitidos porque tanto `pilha` como `fila` são subtipos de `lista_ligada_simples`. Essas duas classes derivadas podem ser escritas de maneira a torná-las não-subtipos de sua classe-pai usando-se uma derivação `private` ao invés `public`. Então, ambas também precisarão reexportar `vazia`, porque ela se tornará oculta a suas instâncias. As novas definições dos tipos `pilha` e `fila`, chamadas `pilha_2` e `fila_2` são mostradas no seguinte:

```
class pilha_2 : private lista_ligada_simples {
public:
    pilha_2() {}
```

```

        void empilha(int valor) {
            lista_ligada_simples :: insere_no_topo(int valor);
        }
        int desempilha() {
            return lista_ligada_simples :: remove_do_topo();
        }
        lista_ligada_simples:: vazia;
    };
    class fila_2 : private lista_ligada_simples {
        public:
            fila_2() {}
            void enfileira(int valor) {
                lista_ligada_simples :: insere_no_fim(int valor);
            }
            int desenfileira() {
                lista_ligada_simples :: remove_do_topo();
            }
            lista_ligada_simples:: vazia;
    };
}

```

As duas versões de `pilha` e `fila` ilustram a diferença entre subtipos e tipos derivados que não são subtipos.

Uma das razões pelas quais os amigos (*friends*) são necessários encontra-se na necessidade de escrever um subprograma que deve acessar os membros de duas classes diferentes. Por exemplo, suponhamos que um programa use uma classe para vetores e uma para matrizes, e que um subprograma seja necessário para multiplicar objetos dessas duas classes. No C++, a função de multiplicar é simplesmente transformada em amiga de ambas.

O C++ oferece a opção de herança múltipla ao permitir que mais de uma classe seja nomeada como pai de uma nova classe. Por exemplo,

```

class A {...};
class B {...};
class C : public A, public B {...};

```

A classe `C` herda todos os membros tanto de `A` como de `B`. Se acontecer de tanto `A` como `B` incluírem membros com o mesmo nome, eles poderão ser referenciados de maneira não-ambígua em objetos do `C` usando-se o operador de resolução de escopo.

Os métodos de sobreposição no C++ devem ter exatamente o mesmo protocolo que o sobreposto. Se houver alguma diferença nos protocolos, o método da subclasse será considerado novo, sem relação com o de mesmo nome na classe ancestral.

### 12.9.3 Vinculação Dinâmica

Todas as funções-membro assim definidas até aqui são vinculadas estaticamente, ou seja, uma chamada a uma delas é vinculada estaticamente a uma definição de função. No C++, uma variável ponteiro ou referência com o tipo de uma classe básica pode ser usada para apontar para objetos de qualquer classe derivada daquela, tornando-a uma variável polimórfica. Quando esta última é usada para chamar uma função definida em uma das classes derivadas, a chamada deverá estar vinculada dinamicamente à definição da função correta. As funções-membro que devem estar dinamicamente vinculadas devem ser declaradas como

virtuais precedendo-se seus cabeçalhos com a palavra reservada **virtual**, que somente pode aparecer em um corpo de classe.

Considere a situação em que se tem uma classe básica chamada **forma**, juntamente com um conjunto de classes derivadas de diferentes tipos de formas, como, por exemplo, círculos, retângulos e assim por diante. Se essas formas precisarem ser exibidas, a função-membro de exibição, **desenhar**, deve ser única para cada subclasse ou para cada tipo de forma. Essas versões de **desenhar** devem ser definidas como virtuais. Quando uma chamada a **desenhar** é feita com um ponteiro para a classe básica das derivadas, essa chamada deve ser vinculada dinamicamente à função membro da classe derivada correta. O exemplo seguinte tem as definições da situação que acabamos de descrever:

```
class forma {
public:
    virtual void desenhar() = 0;
    ...
}
class circulo : public forma {
public:
    virtual void desenhar() { ... }
    ...
}
class retangulo : public forma {
public:
    virtual void desenhar() { ... }
    ...
}
class quadrado : public retangulo {
public:
    virtual void desenhar() { ... }
    ...
}
```

Dadas essas definições, o que apresentamos a seguir tem exemplos tanto de chamadas vinculadas estaticamente como dinamicamente:

```
quadrado s;
retangulo r;
forma &ref_forma = s;      // uma referência ao quadrado s
ref_forma.desenhar();     // vinculado dinamicamente a desenhar
                         // em quadrado
r.desenhar();            // vinculado estaticamente a desenhar
                         // em retangulo
```

Note que a função **desenhar** na definição da classe básica **forma** acima é fixada em zero. Tal sintaxe peculiar é usada para indicar que a função-membro é uma **função virtual pura**, significando que ela não tem nenhum corpo e que não pode ser chamada. Ela deve ser redefinida em classes derivadas. O propósito de uma função virtual pura é fornecer a interface de uma função sem revelar uma de suas implementações. Essa é uma nova forma de ocultação de informação ou encapsulamento.

Qualquer classe que inclua uma função virtual pura é uma **classe abstrata**. Nenhum objeto de uma classe abstrata pode ser criado. Em sentido estrito, um tipo de dado abstrato não pode ter objetos concretos mas, ao contrário, é usado somente para represen-

tar os conceitos de um tipo. As subclasses dele podem, é claro, ter objetos. O C++ oferece classes abstratas para modelar esses tipos realmente abstratos. Se uma subclass de uma classe abstrata não redefinir uma função virtual pura de sua classe-pai, essa função permanecerá como uma função virtual pura.

Juntas, as classes abstratas e a herança suportam uma técnica poderosa para o desenvolvimento de software. Elas permitem que tipos sejam definidos hierarquicamente, de maneira que os tipos relacionados podem ser subclasses de tipos verdadeiramente abstratos que definem suas características abstratas comuns.

A vinculação dinâmica permite que o código com membros como `desenhar` seja escrito antes de todas ou até mesmo de qualquer uma das versões de `desenhar`. Novas classes derivadas poderiam ser adicionadas anos mais tarde, sem exigir qualquer mudança no código que usa esses membros dinamicamente vinculados. Eis um recurso poderoso das linguagens orientadas a objeto.

#### 12.9.4 Avaliação

É natural comparar os recursos orientados a objeto do C++ com os da Smalltalk, o que é feito nesta seção.

A herança do C++ é mais confusa do que a da Smalltalk em termos de controle de acesso. Usando tanto os controles de acesso dentro das definições de classes como os controles de acesso de derivação, e também a possibilidade de funções e de classes amigas, o programador C++ tem um controle altamente detalhado do acesso a membros da classe. Além disso, não obstante haver certo debate sobre seu valor real, o C++ oferece herança múltipla, ao passo que a Smalltalk permite somente a simples.

No C++, o programador pode especificar se deve ser usada a vinculação estática ou a dinâmica. Uma vez que a primeira é mais rápida, é possível obter uma vantagem para aquelas situações em que a outra não é necessária. Além disso, até mesmo a vinculação dinâmica no C++ é rápida quando comparada com a da Smalltalk. Vincular uma chamada a função-membro virtual no C++ a uma definição de função tem um custo fixo, independentemente de quanto distante na hierarquia de heranças a definição aparece. As chamadas a funções virtuais exigem somente mais cinco referências de memória do que as chamadas vinculadas estaticamente (Stroustrup, 1988). Na Smalltalk, entretanto, as mensagens são sempre vinculadas dinamicamente a métodos, quanto mais distante na hierarquia de heranças o método correto estiver, mais tempo ele tomará. A desvantagem de permitir que o usuário decida quais vinculações são estáticas e quais são dinâmicas é que o projeto original deve incluir tais decisões, que talvez tenham de ser mudadas mais tarde.

A verificação estática de tipos do C++ obteve uma vantagem significativa sobre a Smalltalk, na qual toda verificação de tipos é dinâmica. Um programa Smalltalk pode ser compilado com mensagens a métodos não-existentes, então descobertos quando o programa é executado. Um compilador C++ localiza tais erros, que, quando detectados na compilação, são menos dispendiosos do que os encontrados durante a execução.

A Smalltalk é essencialmente sem-tipo, querendo isso dizer que todo o código é efetivamente genérico. Isso proporciona um bocado de flexibilidade, mas a verificação de tipos estática é sacrificada. O C++ fornece classes genéricas por meio de sua facilidade de modelos (conforme descrevemos no Capítulo 11), que mantém os benefícios da verificação de tipos estática.

A principal vantagem da Smalltalk reside na elegância e na sua simplicidade, resultantes da filosofia única de seu projeto. Ela é total e completamente dedicada ao paradigma orientado a objeto, destituída das transigências impostas pelos caprichos de uma base de

Hidden page

Hidden page

Hidden page

## 12.11 Suporte para Programação Orientada a Objeto em Ada 95

A Ada 95 derivou da Ada 83, com algumas extensões significativas. Esta seção apresenta um breve exame das extensões projetadas para suportar a programação orientada a objeto. Uma vez que a Ada 83 incluiu construções para criar tipos de dados abstratos, discutidos no Capítulo 11, os recursos restantes necessários se destinaram a suportar a herança e a vinculação dinâmica. Os objetivos de projeto para tais recursos incluíram as metas de exigir mudanças mínimas no tipo e nas estruturas de pacote da Ada 83 e manter o máximo possível a verificação de tipos estática.

### 12.11.1 Características Gerais

As classes Ada 95 são uma nova categoria de tipos chamada **tipos-marcados** (*tagged types*), os quais são registros ou tipos privados. Eles são encapsulados em pacotes, que permitem que sejam compilados separadamente. Os tipos-marcados são assim chamados porque cada objeto de um tipo-marcado inclui implicitamente uma marca (*tag*) mantida pelo sistema indicando seu tipo. Os subprogramas que definem as operações em um tipo-marcado aparecem na mesma lista de declaração que a do tipo. Considere este exemplo:

```
package PESSOA_PKG is
    type PESSOA is tagged private;
    procedure EXIBE(P : in out PESSOA) ;
    private
        type PESSOA is tagged
            record
                NOME : STRING(1..30);
                ENDERECO : STRING(1..30);
                IDADE : INTEGER;
            end record;
    end PESSOA_PKG;
```

Esse pacote define o tipo PESSOA que pode ser usado como tal ou como a classe-pai de outras classes derivadas.

Diferentemente do C++, não há nenhuma chamada de subprogramas construtores ou destrutores em Ada 95. Tais subprogramas podem ser escritos, mas devem ser chamados explicitamente pelo programador.

### 12.11.2 Herança

A Ada 83 tem uma forma restrita de herança em seus tipos derivados e em seus subtipos. Em ambos, um novo tipo pode ser definido em função de um já existente. Mas a única modificação permitida é restringir a faixa de valores do novo tipo. Os tipos derivados da Ada 95 baseiam-se em tipos-marcados. Novas entidades são adicionadas às herdadas incluindo uma definição de registro. Considere o exemplo:

```

with PESSOA_PKG; use PESSOA_PKG;
package ESTUDANTE_PKG is
    type ESTUDANTE is new PESSOA with
        record
            MEDIA_PONTO_GRAU : FLOAT;
            NIVEL_DO_GRAU : INTEGER;
        end record;
    procedure EXIBE (ST : in ESTUDANTE);
end ESTUDANTE_PKG;

```

Nesse exemplo, o tipo derivado `ESTUDANTE` é definido para ter as entidades de sua classe-pai, `PESSOA`, juntamente com as novas entidades `MEDIA_PONTO_GRAU` e `NIVEL_DO_GRAU`. Ele também redefine o procedimento `EXIBE`. Essa nova classe é definida em um pacote separado para permitir que ela seja mudada sem exigir recompilação do pacote que contém a definição do tipo-pai.

Por esse mecanismo de herança, não existe nenhuma maneira de impedir que as entidades da classe-pai sejam incluídas na derivada. Portanto, classes derivadas podem estender somente as classes-pai e, portanto, são subtipos.

Suponhamos que temos as seguintes definições:

```

P1 : PESSOA;
S1 : ESTUDANTE;
FRED : PESSOA := ("FRED", "321 Mulberry Lane", 35);
FREDDIE : ESTUDANTE := ("FREDDIE", "725 Main St.", 20, 3.25, 3);

```

Uma vez que `ESTUDANTE` é um subtipo de `PESSOA`, a atribuição

```
P1 := FREDDIE;
```

deve ser válida, e é. As entidades `MEDIA_PONTO_GRAU` e `NIVEL_DO_GRAU` de `FREDDIE` são simplesmente ignoradas na coerção exigida.

A questão evidente agora é se uma atribuição na direção oposta é válida; ou seja, se podemos atribuir uma pessoa a um estudante? Na Ada 95, isso é válido de uma maneira que inclua as entidades da subclasse. Em nosso exemplo, o seguinte é legal:

```
S1 := (FRED, 3.05, 2);
```

Para derivar uma classe que não inclui todas as entidades da classe-pai, são usados pacotes de biblioteca filhos. Um pacote de biblioteca filho é simplesmente um pacote cujo nome é prefixado com o do pacote-pai. Pacotes de biblioteca filhos também podem ser usados em lugar das definições amigas do C++. Por exemplo, se for necessário escrever um subprograma de acesso aos 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 poderá acessar os membros de ambos.

A Ada 95 não oferece herança múltipla. Há uma maneira de conseguir um efeito similar usando genéricos, mas não é um método tão elegante como o do C++, e ele não será discutido aqui.

### 12.11.3 Vinculação Dinâmica

A Ada 95 oferece tanto a vinculação estática como a vinculação dinâmica de chamadas a função para definições de função em tipos marcados. A vinculação dinâmica é forçada usando-

se tipos de classe ampla; tipos que, em certo sentido, representam todos os tipos de uma hierarquia de classes. Para um tipo marcado  $\tau$ , um tipo de classe ampla é especificado como **class** de  $\tau$ . Se quiséssemos escrever um procedimento para chamar qualquer um dos dois procedimentos EXIBE definidos em PESSOA e ESTUDANTE, o seguinte funcionaria:

```
procedure EXIBE_QUALQUER_PESSOA(P: in out PESSOA'class) is
begin
    EXIBE (P);
end EXIBE_QUALQUER_PESSOA;
```

Esse procedimento pode ser chamado com as duas chamadas seguintes:

```
with PESSOA_PKG; use PESSOA_PKG;
with ESTUDANTE_PKG.; use ESTUDANTE_PKG;
P : PESSOA;
S : ESTUDANTE;
EXIBE_QUALQUER_PESSOA(P); - chama o EXIBE em PESSOA
EXIBE_QUALQUER_PESSOA(S); - chama o EXIBE em ESTUDANTE
```

Podemos ter ponteiros polimórficos ao defini-los para que tenham o tipo de classe ampla, como em

```
type QUALQUER_PESSOA_PTR is access PESSOA'class;
```

Tipos básicos puramente abstratos podem ser definidos em Ada 95 incluindo-se a palavra reservada **abstract** nas definições de tipos e nas definições de subprograma. Além disso, as definições de subprograma não podem ter corpos. Considere este exemplo:

```
package BASE_PKG is
    type T is abstract tagged null record;
    procedure DO_IT (A : T) is abstract;
end BASE_PKG;
```

#### 12.11.4 Avaliação

Uma comparação detalhada dos recursos orientados a objeto do C++ e da Ada 95 não é possível se baseando somente na escassa descrição desses recursos aqui apresentada. Porém, algumas distinções podem ser feitas entre os dois projetos.

O C++, evidentemente, oferece uma forma melhor de herança múltipla do que a Ada 95. Porém, o uso de unidades de biblioteca-filhas para controlar o acesso às entidades da classe pai parece ser uma solução mais limpa do que as funções e classes amigas do C++. Por exemplo, se a necessidade de um amigo não for conhecida quando uma classe for definida, ela precisará ser mudada e recompilada quando essa necessidade for descoberta. Na Ada 95, novas classes em novos pacotes-filho poderão ser definidas sem perturbar o pacote pai.

O projeto de construtores e de destrutores do C++ para inicialização de objetos e para a manipulação de alocação e desalocação no monte é bom, e a Ada 95 não inclui essas capacidades.

Outra diferença entre as duas linguagens é que o projetista de uma classe-raiz C++ deve decidir se uma função-membro particular será vinculada estática ou dinamicamente. Se for feita a escolha em favor da estática, mas uma mudança posterior no sistema exigir vinculação dinâmica, a classe-raiz deverá ser mudada. Na Ada 95, essa decisão de projeto

não precisa ser tomada no projeto da classe-raiz. Cada chamada pode especificar por si mesma se ela será estática ou dinamicamente vinculada, independentemente do projeto da classe-raiz.

Uma diferença mais sutil é que a vinculação dinâmica em C++ se restringe a ponteiros e a referências a objetos, em vez de aos próprios objetos. A Ada 95 não tem essa restrição; sendo assim, nesse caso, a Ada 95 é mais ortogonal.

## **12.12 Suporte para Programação Orientada a Objeto em Eiffel**

---

A Eiffel é uma linguagem orientada a objeto pura pelo fato de ter sido projetada especificamente para suportar programação orientada a objeto e não se baseia em nenhuma linguagem existente. Além disso, subprogramas podem ser ativados somente por meio de objetos. Sua sintaxe é similar à do Pascal e da Ada. Ela tem diversas características peculiares, como, por exemplo, a inclusão de asserções. Porém, neste capítulo, discutiremos somente seu suporte para programação orientada a objeto, especificamente, a herança e a vinculação dinâmica.

### **12.12.1 Características Gerais**

Na versão original da linguagem, todos os objetos eram alocados no monte e acessados pelas referências. As versões posteriores acrescentaram um segundo tipo de objeto, chamado *objeto expandido*, alocado na pilha. Os objetos expandidos proporcionam uma maneira de se ter variáveis cujos valores são reais, em oposição a serem referências à memória; isso sem ter um segundo sistema de tipos para os tipos básicos. Assim, essas variáveis se comportam como os tipos básicos do C++ e do Java, exceto que elas são sempre inicializadas. Por exemplo, INTEGER é um tipo expandido da classe INT\_REF. Variáveis desse tipo são similares às variáveis do tipo int do C++, mas INTEGER é uma classe em todos os sentidos, ou seja, pode ter subclasses. Há três operações predefinidas para todos os objetos: *copy*, *clone* e *equal*. *copy* copia as entidades de um objeto alocado para outro. *clone* cria primeiramente o armazenamento para um objeto e depois copia as entidades de outro objeto para esse novo espaço. *equal* testa os dois objetos quanto à igualdade, considerando suas entidades, não apenas os valores de referência.

Os métodos da Eiffel são chamados **rotinas**; suas variáveis de instância são chamadas **atributos**. As rotinas e atributos de uma classe juntos são chamadas **recursos** deles.

As variáveis de referência são definidas da maneira usual, como em

```
stk1 : pilha;
```

Porém, uma segunda instrução é necessária para alocar um objeto. Isso é feito com um operador, !!, como em

```
!!stk1;
```

A criação de objetos inclui implicitamente a inicialização de atributos usando valores padrão. As rotinas construtoras de classes podem ser definidas pelo usuário para inicialização com valores não-padrão. Construtores são definidos em uma classe **creation** e expli-

citamente chamados na instrução de criação do objeto. No seguinte exemplo, `initComplexo` é um construtor:

```
class Complexo
  creation
    initComplexo
  feature
    parte_real, parte_imag : REAL
  feature
    initComplexo(r, i : REAL) is
      do
        parte_real := r;
        parte_imag := i
      end;
    ...
  
```

Agora, com o seguinte, podemos criar uma referência a um objeto `Complexo` e criar e inicializar esse objeto:

```
cl : Complexo;
!!cl.initComplexo(2.4, -3.2);
```

Como acontece com o Java, não há nenhuma operação de desalocação explícita na Eiffel. Quando um objeto não mais pode ser referenciado, ele é implicitamente desalocado e um processo de coleta de lixo reclama, por fim, seu armazenamento.

### 12.12.2 Herança

O pai de uma classe é especificado com a cláusula `inherit`, como em

```
class quadrado
  inherit retangulo
  ...

```

Uma definição de classe pode incluir uma ou mais cláusulas `feature`. Uma cláusula `feature` sem nenhum qualificador é visível tanto a subclasses como a clientes. Se o qualificador `{none}` for anexado à palavra reservada `feature`, os recursos que ela define são ocultos tanto das subclasses como dos clientes. Se o nome da classe for usado como um qualificador, os recursos são ocultos dos clientes, mas estarão visíveis às subclasses. O segmento de código seguinte ilustra esses três níveis de visibilidade:

```
class filho
  inherit pai
  feature
    - Os recursos aqui definidos são visíveis aos clientes e
    - subclasses
  feature {filho}
    - Os recursos aqui definidos são ocultos dos clientes, mas
    - visíveis às subclasses
  feature {none}
    - Os recursos aqui definidos são ocultos tanto dos clientes como
    - das subclasses
```

```
...
end;
```

Os recursos herdados podem ser ocultados dentro da subclasse usando-se **undefine**. Isso, evidentemente, impede a subclasse de ser um subtipo. Para controlar o acesso de clientes a recursos herdados, uma cláusula **export** pode ser usada.

As classes abstratas são definidas adicionando-se a palavra reservada **deferred** no início da definição de classe, como em

```
deferred class figura
```

Essa classe inclui um ou mais recursos abstratos. Qualquer subclasse de uma classe abstrata que não seja abstrata em si deve incluir, é claro, definições dos recursos abstratos da classe-pai.

A Eiffel suporta herança múltipla, especificada simplesmente tendo-se mais de uma cláusula **inherit**.

### 12.12.3 Vinculação Dinâmica

Todas as vinculações de mensagens a métodos na Eiffel são dinâmicas. As rotinas de subclasse podem sobrepor-se a rotinas herdadas. Para ser uma rotina que se sobreponha, os tipos dos parâmetros formais devem ter compatibilidade de atribuição com os da rotina sobreposta. Além disso, o tipo de retorno da rotina que sobreponha deve ter compatibilidade de atribuição com o da rotina sobreposta. Todos os recursos de sobreposição devem ser definidos na cláusula **redefine**.

O acesso a recursos sobrepostos pode ser mantido colocando-se seus nomes em uma cláusula **rename**.

### 12.12.4 Avaliação

O suporte da Eiffel para programação orientada a objeto é similar ao do Java. Em nenhum dos casos é suportada a programação baseada em procedimentos e, em ambos os casos, quase todas as vinculações de mensagens a métodos são dinâmicas. Contudo, a elegância e o projeto limpo das classes e da herança da Eiffel estão em segundo lugar somente em relação aos da Smalltalk.

---

## 12.13 O modelo de objetos da JavaScript

---

Embora a JavaScript não seja uma linguagem de programação orientada a objeto, ela usa um modelo de objetos que é levemente baseado nos objetos do C++ e do Java. O projeto da JavaScript é, portanto, uma alternativa interessante aos conceitos tradicionais de suporte de linguagem para objetos.

Hidden page

### 12.13.3 Criação e modificação de objetos

Os objetos são criados com uma expressão `new` a qual deve incluir uma chamada a um método construtor. Este cria as propriedades que caracterizam o novo objeto. Em uma linguagem orientada a objeto o operador `new` cria um objeto particular, isto é, um objeto de um tipo e tendo uma coleção específica de membros. Construtores nestas linguagens inicializam os membros sem criá-los. Na JavaScript o operador `new` cria um objeto em branco, sem propriedades. O construtor tanto cria como inicializa as propriedades.

A instrução seguinte cria um objeto usando o construtor para o objeto predefinido `Object`, o qual não cria as propriedades:

```
var meu_objeto = new Object();
```

A variável `meu_objeto` referencia o novo objeto. Os construtores serão discutidos abaixo.

As propriedades de um objeto são acessadas usando a notação de ponto, na qual a primeira palavra é o nome do objeto e a segunda é o nome da propriedade. As propriedades não são como variáveis de fato. Elas são as chaves para o *hash*: assim como acontece com as chaves *hash* da Perl, elas não possuem valor intrínseco. Elas são usadas como variáveis do objeto para acessar os valores das propriedades. Uma vez que elas não são variáveis, as propriedades jamais são declaradas.

Como foi dito acima, em qualquer momento durante a interpretação as propriedades podem ser adicionadas ou eliminadas em um objeto. Uma propriedade de um objeto é criada ao se atribuir valor a ela. Considere o seguinte exemplo:

```
var meu_carro = new Object(); //cria um objeto em branco
meu_carro.marca = "Ford"; //cria e inicializa a propriedade
                           marca
meu_carro.modelo = "Fiesta"; //cria e inicializa a propriedade
                             modelo
```

Esse código cria um novo objeto, `meu_carro` com duas propriedades: `marca` e `modelo`. Uma vez que objetos podem ser aninhados, podemos criar um novo objeto, que é uma propriedade de `meu_carro`, com suas propriedades particulares. Por exemplo:

```
meu_carro.motor = new Object();
meu_carro.motor.config = "V6";
meu_carro.motor.hp = 200;
```

Ao se tentar acessar uma propriedade inexistente de um objeto, o valor `undefined` é retornado. Uma propriedade pode ser eliminada com `delete`. Por exemplo:

```
delete meu_carro.modelo;
```

Os construtores da JavaScript são métodos especiais que criam e inicializam as propriedades de objetos recém-criados. Toda expressão `new` deve incluir uma chamada a um construtor. Os construtores são chamados de fato pelo operador `new` que os precede imediatamente na expressão `new`.

Evidentemente, um construtor deve poder referenciar o objeto no qual ele opera. A JavaScript possui uma variável de referência predefinida para este fim, chamada `this`. Quando

Hidden page

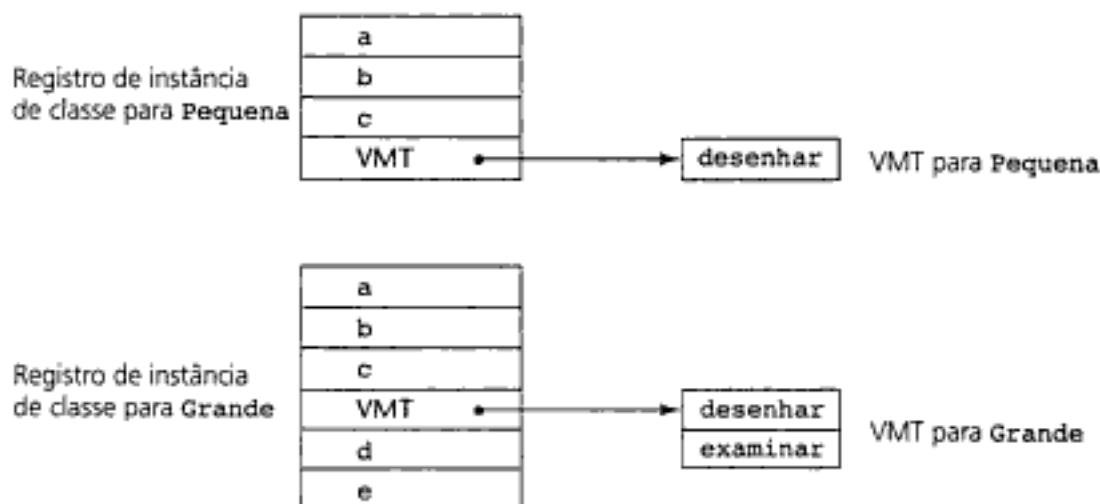
Hidden page

```

public int d, e;
public void desenhar() { ... }
public void examinar() { ... }
}

```

Os RICs para as classes Pequena e Grande, juntamente com seus VMTs, são mostrados na Figura 12.5.



**FIGURA 12.5** Registros de instância de classe e tabelas de métodos virtuais (VMT).

## RESUMO

A programação orientada a objeto inclui três conceitos fundamentais: tipos de dados abstratos, herança e vinculação dinâmica. As linguagens de programação orientadas a objeto suportam o paradigma com classes, métodos, objetos e passagem de mensagens.

A discussão a respeito das linguagens de programação orientadas a objeto neste capítulo gira em torno de sete questões de projeto: exclusividade de objetos, subclasses e subtipos, herança de implementação e interface, verificação de tipo e polimorfismo, herança simples e múltipla, vinculação dinâmica e desalocação explícita ou implícita de objetos.

A Smalltalk é uma linguagem orientada a objeto pura — tudo é objeto e toda a computação é realizada pela passagem de mensagens. Métodos são construídos a partir de expressões. Uma expressão descreve um objeto, que vem a ser o valor da expressão. As estruturas de controle da Smalltalk, igual a tudo mais, são construídas usando objetos e mensagens. Não obstante elas terem uma aparência bastante convencional, sua semântica é muito diferente da semântica das estruturas correspondentes das linguagens imperativas. Na Smalltalk, todas as subclasses são subtipos. Toda verificação de tipos e vinculação de mensagens a métodos é dinâmica e toda herança é simples. A Smalltalk não tem nenhuma operação de desalocação explícita.

O C++ oferece suporte para abstração de dados, para herança e para vinculação dinâmica opcional de mensagens a métodos, além de todos os recursos convencionais do C. Isso significa que ele tem dois sistemas de tipo distintos. Embora a vinculação dinâmica da Smalltalk proporcione bem mais flexibilidade de programação do que a linguagem híbrida, C++, ela é bem menos eficiente. O C++ oferece herança múltipla e desalocação de objetos

Hidden page

Hidden page

5. Escreva a seguinte estrutura de laço **for** C em Smalltalk:

```
for (indice = 10; indice > 0; indice --)
    soma += indice;
```

6. Escreva a seguinte construção de seleção C em Smalltalk:

```
if (cont < 10)
    resposta = 1;
else
    resposta = cont = 0;
```

7. Escreva um método de instância Smalltalk que aceite quatro valores inteiros, onde os dois primeiros são o numerador e o denominador de uma fração e os dois últimos, de maneira semelhante, representam outra fração. Seu método deve retornar um objeto vetor de dois elementos que representa o numerador e o denominador do produto das duas frações dadas.

8. Compare a vinculação dinâmica da Eiffel, do C++, da Smalltalk, da Ada 95 e do Java.

9. Compare os controles de acesso de entidades de classe do Eiffel, do C++, da Smalltalk, da Ada 95 e do Java.

10. Compare a herança simples da Eiffel, do C++, da Smalltalk, da Ada 95 e do Java.

11. Compare a herança múltipla da Eiffel e do C++.

12. Compare a herança múltipla do C++ com aquela oferecida pelas interfaces em Java.

# Capítulo 13

## Concorrência



### Niklaus Wirth

Niklaus Wirth, do ETH, de Zurique, tem estado continuamente envolvido no projeto de linguagens desde a década de 1960. Ele deixou a equipe de projeto do ALGOL 68 no início da década de 60 para desenvolver o ALGOL-W. Ele também projetou a Euler e a PL/360 naquela época. Desde então, tem sido responsável pelo desenvolvimento das linguagens Pascal, Modula, Modula-2 e Oberon.

- 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** Concorrência em Ada 95
- 13.7** Linhas de Execução Paralela Java
- 13.8** Concorrência no Nível de Instrução

Este capítulo inicia-se com descrições dos vários tipos de concorrência no nível de subprograma ou de unidade, e no de instrução. Nesta introdução, está incluída uma breve descrição de alguns tipos comuns de arquiteturas de computador com multiprocessador. Em seguida, apresentamos uma extensa discussão da concorrência no nível de unidade. Iniciamos descrevendo os conceitos fundamentais que devem ser entendidos antes de discutirmos a concorrência no nível de unidade, incluindo a sincronização da competição e a da cooperação. Em seguida, serão descritas as questões de projeto para oferecer suporte de linguagem para a concorrência. Então, apresentaremos uma discussão detalhada, incluindo exemplos de programa das três principais abordagens ao suporte de linguagem para concorrência: semáforos, monitores e passagem de mensagens. Um exemplo de programa com pseudocódigo é usado para demonstrar como os semáforos podem ser usados. Um exemplo de programa em Concurrent Pascal é usado para ilustrar o uso de monitores; para passagem de mensagens, é usado um programa Ada. Os recursos desta que suportam a concorrência serão descritos com algum detalhe. Eles incluem tarefas, declarações `entry`, cláusulas `accept` e `select`, proteções (guards), instruções `delay` e `terminate`. Após a discussão a respeito da Ada, encontra-se uma breve introdução aos novos recursos da linguagem Ada 95, que suporta a concorrência, inclusive as unidades protegidas e a passagem assíncrona de mensagens. O último exemplo de suporte de linguagem para concorrência no nível de unidade é o do Java. A última seção do capítulo tem uma discussão sobre concorrência no nível de instrução, incluindo uma breve discussão de parte do suporte de linguagem oferecido para ela no High-Performance FORTRAN.

### 13.1 Introdução

---

A concorrência é naturalmente dividida em nível de instrução de máquina (executando duas ou mais instruções de máquina simultaneamente), nível de instrução (executando duas ou mais instruções simultaneamente), nível de unidade (executando duas ou mais unidades de subprograma simultaneamente) e em nível de programa (executando dois ou mais programas simultaneamente). Uma vez que não há nenhuma questão de linguagem nelas envolvida, não discutiremos a concorrência no nível de instrução de máquina e no nível de programa. A concorrência, tanto no nível de subprograma como no de instrução, será discutida com foco no nível de subprograma.

A execução concorrente de unidades de programa pode ocorrer fisicamente em processadores separados ou logicamente usando alguma forma de tempo fatiado<sup>1</sup> em um sistema de computador de um único processador. À primeira vista, isso pode parecer um conceito simples, mas constitui um desafio significativo ao projetista de linguagem de programação.

Métodos de controle concorrentes aumentam a flexibilidade de programação. Eles foram inventados originalmente para serem usados em problemas particulares enfrentados em sistemas operacionais, mas podem ser usados para uma variedade de outras aplicações de programação. Por exemplo, muitos sistemas são projetados para simular sistemas físicos reais, cuja boa parte consiste em múltiplos subsistemas concorrentes. Para tais aplicações, a forma mais restrita de controle de subprograma é inadequada.

<sup>1</sup>N. de T. Tempo fatiado (ou parcelado): tradução literal de time-sliced — Um intervalo fixo de tempo alocado a cada usuário ou programa em um sistema multitarefa ou de timesharing.

A concorrência em nível de instrução é muito diferente da concorrência no nível de unidade. Do ponto de vista de um projetista, a concorrência em nível de instrução é, em grande parte, uma questão de especificar como os dados devem ser distribuídos em memórias múltiplas e quais instruções podem ser executadas concurrentemente.

A intenção deste capítulo é discutir os aspectos de concorrência mais relevantes para as questões de projeto de linguagem, em vez de apresentar um estudo definitivo de todas as questões da concorrência. Isso, evidentemente, não seria apropriado a um livro sobre linguagens de programação.

### 13.1.1 Arquiteturas de Multiprocessador

Um grande número de diferentes arquiteturas de computador tem mais de um processador e pode suportar alguma forma de execução concorrente. Antes de começarmos a discutir tipos de execução concorrente de programas e de instruções, descreveremos brevemente algumas dessas arquiteturas.

Os primeiros computadores com processadores múltiplos tinham um destes para propósitos gerais e um ou mais processadores adicionais usados somente para operações de entrada e saída. Isso permitia aos computadores do final da década de 50 executar concurrentemente um programa enquanto realizava a entrada ou saída de outros programas. Uma vez que esse tipo de concorrência não requer suporte de linguagem, nós não mais a consideraremos.

No início da década de 60, havia máquinas com processadores múltiplos completos. Eles eram usados pelo despachante de tarefas do sistema operacional, que simplesmente as distribuía separadas de uma fila de tarefas em lote para os processadores também separados. Sistemas com tal estrutura suportavam concorrência em nível de programa.

Em meados da década de 60 surgiram computadores multiprocessados providos de diversos processadores parciais idênticos alimentados com certas instruções de um único fluxo de instruções. Por exemplo, algumas máquinas tinham dois ou mais multiplicadores de números reais, enquanto outras tinham duas ou mais unidades aritméticas completas. Os compiladores dessas máquinas precisavam determinar quais instruções podiam ser executadas concurrentemente e programá-las de maneira apropriada. Os sistemas com essa estrutura suportavam concorrência em nível de instrução.

Atualmente, há muitos tipos diferentes de computadores com multiprocessador, entre os quais descreveremos as duas categorias mais comuns nos dois parágrafos seguintes.

Os computadores com múltiplos processadores que executam a mesma instrução simultaneamente, cada uma em dados diferentes, são chamados de arquitetura *Single-Instruction Multiple-Data* (SIMD). Em um computador SIMD, cada processador tem sua própria memória local. Um processador controla a operação dos outros processadores. Uma vez que todos, exceto o controlador, executam a mesma instrução ao mesmo tempo, nenhuma sincronização é necessária no software. Talvez as máquinas SIMD mais populares pertençam à categoria de processadores vetoriais. Eles têm grupos de registradores que armazenam os operandos de uma operação vetorial em que a mesma instrução é executada em todo o grupo de operandos simultaneamente. Os tipos de programas que mais podem se beneficiar dessa arquitetura são comuns na computação científica, uma área da computação freqüentemente alvo das máquinas de multiprocessador.

Computadores que possuem múltiplos processadores operando independentemente, mas cujas operações podem ser sincronizadas, são chamados de *Multiple-Instruction Multiple-Data* (MIMD). Cada processador em um computador MIMD executa seu próprio fluxo de instruções. Eles podem aparecer em duas configurações distintas: sistemas distri-

buidos e de memória compartilhada. As máquinas MIMD distribuídas, nas quais cada processador tem sua própria memória, podem ser construídas em um único gabinete ou distribuídas por uma grande área. Suas versões de memória compartilhada evidentemente devem oferecer algum meio de sincronização para impedir conflitos de acesso à memória. Até mesmo as máquinas MIMD distribuídas exigem que a sincronização opere em conjunto em programas únicos. Os computadores MIMD são mais caros e mais gerais do que os computadores SIMD e logicamente suportam concorrência no nível de unidade.

### 13.1.2 Categorias de Concorrência

Há duas categorias distintas de controle concorrente de unidade. A categoria mais geral de concorrência é aquela em que diversas unidades do mesmo programa literalmente executam simultaneamente, supondo que mais de um processador esteja disponível. Isso é **concorrência física**. Um leve afrouxamento desse conceito de concorrência permite que o programador e o programa suponham a existência de múltiplos processadores fornecendo concorrência real quando, de fato, a execução real dos programas está se desenvolvendo intercaladamente em um único processador. Isso é **concorrência lógica**. Ela é semelhante à ilusão de execução simultânea oferecida a diferentes usuários de um sistema de computador com multiprogramação. Do ponto de vista do programador e do projetista da linguagem, a concorrência lógica é o mesmo que a física. Cabe ao implementador da linguagem fazer a correspondência da concorrência lógica com o hardware subjacente. Tanto uma como a outra permitem que o conceito de concorrência seja usado como uma metodologia de projeto de programas. No restante deste capítulo, estaremos referindo-nos à concorrência lógica quando usarmos o termo concorrência sem qualificação.

Uma técnica útil para visualizar o fluxo de execução de um programa é imaginar uma linha (*thread*) disposta nas instruções do texto-fonte do programa. Cada instrução atingida em uma execução particular é coberta pela linha que representa essa execução. Seguir visualmente a linha ao longo do programa-fonte possibilitará rastrear o fluxo de execução pela versão executável do programa. Uma **linha de controle** em um programa é a sequência de pontos do programa atingidos à medida que o controle flui por ele.

Os programas com co-rotinas (veja o Capítulo 9), não obstante, às vezes, serem chamados quase concorrentes, têm uma única linha de controle. Os programas executados com concorrência física podem ter múltiplas linhas de controle. Cada processador pode executar uma das linhas. Ainda que a execução de programas logicamente concorrentes possa ter de fato somente uma única linha de controle, eles somente podem ser projetados e analisados imaginando a existência de múltiplas linhas de controle. Quando um programa multilinhas é executado em uma máquina com um único processador, suas linhas são mapeadas a uma única linha. Ele se torna, nesse caso, virtualmente um programa multilinhas.

A concorrência em nível de instrução é um conceito relativamente simples. Os laços que incluem comandos operando sobre elementos de um vetor são desenrolados de maneira que o processamento possa ser distribuído em múltiplos processadores. Por exemplo, um laço que executa 500 repetições e inclui uma instrução que opera sobre um dos 500 elementos de um vetor pode ser desenrolado de modo que cada um dos 10 diferentes processadores possa processar 50 dos elementos do vetor.

Hidden page

Hidden page

Hidden page

5. Morta: uma tarefa morta não está mais ativa em qualquer sentido. Ela morre quando sua execução é concluída, ou quando é explicitamente morta pelo programa.

Uma questão importante na execução de tarefas é a seguinte: como uma tarefa pronta é escolhida para mudar para o estado “rodando” quando a tarefa atualmente em execução foi bloqueada ou sua fatia de tempo extinguiu-se? Diversos algoritmos diferentes têm sido usados para tal escolha, alguns baseados em níveis de prioridade especificáveis. O algoritmo que escolhe é implementado no scheduler.

Associado à execução concorrente de tarefas e ao uso de recursos compartilhados está o conceito de vivência (*liveness*). No ambiente dos programas seqüenciais, um programa tem a característica de **vivência** se continuar a executar, levando, por fim, à conclusão. Em termos gerais, vivência significa supormos que algum evento — digamos, a conclusão de um programa — ocorrerá, ou seja, ela progride continuamente. No ambiente da concorrência e do uso de recursos compartilhados, a vivência de uma tarefa pode deixar de existir, significando que o programa não pode prosseguir e, assim, jamais finalizará.

Por exemplo, suponhamos que as tarefas A e B precisem dos recursos compartilhados x e y para concluir seu trabalho. Suponhamos ainda que a A ganhe a posse de x, e a B, de y. Depois de alguma execução, a tarefa A precisa do recurso y para prosseguir, de modo que solicita y, mas deve esperar até que B o libere. Similarmente, a tarefa B solicita x, mas deve esperar que A o libere. Nenhuma delas renuncia ao recurso que possui e, como resultado, ambas perdem sua vivência, garantindo que a execução do programa nunca se completará normalmente. Esse tipo particular de perda de vivência é chamada **enlace mortal** (*dead-lock*), que é uma séria ameaça à confiabilidade de um programa e, portanto, sua ausência exige séria consideração no projeto tanto da linguagem quanto do programa.

Agora estamos prontos para discutir alguns dos mecanismos lingüísticos para oferecer controle de unidades concorrentes.

### 13.2.2 Projeto de Linguagem para Concorrência

Uma série de linguagens foi projetada para suportar concorrência, iniciando-se com a PL/I em meados da década de 60 e incluindo as contemporâneas, Ada 95 e Java.

### 13.2.3 Questões de Projeto

As questões de projeto mais importantes de suporte de linguagem para concorrência já foram extensamente discutidas: sincronização de competição e de cooperação.

Além dessas, há diversas questões de projeto de importância secundária. Destaca-se, entre elas, como oferecer planejamento de tarefas. Há também as questões de como e quando as tarefas começam e finalizam sua execução, e como e quando elas são criadas. Porém, em nome da simplicidade, nossa discussão a respeito da concorrência é intencionalmente incompleta, e somente as questões de projeto listadas acima são discutidas nesta seção.

As seções seguintes discutem três respostas alternativas para as questões de projeto relativas à concorrência: semáforos, monitores e passagem de mensagens.

## 13.3 Semáforos

Um semáforo é um mecanismo muito simples que pode ser usado para oferecer sincronização de tarefas. Nos parágrafos seguintes, iremos descrevê-lo e discutiremos como ele pode ser usado para tal finalidade.

### 13.3.1 Introdução

Em um esforço para oferecer sincronização de competição pelo acesso mutuamente exclusivo a estruturas de dados compartilhadas, Edsger Dijkstra idealizou os semáforos em 1965 (Dijkstra, 1968b). Eles também podem ser usados para oferecer sincronização de cooperação.

Um semáforo é uma estrutura de dados que consiste em um número inteiro e em uma fila que armazena descritores de tarefas. Estes últimos são constituídos de uma estrutura de dados que armazena todas as informações relevantes sobre o estado de execução de uma tarefa. O conceito de semáforo consiste na colocação de proteções (*guards*) em torno do código que acessa a estrutura para oferecer acesso limitado a uma estrutura de dados. Uma proteção é um dispositivo linguístico que permite a execução do código protegido somente sob uma condição especificada. Uma proteção pode ser usada para permitir que somente uma tarefa acesse uma estrutura de dados compartilhados de cada vez. Um semáforo é uma implementação de proteção. Uma parte integrante de um mecanismo de proteção é uma técnica para assegurar que todas as execuções tentadas do código protegido venham por fim a acontecer. Isso é realizado fazendo com que os pedidos de acesso que ocorrem quando não podem ser satisfeitos, sejam armazenados na fila descritora de tarefas, da qual eles terão permissão depois para sair e para executar o código protegido. Esta é a razão pela qual um semáforo deve ter tanto um contador como uma fila descritora de tarefas.

As duas únicas operações oferecidas por semáforos foram originalmente chamadas P e V por Dijkstra, em função das duas palavras holandesas, *passeren* (passar) e *vrygeren* (liberar) (Andrews e Schneider, 1983). Na discussão seguinte, nos referiremos a elas como esperar e liberar.

### 13.3.2 Sincronização de Cooperação

Ao longo de grande parte deste capítulo, usamos o exemplo de um retentor compartilhado para ilustrar as diferentes abordagens para a sincronização de cooperação e de competição. Para a sincronização de cooperação, esse retentor deve ter alguma maneira de registrar tanto o número de posições vazias como o de preenchidas nele. O componente contador de uma variável semáforo pode ser usado para essa finalidade. Uma variável-semáforo, digamos *lugaresvagos*, pode ser usada para armazenar o número de localizações vazias em

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

procedimentos do monitor, inicia-se com a instrução `init` e encerra-se com o programa. O escopo das mesmas restringe-se ao próprio monitor. Os procedimentos exportados deste podem ser chamados por processos ou por procedimentos em outros monitores.

### 13.4.2 Sincronização de Competição

Um dos recursos mais importantes dos monitores é que os dados compartilhados residem neles em vez de em qualquer uma das unidades-clientes. Assim, o programador não sincroniza acessos mutuamente exclusivos a dados compartilhados pelo uso de semáforos ou de outros mecanismos. Uma vez que todos os acessos residem no monitor, a sua implementação pode ser feita de modo a garantir um acesso sincronizado, simplesmente permitindo apenas um acesso de cada vez. As chamadas a procedimentos do monitor são enfileiradas implicitamente se o monitor estiver ocupado no momento da chamada.

### 13.4.3 Sincronização de Cooperação

Não obstante o acesso mutuamente exclusivo a dados compartilhados ser intrínseco a um monitor, a cooperação entre processos ainda cabe ao programador. Em particular, o programador deve garantir que o retentor compartilhado não experimente escassez ou estouro. Para esse propósito, o Concurrent Pascal tem um tipo de dados especial, `queue`, e duas operações nele, `delay` e `continue`. O `queue` é uma forma de semáforo, e as duas operações relacionam-se às operações de semáforo enviar (`send`) e de liberar (`release`). Uma variável `queue` armazena processos que estão esperando para usar uma estrutura de dados compartilhada.

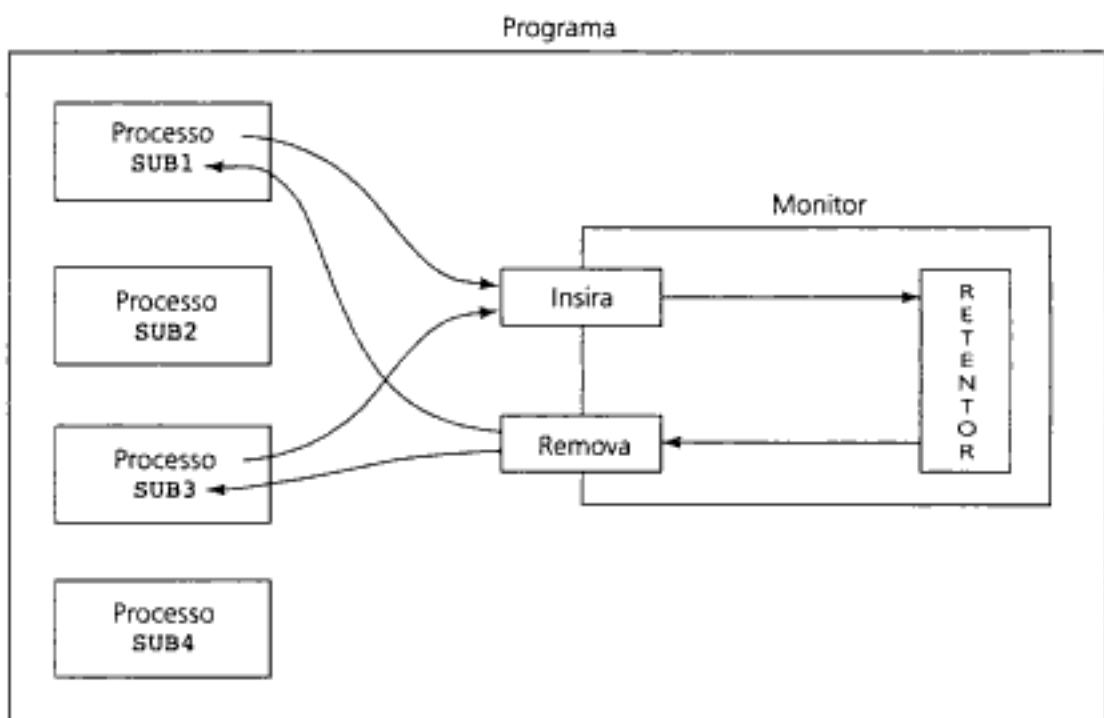
A operação `delay` toma `queue` como parâmetro. Essa ação serve para colocar o processo que a chama na fila especificada e retirar seus direitos de acesso exclusivo a estruturas de dados do monitor. Assim, o processo que executa `delay` tem sua execução suspensa. O monitor está, então, disponível a outros processos. Desse modo, `delay` difere da operação de semáforo esperar porque `delay` sempre bloqueia o chamador.

A operação `continue` também toma um parâmetro `queue`. Sua ação é desconectar o processo que a chama do monitor, liberando-o, assim, de ser acessado por outros processos; `continue` examina, então, a fila especificada. Se esta contiver um processo, ele será removido, e sua execução, suspensa por uma operação `delay`, será reiniciada. A `continue` difere da operação liberar para semáforos porque liberar sempre tem algum efeito, enquanto que `continue` nada faz se a fila estiver vazia.

Um programa que contém quatro processos e um monitor que fornece acesso sincronizado a um retentor compartilhado de maneira concorrente é mostrado na Figura 13.2 (ver p. 498).

Usando o tipo de dados `queue` e as operações `delay` e `continue`, pode-se construir um monitor que controla um retentor compartilhado, fornecendo, assim, tanto sincronização de competição como de cooperação. No exemplo seguinte, um retentor compartilhado é implementado como uma lista logicamente circular de 100 números inteiros.

```
type retentordados =
  monitor
  const tamret = 100;
  var ret : array [1..tamret] of integer;
    proximo_dentro,
```



**FIGURA 13.2** Um programa que usa um monitor para controlar o acesso a um retentor compartilhado.

```

    proximo_fora : 1..tamret;
    preenchido : 0.. tamret;
    fila_envia,
    fila_recebe : queue;

procedure entry deposita(item : integer);
begin
  if preenchido = tamret
    then delay(fila_envia);
  ret[proximo_dentro] := item;
  proximo_dentro := (proximo_dentro mod tamret) + 1;
  preenchido := preenchido + 1;
  continue(fila_recebe)
end;

procedure entry busca (var item : integer);
begin
  if preenchido = 0
    then delay(fila_recebe);
  item := ret[proximo_fora];
  proximo_fora := (proximo_fora mod tamret) + 1;
  preenchido := preenchido - 1;
  continue(fila_envia)
end;

```

Hidden page

### 13.4.4 Avaliação

Os monitores são uma maneira melhor de fornecer sincronização de competição do que os semáforos, principalmente devido aos problemas destes últimos, conforme discutimos na Seção 13.3. O uso de variáveis do tipo `queue` para fornecer sincronização de cooperação por `delay` e `continue` está sujeito, entretanto, aos mesmos problemas que os semáforos usados em outras linguagens para esse propósito.

## 13.5 Passagem de Mensagens

A construção monitor é um método confiável e seguro para fornecer sincronização de competição para acesso a dados compartilhados em unidades concorrentes que compartilham uma única memória. Entretanto, considere o problema de sincronizar as unidades em um sistema distribuído, no qual cada processador tem sua própria memória, em vez de uma única compartilhada. Evidentemente, a construção monitor não modela essa situação naturalmente. Porém, a sincronização em um sistema distribuído pode ser obtida muito naturalmente com a passagem de mensagens.

### 13.5.1 Introdução

Os primeiros esforços para projetar linguagens que oferecem a capacidade de passar mensagens entre tarefas concorrentes foram feitos por Brinch Hansen (1978) e Hoare (1978). Os desenvolvedores pioneiros da passagem de mensagens também produziram uma técnica para manipular o problema de o que fazer quando múltiplos pedidos simultâneos são feitos por outras tarefas para se comunicar com uma em especial. Decidiu-se que alguma forma de não-determinismo seria necessária para assegurar um tipo de justiça na escolha de qual desses pedidos seria satisfeito primeiro. Essa justiça pode ser definida de várias maneiras, mas, em geral, significa que todos os solicitantes recebem uma oportunidade igual de comunicarem-se com determinada tarefa. As construções não-determinísticas para controle em nível de instrução ou comandos protegidos foram introduzidas por Dijkstra (1975). Eles foram discutidos no Capítulo 8 e são a base da construção projetada para controlar a passagem de mensagens.

### 13.5.2 O Conceito de Passagem de Mensagem Síncrona

A passagem de mensagens pode ser síncrona ou assíncrona. A passagem de mensagem assíncrona da Ada 95 será descrita na Seção 13.6. Descreveremos aqui a síncrona. O conceito básico desta é o de que as tarefas estão freqüentemente ocupadas e, assim sendo, não estão expostas a interrupções por outras unidades. Suponhamos que as tarefas **A** e **B** estejam em execução, e que **A** deseje enviar uma mensagem a **B**. Evidentemente, se **B** estiver ocupada, não será desejável permitir que outra tarefa interrompa-a. Isso causaria uma ruptura no processamento atual dela. Além disso, as mensagens normalmente causam um processamento associado no receptor, que pode não ser sensato se outro processamento estiver incompleto. A alternativa é fornecer um mecanismo lingüístico que permita a uma tarefa especificar a outras tarefas quando estará pronta para receber mensagens. Isso é bastante

semelhante a um executivo instruir sua secretária a segurar todas as chamadas telefônicas recebidas até que outra atividade, talvez uma reunião importante, esteja terminada. Mais tarde, o executivo dirá à secretária que agora está disponível para receber um dos telefonemas da espera.

Uma tarefa pode ser idealizada de modo a poder suspender sua execução em certo ponto, porque está ociosa ou porque precisa de informações de outra unidade antes que possa prosseguir. Isso é semelhante a uma pessoa que aguarda um telefonema importante. Em alguns casos, não há outra coisa a fazer a não ser sentar e esperar. Nessa situação, se a tarefa A quiser enviar uma mensagem a B, e se esta estiver disposta a receber, a mensagem poderá ser transmitida. Essa transmissão real é chamada **rendezvous**. Note que um *rendezvous* pode ocorrer somente se tanto o emissor como o receptor quiserem que ele aconteça. A informação da mensagem pode ser transmitida em qualquer ou ambas as direções.

Tanto a sincronização de cooperação quanto a sincronização de competição de tarefas podem ser manipuladas convenientemente com o modelo de passagem de mensagens, conforme descreveremos nas subseções seguintes.

### 13.5.3 O Modelo de Passagem de Mensagens em Ada 83

O projeto da Ada para tarefas baseia-se parcialmente no trabalho de Brinch Hansen e Hoare cuja passagem de mensagens é a base de projeto, e usa-se o não-determinismo para escolher entre tarefas concorrentes que enviam mensagens.

O modelo completo de tarefas da Ada é complexo, e a discussão seguinte sobre ele deve ser limitada. Aqui, o foco será sobre a versão Ada do mecanismo de passagem de mensagens síncrono.

As tarefas Ada podem ser mais ativas do que os monitores. Os monitores são entidades passivas que fornecem serviços de gerenciamento para os dados compartilhados armazenados neles. Eles fornecem seus serviços, mas somente quando esses serviços são solicitados. Quando usadas para gerenciar dados compartilhados, as tarefas Ada podem ser imaginadas como gerenciadores possíveis de residir com o recurso que gerenciam. Elas têm diversos mecanismos, alguns determinísticos e outros não-determinísticos, que permitem a escolha entre solicitações concorrentes para acesso a seus recursos.

A forma das tarefas Ada é similar à dos seus pacotes. Há 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 localizações nas quais ela pode aceitar mensagens de outras tarefas. É natural que eles sejam listados na parte de especificação de uma tarefa. Uma vez que um *rendezvous* pode envolver uma troca de informações, as mensagens podem ter parâmetros; por conseguinte, os pontos de entrada das tarefas também devem permitir parâmetros, que também devem ser descritos na parte de especificação. Na aparência, uma especificação de tarefa é muito semelhante à de pacote de um tipo de dados abstrato.

Como um exemplo de especificação de tarefa Ada, considere o seguinte, que inclui um único ponto de entrada chamado **ENTRADA\_1**, que tem um parâmetro de modo de entrada (*in-mode*):

```
task TAREFA_EXEMPLO is
    entry ENTRADA_1(ITEM : in INTEGER);
end TAREFA_EXEMPLO;
```

Um corpo de tarefa deve incluir alguma forma sintática de pontos de entrada que corresponda às cláusulas **entry** na parte de especificação dessa tarefa. Na Ada, eles são especificados pelas cláusulas **accept**, introduzidas pela palavra-reservada **accept**. Uma

**cláusula accept** é definida como a faixa de instruções que se inicia com a **accept** e se encerra com a coincidente **end**. As cláusulas **accept** são relativamente simples, mas outras construções nas quais elas podem ser incorporadas podem tornar a sua semântica bastante complexa. Uma cláusula **accept** simples tem a forma

```
accept nome_da_entrada (parâmetros formais) do
  ...
end nome_da_entrada;
```

O nome **accept** coincide com o de uma cláusula **entry** na parte de especificação de tarefa associada. Os parâmetros opcionais oferecem os meios de comunicar dados entre a tarefa chamadora e a chamada. As instruções entre o **do** e o **end** definem as operações que se desenvolvem durante o *rendezvous*. Essas instruções são chamadas conjuntamente de **corpo da cláusula accept**. Durante o *rendezvous* real, a tarefa emissora é suspensa.

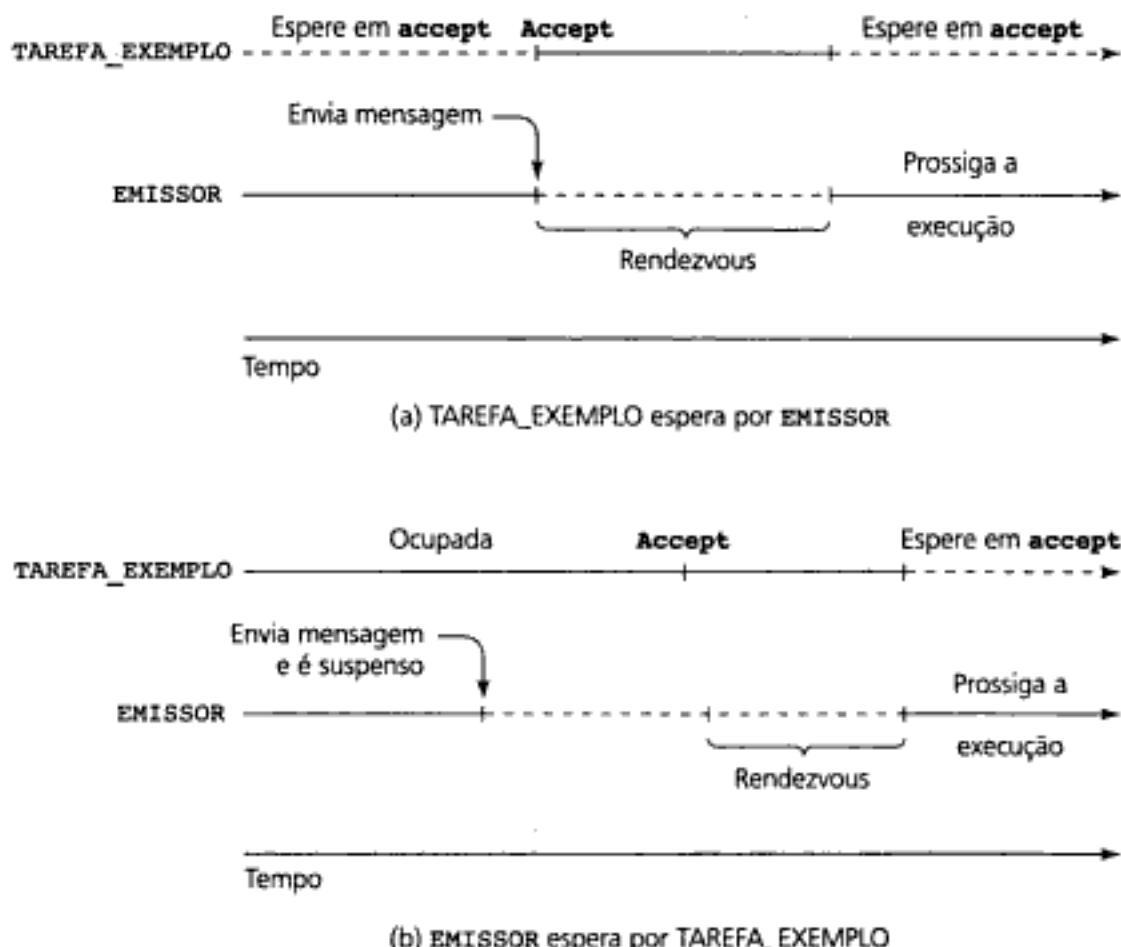
As tarefas Ada comunicam-se com outras usando o mecanismo de *rendezvous*. Quando um ponto de entrada de tarefas ou uma cláusula **accept** recebe uma mensagem que, por algum motivo, não está preparado para aceitar, a tarefa emissora deve ser suspensa até que o ponto de entrada na receptora esteja preparado para receber a mensagem. Logicamente, o ponto de entrada também deve lembrar-se das tarefas emissoras que enviaram mensagens rejeitadas. Para tal finalidade, cada cláusula **accept** de uma tarefa tem uma fila associada a ela, que registra uma lista de outras tarefas que tentaram se comunicar com o ponto de entrada associado.

O que apresentamos a seguir é o corpo esquemático da tarefa cuja especificação foi apresentada acima:

```
task body TAREFA_EXEMPLO is
begin
loop
  accept ENTRADA_1(ITEM : in INTEGER) do
    ...
  end ENTRADA_1;
end loop;
end TAREFA_EXEMPLO;
```

A cláusula **accept** desse corpo de tarefa é a implementação da **entry** chamada **ENTRADA\_1** na especificação de tarefa. Se a execução de **TAREFA\_EXEMPLO** iniciar-se e atingir **accept** de **ENTRADA\_1** antes que qualquer outra tarefa envie uma mensagem a este último, **TAREFA\_EXEMPLO** será suspensa. Se outra tarefa enviar uma mensagem a **ENTRADA\_1** enquanto **TAREFA\_EXEMPLO** estiver suspensa em **accept**, ocorrerá um *rendezvous*, e o corpo de **accept** será executado. Então, por causa do laço, a execução prosseguirá até **accept** novamente. Se nenhuma tarefa de chamada adicional tiver enviado uma mensagem a **ENTRADA\_1**, a execução será novamente suspensa para esperar pela mensagem seguinte.

Um *rendezvous* pode ocorrer de duas maneiras básicas neste exemplo simples. Primeiro, a tarefa receptora, **TAREFA\_EXEMPLO**, pode estar esperando que outra tarefa envie uma mensagem à **ENTRADA\_1**. Quando a mensagem é enviada, ocorre o *rendezvous*. Esta é a situação descrita acima. Segundo, a tarefa receptora pode estar ocupada com um *rendezvous* ou com algum outro processamento não-associado com um *rendezvous*, quando outra tarefa tenta enviar uma mensagem à mesma entrada. Nesse caso, o emissor é suspenso até que o receptor esteja livre para aceitar essa mensagem em um *rendezvous*. Se diversas mensagens chegarem enquanto o receptor estiver ocupado, os emissores serão enfileirados para esperar sua vez em um *rendezvous*.



**FIGURA 13.3** Duas maneiras segundo as quais pode ocorrer um *rendezvous* com TAREFA\_EXEMPLO.

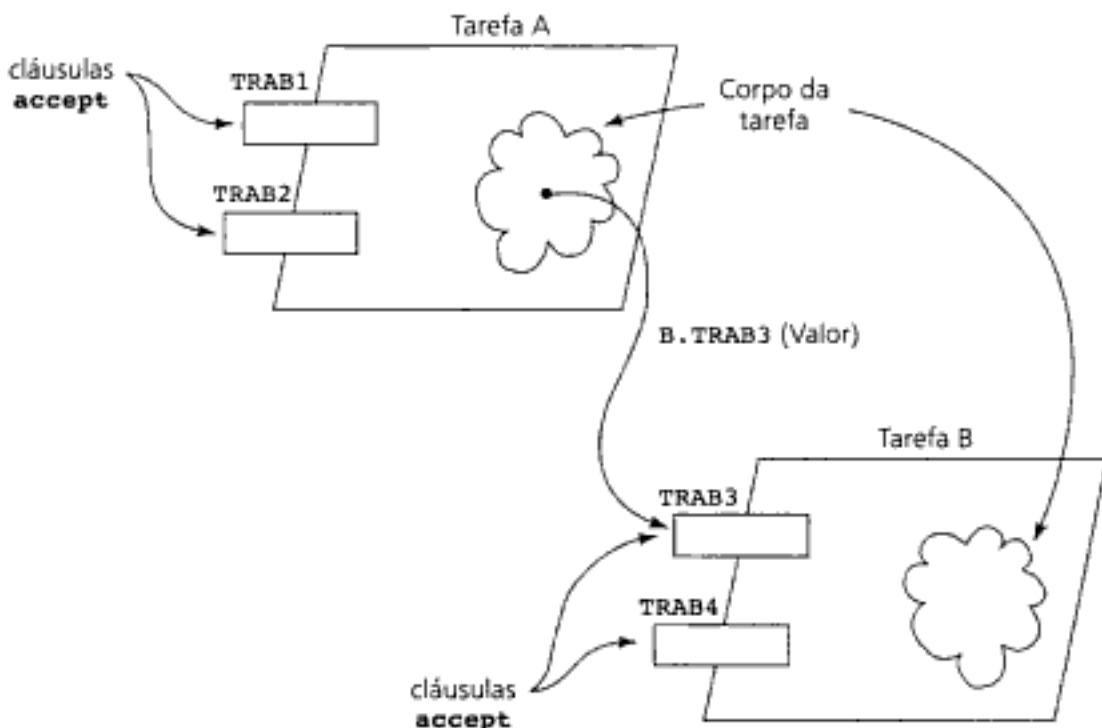
Os dois *rendezvous* que acabamos de descrever são ilustrados com os diagramas de linha de tempo da Figura 13.3.

As tarefas não precisam de pontos de entrada. Elas são chamadas de **tarefas atuantes** porque não esperam por um *rendezvous* para fazer um trabalho útil. As tarefas atuantes podem fazer *rendezvous* com outras tarefas, enviando mensagens a elas. Em contraste com as tarefas atuantes, uma tarefa pode ter pontos de entrada, mas poucos ou nenhum outro código a não ser um associado com mensagens de aceitação, de modo que ela somente pode reagir a outras tarefas. Ela é chamada de **tarefa servidora**.

Uma tarefa Ada que envia uma mensagem a outra tarefa deve saber o nome da entrada nessa tarefa. Porém, o oposto não é verdadeiro: uma entrada de tarefa não precisa saber o nome da tarefa da qual ela aceitará mensagens. Essa assimetria contrapõe-se ao projeto da linguagem conhecida como CSP (Communicating Sequential Processes) (Hoare, 1978). Na CSP, que também usa o modelo de passagem de mensagens para a concorrência, as tarefas aceitam mensagens somente de tarefas explicitamente nomeadas. A desvantagem disso é que bibliotecas de tarefas não podem ser construídas para uso geral.

O método gráfico usual para descrever um *rendezvous* no qual a tarefa A envia uma mensagem à tarefa B é mostrado na Figura 13.4 (ver p. 504).

As tarefas Ada são tipos e, como tal, podem ser anônimas ou nomeadas. Uma tarefa Ada com um tipo nomeado pode ser criada dinamicamente usando-se o operador **new** e referenciada por um ponteiro. Por exemplo, considere o seguinte:



**FIGURA 13.4** Representação gráfica de um *rendezvous* causado por uma mensagem enviada da tarefa A para a tarefa B.

```

task type RETENTOR is
    entry DEPOSITA(VALOR: in INTEGER);
    entry BUSCA(VALOR : out INTEGER);
end;
type PTR_RET is access RETENTOR;
...
RET : PTR_RET
RET := new RETENTOR;
  
```

As tarefas são declaradas na parte de declaração de um pacote, de um subprograma ou de um bloco. Elas iniciam sua execução ao mesmo tempo que as instruções do código cuja parte declarativa é anexada. Por exemplo, uma tarefa declarada em um programa principal inicia a execução ao mesmo tempo que a primeira instrução no corpo de código do programa principal é executada. As tarefas criadas com `new` iniciam sua execução imediatamente. A finalização de tarefas, uma questão complexa, será discutida posteriormente nesta seção.

As tarefas podem ter qualquer número de entradas. A ordem em que as cláusulas `accept` associadas aparecem nelas determina a ordem em que as mensagens podem ser aceitas. Se for uma tarefa que tem mais de um ponto de entrada e exige que elas sejam capazes de receber mensagens em qualquer ordem, ela usará uma instrução `select` para conter as entradas, como no seguinte:

```

task body TAREFA_EXEMPLO is
    loop
        select
            accept ENTRADA_1(parâmetros formais) do
  
```

```

    ...
    end ENTRADA_1;
    ...
or
    accept ENTRADA_2(parâmetros formais) do
    ...
end ENTRADA_2;
...
end select;
end loop;
end TAREFA_EXEMPLO;

```

Nessa tarefa, há duas cláusulas **accept**, cada uma com uma fila associada. A ação da **select**, quando ela é executada, é examinar as filas associadas com as duas **accept**. Se uma das filas estiver vazia, mas a outra contiver pelo menos uma mensagem de espera, a **accept** associada com a mensagem de espera terá um *rendezvous* com a tarefa que enviou a primeira mensagem recebida. Se ambas as **accept** tiverem filas vazias, a **select** esperará até que uma das entradas seja chamada. Se ambas as **accept** tiverem filas não-vazias, uma delas será não-deterministicamente escolhida para ter um *rendezvous* com uma de suas chamadoras. O laço forçará a instrução **select** a ser executada, repetidamente, para sempre.

O **end** de uma cláusula **accept** marca o final do código que atribui ou referencia os parâmetros formais dela. O código, se houver um, entre a **accept** e o **or** seguinte (ou o **end select** se a cláusula **accept** for a última) é chamado **cláusula accept estendida**, que é executada somente depois que a **accept** (imediatamente precedente) for. Tal execução da **accept** estendida não faz parte do *rendezvous* e pode acontecer em paralelo com a tarefa chamadora. O emissor é suspenso durante o *rendezvous*, mas reiniciado (posto de volta na fila pronta) quando o final da **accept** é atingido. Se esta última não tiver nenhum parâmetro formal, o **do-end** não será exigido, e a **accept** poderá consistir inteiramente em uma cláusula **accept** estendida, usada exclusivamente para sincronização.

### 13.5.4 Sincronização de Cooperação

Cada cláusula **accept** pode ter uma proteção (*guard*) anexada, na forma de uma cláusula **when**, que pode retardar o *rendezvous*. Por exemplo,

```

when not CHEIO(RETENTOR) =>
    accept DEPOSITA(NOVO_VALOR) do

```

Uma **accept** com uma **when** pode ser aberta ou fechada. Se a expressão booleana da **when** for atualmente verdadeira, **accept** será **aberta**; se a expressão booleana for falsa, a **accept** será **fechada**. Uma **accept** sem proteção é sempre aberta e está disponível para *rendezvous*; uma fechada não pode ter *rendezvous*.

Suponhamos que haja diversas cláusulas **accept** protegidas em uma **select**. Essa última usualmente é colocada em um laço infinito. Ele faz com que a **select** seja executada repetidamente, com cada **when** avaliada em todas repetições. Isso faz com que uma lista de cláusulas **accept** abertas seja construída. Se exatamente uma destas tiver uma fila não-vazia, uma mensagem será tomada dela e ocorrerá um *rendezvous*. Se mais de uma das cláusulas **accept** abertas tiver filas não-vazias, uma fila será escolhida de maneira não-determinística, uma mensagem será tirada dela e um *rendezvous* irá desenvolver-se. Se as

filas de todas as cláusulas abertas estiverem vazias, a tarefa esperará que uma mensagem chegue a uma dessas cláusulas **accept**, em cujo tempo ocorrerá um *rendezvous*. Se uma **select** for executada e toda **accept** estiver fechada, o resultado será uma exceção ou erro em tempo de execução. Essa possibilidade pode ser evitada ao certificar-se de que uma das cláusulas **when** seja sempre verdadeira ou adicionando uma cláusula **else** na **select**. A **else** pode incluir qualquer seqüência de instruções, exceto uma **accept**.

Uma cláusula **select** pode ter uma instrução especial, **terminate**, selecionada somente quando ela é aberta e nenhuma outra cláusula **accept** está aberta. Uma **terminate**, quando selecionada, significa que a tarefa concluiu seu trabalho, mas ainda não se encerrou. A finalização de tarefas será discutida posteriormente nesta seção.

### 13.5.5 Sincronização de Competição

Os recursos descritos até aqui se prestam para a sincronização de cooperação e de comunicação entre tarefas. A seguir, discutiremos como o acesso mutuamente exclusivo a estruturas de dados compartilhadas pode ser imposto.

Se o acesso a uma estrutura de dados precisar ser controlado por uma tarefa, tal acesso mutuamente exclusivo poderá ser obtido declarando-se a estrutura de dados dentro de uma tarefa. A semântica de execução de tarefas normalmente assegura o acesso mutuamente exclusivo à estrutura, porque somente uma cláusula **accept** na tarefa pode estar ativa em determinado momento. A única exceção a isso ocorre quando tarefas são aninhadas em procedimentos ou em outras tarefas. Por exemplo, se uma tarefa que define uma estrutura de dados compartilhada tiver uma tarefa aninhada, esta também poderá acessar a estrutura compartilhada, o que poderia destruir a integridade dos dados. Dessa forma, tarefas que pretendem controlar o acesso a uma estrutura de dados compartilhada não devem definir tarefas.

Apresentamos, a seguir, um exemplo de tarefa Ada para fornecer acesso sincronizado a um retentor. Com efeito, ele é muito semelhante ao nosso exemplo de monitor.

```

task TAREFA_RET is
    entry DEPOSITA(ITEM : in INTEGER);
    entry BUSCA(ITEM : out INTEGER);
end TAREFA_RET;
task body TAREFA_RET is
    TAMRET: constant INTEGER := 100;
    RET : array (1..TAMRET) of INTEGER;
    PREENCHIDO : INTEGER range 0..TAMRET := 0;
    PROXIMO_DENTRO,
    PROXIMO_FORA : INTEGER range 1..TAMRET := 1;
begin
loop
    select
        when PREENCHIDO < TAMRET =>
            accept DEPOSITA(ITEM : in INTEGER) do
                RET(PROXIMO_DENTRO) := ITEM;
            end DEPOSITA;
            PROXIMO_DENTRO := (PROXIMO_DENTRO mod TAMRET) + 1;
            PREENCHIDO := PREENCHIDO + 1;
        or

```

```

when PREENCHIDO > 0 =>
    accept BUSCA(ITEM : out INTEGER) do
        ITEM := RET(PROXIMO_FORA);
    end BUSCA;
    PROXIMO_FORA := (PROXIMO_FORA mod TAMRET) + 1;
    PREENCHIDO := PREENCHIDO - 1;
end select;
end loop;
end TAREFA_RET;

```

Neste exemplo, ambas as cláusulas `accept` são estendidas. Elas permitem execução concorrente de `TAREFA_RET` com as tarefas chamadoras.

As tarefas para o produtor e consumidor que poderiam usar `TAREFA_RET` têm a seguinte forma:

```

task PRODUTOR;
task CONSUMIDOR;
task body PRODUTOR is
    NOVO_VALOR : INTEGER;
begin
    loop
        -- produza NOVO_VALOR --
        TAREFA_RET.DEPOSITA(NOVO_VALOR);
    end loop;
end PRODUTOR;

task body CONSUMIDOR is
    VALOR_ARMAZENADO : INTEGER;
begin
    loop
        TAREFA_RET.BUSCA(VALOR_ARMAZENADO);
        -- consuma VALOR_ARMAZENADO --
    end loop;
end CONSUMIDOR;

```

### 13.5.6 Finalização de Tarefas

Agora, encaminharemos a questão da finalização de tarefas. Primeiro, devemos definir o que é conclusão de tarefas. A execução de uma tarefa é **concluída** se o controle tiver alcançado o final de seu corpo de código. Isso pode ocorrer porque é levantada uma exceção para a qual não há nenhum manipulador (a manipulação de exceções em Ada será descrita no Capítulo 14). Se uma tarefa não tiver criado quaisquer outras tarefas, chamadas dependentes, ela será finalizada quando sua execução for concluída. Uma tarefa que criou tarefas dependentes é finalizada quando a execução de seu código for concluída e todas as suas dependentes finalizadas. Uma tarefa pode encerrar sua execução esperando em uma cláusula `terminate` aberta. Nesse caso, a tarefa é finalizada somente quando seu mestre (o bloco, o subprograma ou a tarefa que a criou) e todas estas que dependem desse mestre tiverem sido finalizadas ou estiverem esperando em uma cláusula `terminate` aberta. Nesse caso, estas tarefas são finalizadas simultaneamente. Um bloco ou um subprograma não é encerrado até que todas as suas tarefas dependentes sejam finalizadas.

### 13.5.7 Prioridades

Podem ser atribuídas prioridades tanto aos tipos nomeados quanto aos anônimos. Isso é feito com um *pragma*<sup>1</sup>, como em

```
pragma priority(expressão);
```

O valor da expressão especifica a prioridade relativa para a tarefa ou para a definição de seu tipo na qual ele aparece. A faixa possível de valores de prioridade depende da implementação. A mais alta prioridade possível pode ser especificada com um atributo *last* do tipo *priority*, definido em *System*. *System* é um pacote predefinido. Por exemplo, o seguinte especifica a mais alta prioridade em qualquer implementação:

```
pragma priority(System.priority'last);
```

As prioridades nas tarefas Ada aplicam-se somente a tarefas no estado pronto. Elas são usadas pelo scheduler para especificar qual tarefa deverá mudar para o estado "rodando" em seguida. Se houver três tarefas esperando em uma cláusula *accept* particular e elas tiverem prioridades diferentes, essas prioridades não afetarão a que terá *rendezvous* primeiro.

### 13.5.8 Semáforos Binários

Se o acesso a uma estrutura de dados precisar ser controlado e não estiver encapsulado em uma tarefa, outros meios devem ser usados para fornecer acesso mutuamente exclusivo. Uma maneira é construir uma tarefa do tipo semáforo binário para usar com a que referencia a estrutura de dados. Essa tarefa semáforo binário poderia ser definida da seguinte maneira:

```
task SEMAFORO_BINARIO is
    entry WAIT;
    entry RELEASE;
end SEMAFORO_BINARIO;

task body SEMAFORO_BINARIO is
begin
loop
    accept WAIT;
    accept RELEASE;
end loop;
end SEMAFORO_BINARIO;
```

O propósito é garantir que as operações *WAIT* e *RELEASE* ocorram alternativamente.

A tarefa *SEMAFORO\_BINARIO* ilustra as simplificações possíveis quando mensagens Ada são passadas somente para sincronização, em vez de também passarem dados. Especificamente, note a forma simples de cláusulas *accept* que não precisam de corpos.

<sup>1</sup>N. de T. *Pragma*: (informação pragmática) uma forma padronizada de comentário que tem significado para um compilador. Ele pode usar uma sintaxe especial ou uma forma específica dentro da sintaxe normal de comentários. Um *pragma* normalmente transmite informações não-essenciais que, muitas vezes, pretendem ajudar o compilador a otimizar o programa.

O uso da tarefa `SEMAFORO_BINARIO` para fornecer acesso mutuamente exclusivo a uma estrutura de dados compartilhada deveria desenvolver-se exatamente como o uso de semáforos no exemplo de programa da Seção 13.3. Naturalmente, esse uso dos semáforos padece de todos os problemas potenciais lá discutidos.

À semelhança dos semáforos, monitores também podem ser simulados com a capacidade de tarefas da Ada, as quais proporcionam acesso mutuamente exclusivo implícito, exatamente como fazem os monitores. Desse modo, o modelo de tarefas Ada suporta tanto semáforos como monitores.

### 13.5.9 Avaliação

Na ausência de processadores distribuídos com memórias independentes, a escolha entre monitores e passagem de mensagens como meio de oferecer sincronização de competição é uma questão de gosto. A sincronização de cooperação com passagem de mensagens depende menos do uso correto do que os semáforos (os quais são exigidos com os monitores). Acima de tudo, entretanto, a passagem de mensagens é ligeiramente melhor, mesmo em um ambiente de memória compartilhada.

Para sistemas distribuídos, entretanto, a passagem de mensagens é um modelo melhor para concorrência, porque suporta naturalmente o conceito de processos separados executados paralelamente em processadores separados.

## 13.6 Concorrência em Ada 95

Uma das metas do projeto da Ada 95 era melhorar as capacidades da Ada 83 para especificar a concorrência. O uso exclusivo que a Ada 83 fazia do modelo de passagem de mensagens para controlar o acesso a dados compartilhados resulta em uma execução lenta por causa da complexidade do mecanismo de *rendezvous*. Para aliviar essa situação, a Ada 95 inclui objetos protegidos, os quais proporcionam controle de acesso mais conveniente e eficiente para dados compartilhados. A Ada 95 também inclui um método para oferecer comunicação assíncrona de tarefas. Primeiro, discutiremos os objetos protegidos.

### 13.6.1 Objetos Protegidos

Na Ada 83, o acesso a dados compartilhados é controlado contendo-se os dados em uma tarefa e permitindo-se acesso somente pelas entradas de tarefa, as quais se prestam implicitamente para sincronização de competição. Um problema com esse método é a dificuldade de implementar o mecanismo de *rendezvous* eficientemente. Os objetos protegidos Ada 95 apresentam um método alternativo para oferecer sincronização de competição que não precisa envolver *rendezvous*.

Um objeto protegido não é uma tarefa; ele se assemelha mais a um monitor. Objetos protegidos podem ser acessados ou por subprogramas protegidos ou por entradas similares àquelas existentes em tarefas. Eles podem ser procedimentos protegidos, os quais oferecem acesso mutuamente exclusivo de leitura-escrita aos dados do objeto ou de funções protegidas, que oferecem acesso concorrente somente de leitura a 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, o que permite acesso concorrente somente de leitura.

As chamadas de entrada a um objeto protegido proporcionam comunicação síncrona com uma ou mais tarefas usando o mesmo objeto protegido. Essas chamadas de entrada proporcionam acesso similar àquele oferecido aos dados contidos em uma tarefa.

O problema do retentor solucionado com uma tarefa na subseção anterior pode ser resolvido de maneira mais simples com um objeto protegido.

```

protected RETENTOR is
    entry DEPOSITA(ITEM : in INTEGER);
    entry BUSCA(ITEM : out INTEGER);
private
    TAMRET : constant INTEGER := 100;
    RET : array (1..TAMRET) of INTEGER;
    PREENCHIDO : INTEGER range 0..TAMRET := 0;
    PROXIMO_DENTRO,
    PROXIMO_FORA : INTEGER range 1..TAMRET := 1;
end RETENTOR;

protected body RETENTOR is
    accept DEPOSITA(ITEM : in INTEGER)
        when PREENCHIDO < TAMRET is
    begin
        RET(PROXIMO_DENTRO) := ITEM;
        PROXIMO_DENTRO := (PROXIMO_DENTRO mod TAMRET) + 1;
        PREENCHIDO := PREENCHIDO + 1;
    end DEPOSITA;
    accept BUSCA(ITEM : out INTEGER) when PREENCHIDO > 0 is
    begin
        ITEM := RET(PROXIMO_FORA);
        PROXIMO_FORA := (PROXIMO_FORA mod TAMRET) + 1;
        PREENCHIDO := PREENCHIDO - 1;
    end BUSCA;
end RETENTOR;

```

### 13.6.2 Mensagens Assíncronas

A outra adição significativa que a Ada 95 faz às capacidades de concorrência de sua versão 83 é a capacidade das tarefas enviarem mensagens assíncronas a outras tarefas. O rendezvous da Ada 83 é estritamente síncrono; tanto o emissor como o receptor devem estar preparados para comunicação antes de comunicarem-se de fato pelo rendezvous.

Uma tarefa Ada 95 pode ter uma cláusula **select** especial, **asynchronous select**, a qual pode reagir imediatamente a mensagens de outras tarefas. Essa cláusula pode ter qualquer uma das duas diferentes alternativas de acionamento: uma chamada de entrada (**entry**) ou uma instrução **delay**. Além da parte de acionamento, a cláusula **select** assíncrona tem uma parte “abortável”, que poderia conter qualquer sequência de instruções Ada. A semântica de uma cláusula **select** assíncrona é que ela executa apenas uma de suas duas partes. Se o evento disparador ocorrer (ou a chamada **entry** for recebida, ou a conta-

gem de tempo do **delay** finalizar), ela executará essa parte. Caso contrário, ela executará a cláusula abortável. Os dois exemplos seguintes de cláusulas **select** assíncronas aparecem no manual de referência da Ada 95 (AARM, 1995). No primeiro segmento de código, a cláusula abortável é executada repetidamente (por causa do laço) até que a chamada a **Terminal.Espero\_por\_interrupcao** seja recebida. No segundo segmento de código, a função chamada na cláusula abortável executa durante, pelo menos, cinco segundos. Se ela não for finalizada, então a **select** será encerrada.

```
-- Loop de comando principal para um interpretador de comando
loop
    select
        Terminal.Espero_por_interrupcao;
        Put_Line("Interrompido");
    then abort
        -- Isso será abandonado após a interrupção do terminal
        Put_Line(" -> ");
        Get_Line(Command, Last);
        Process_Command(Command (1..Last));
    end select;
end loop;
-- Um cálculo de tempo limitado
select
    delay 5.0;
    Put_Line("O cálculo não converge");
then abort
    -- Este cálculo deve encerrar-se em 5,0 segundos;
    -- Se não, irá presumir-se que ele diverge.
    Funcao_Recursiva_Muito_Complicada(X, Y);
end select;
```

## 13.7 Linhas de execução paralela Java

As unidades concorrentes em Java são objetos que incluem um método chamado **run**, cujo código pode estar em execução concorrente com outros desses métodos e com o método **main**. Há duas maneiras de definir uma classe cujos objetos podem ter métodos concorrentes. Uma delas é definir uma subclasse da classe predefinida **Thread**, a qual fornece suporte para o método **run**. Essa é uma técnica freqüente, mas nem sempre aceitável. Lembre-se do Capítulo 12 que o Java não suporta herança múltipla. Porém, uma classe pode herdar de uma classe e implementar uma interface, a qual é um tipo de classe abstrata. Portanto, uma classe pode herdar de seu pai natural e implementar a interface **Runnable**, a qual oferece suporte parcial para concorrência.

### 13.7.1 A Classe **Thread**

Os evidentes aspectos essenciais de **Thread** são dois métodos chamados **run** e **start**. O método **run** é sempre sobreposto por subclasses de **Thread**. Ele é exatamente o lugar onde

Hidden page

Quando há linhas com diferentes prioridades, o comportamento do scheduler é controlado pelas prioridades. Quando uma linha em execução é bloqueada, eliminada ou sua fatia de tempo extingue-se, o scheduler escolhe a linha da fila de tarefas prontas que tem a prioridade mais elevada. Uma linha com prioridade mais baixa será executada somente se outra com prioridade mais alta não estiver na fila de tarefas prontas quando a oportunidade surgir.

### 13.7.3 Sincronização de Competição

Em Java, a sincronização de competição é obtida forçando os métodos que acessam dados compartilhados a serem completamente executados antes que outro método seja executado sobre o mesmo objeto. Em outras palavras, podemos especificar que, assim que um método particular iniciar sua execução, essa execução será concluída antes que qualquer outro método inicie sua execução sobre o mesmo objeto. Esse método coloca um bloqueio no objeto, o qual impede outros métodos de serem executados sobre ele. Isso é especificado adicionando-se o modificador **synchronized** à definição do método, como na seguinte definição esquemática de classe:

```
class GerenciaRetentor {
    private int [100] retentor;
    ...
    public synchronized void deposita(int item) { ... }
    public synchronized int busca() { ... }
    ...
}
```

Os dois métodos definidos em *GerenciaRetentor* são qualificados com **synchronized**, o que os impede de interferir um no outro enquanto executam sobre o mesmo objeto, mesmo que sejam chamados por linhas separadas.

Em alguns casos, o número de instruções que lida com a estrutura de dados compartilhada é significativamente menor do que o número de outras instruções no método em que reside. Nesses casos, é melhor sincronizar o segmento de código que acessa ou muda a estrutura de dados compartilhada em vez de o método inteiro. Isso pode ser feito com a denominada instrução sincronizada, cuja forma geral é:

```
synchronized(expressão)
    instrução
```

em que a expressão deve ser avaliada para um objeto, e a instrução pode ser uma única instrução ou uma instrução composta. O objeto é bloqueado durante a execução da instrução ou da instrução composta. Assim, a instrução ou a instrução composta são executadas exatamente como se fossem o corpo de um método sincronizado.

Um objeto com métodos sincronizados definidos deve ter uma fila a ele associada que armazene os métodos sincronizados que tentaram executar sobre ele enquanto sofria a operação de outro método sincronizado. Quando um método sincronizado conclui sua execução sobre um objeto, um método à espera na fila de espera do objeto, se houver, é colocado na fila de tarefas prontas.

### 13.7.4 Sincronização de Cooperação

A sincronização de cooperação em Java é realizada usando-se os métodos `wait` e `notify` definidos em `Object`, a classe-raiz de todas as classes Java. Todas as classes, exceto `Object`, herdam esses métodos. O método `wait` é colocado em um laço que testa a condição de acesso legal. Se a condição for falsa, a linha é posta em uma fila para esperar. O método `notify` é chamado para dizer a uma linha à espera que o que ela estava esperando aconteceu.

`wait` e `notify` somente podem ser chamados de dentro de um método sincronizado, porque eles usam o bloqueio colocado em um objeto por esse método.

O método `wait` pode jogar `InterruptedException`, descendente de `Exception`. Portanto, qualquer código que chame `wait` também deve pegar `InterruptedException`. Presumindo que a condição para a qual esperamos que isso aconteça seja chamada `aCondicao`, a maneira convencional de usarmos `wait` é a seguinte:

```
try {
    while (!aCondicao)
        wait();
    -- Faça qualquer coisa que seja necessária depois que
       aCondicao acontecer
}
catch(InterruptedException meuProblema) { ... }
```

A cláusula `try` define o escopo da manipulação de exceções, e `catch` é o manipulador de exceções para a cláusula `try`.

O programa seguinte implementa uma fila circular para armazenar valores inteiros. Ele ilustra tanto a sincronização de cooperação como a de competição.

```
// Fila
// Esta classe implementa uma fila circular para armazenar valores
// int. Ela inclui um construtor para alocar e inicializar a fila
// em um tamanho especificado. Ela tem métodos sincronizados para
// inserir valores e para remover valores da fila.

class Fila {
    private int [] F;
    private int proximoDentro,
                proximoFora,
                preenchido,
                tamFila;

    public Fila(int tamanho) {
        F = new int [tamanho];
        preenchido = 0;
        proximoDentro = 1;
        proximoFora = 1;
        tamFila = tamanho;
    } //** fim do construtor Fila
```

```

public synchronized void deposita (int item) {
    try {
        while (preenchido == tamFila)
            wait();
        F [proximoDentro] = item;
        proximoDentro = (proximoDentro * tamFila) + 1;
        preenchido++;
        notify();
    } /** fim da cláusula try
    catch(InterruptedException e) {}
} /** fim do método deposita

public synchronized int busca() {
    int item = 0;
    try {
        while (preenchido == 0)
            wait();
        item = F [proximoFora];
        proximoFora = (proximoFora * tamFila) + 1;
        preenchido--;
        notify();
    } /** fim da cláusula try
    catch(InterruptedException e) {}
    return item;
} /** fim do método busca
} /** fim da classe Fila

```

Note que o manipulador de exceções (**catch**) nada faz aqui.

As classes que definem objetos produtores e consumidores que poderiam usar a classe **Fila** podem ser definidas como:

```

class produtor extends Thread {
    private Fila retentor;
    public Produtor(Fila F) {
        retentor = F;
    }
    public void run() {
        int novo_item;
        while (true) {
            // -- Crie um novo_item
            retentor.deposita(novo_item);
        }
    }
}

class Consumidor extends Thread {
    private Fila retentor;
    public Consumidor(Fila F) {
        retentor = F;
    }
}

```

Hidden page

O problema encaminhado pelas construções de linguagem que discutiremos é o de minimizar a comunicação necessária entre processadores e as memórias destes. A suposição é que seja mais rápido para um processador acessar dados em sua própria memória do que na de algum outro. Compiladores bem projetados podem contribuir muito 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 usada.

### 13.8.1 High-Performance FORTRAN

O High-Performance FORTRAN (HPF) (ACM, 1993b) é uma coleção de extensões ao FORTRAN 90 destinada a permitir que os programadores especifiquem informações ao compilador para ajudá-lo a otimizar a execução de programas em computadores de multiprocessador. O HPF inclui tanto novas instruções de especificação como subprogramas intrínsecos ou incorporados. Esta seção discutirá somente algumas das novas instruções.

As principais instruções de especificação do HPF servem para especificar o número de processadores, a distribuição de dados nas memórias desses processadores e o alinhamento de dados com outros dados em termos de colocação de memória. As instruções de especificação do HPF aparecem como comentários especiais em um programa FORTRAN. Cada uma delas é introduzida pelo prefixo !HPF\$, em que ! é o caractere usado para iniciar linhas de comentários no FORTRAN 90. Tal prefixo torna-os invisíveis aos compiladores FORTRAN 90, mas fáceis de serem reconhecidos por compiladores HPF.

A especificação PROCESSORS tem a forma:

```
!HPF$ PROCESSORS procs (n)
```

Essa instrução é usada para especificar ao compilador o número de processadores que podem ser usados pelo código gerado para esse programa. A informação é usada em conjunto com outras especificações para dizer ao compilador como os dados devem ser distribuídos para as memórias associadas aos processadores.

A instrução DISTRIBUTE especifica quais dados devem ser distribuídos e o tipo de distribuição a ser usada. Sua forma é

```
!HPF$ DISTRIBUTE (tipo) ONTO procs :: lista_de_identificadores
```

Nessa instrução, o tipo pode ser BLOCK ou CYCLIC. A lista de identificadores são os nomes das variáveis do tipo vetor que devem ser distribuídas. Uma variável especificada como distribuída em bloco (BLOCK distributed) é dividida em *n* grupos iguais, em que cada grupo consiste em coleções contíguas de elementos do vetor uniformemente distribuídos nas memórias de todos os processadores. Por exemplo, se um vetor chamado LISTA com 500 elementos distribuídos em bloco em cinco processadores, os 100 primeiros elementos de LISTA serão armazenados na memória do primeiro processador e assim por diante. Uma distribuição CYCLIC especifica que elementos individuais do vetor são armazenados ciclicamente na memória dos processadores. Por exemplo, se LISTA for distribuída ciclicamente (CYCLIC distributed), novamente em cinco processadores, o primeiro elemento de LISTA será armazenado na memória do primeiro, o segundo elemento na memória do segundo e assim por diante.

A forma da instrução ALIGN é

```
ALIGN elemento_vetor1 WITH elemento_vetor2
```

ALIGN é usada para relacionar a distribuição de um vetor com a de outro. Por exemplo,

Hidden page

Monitores são abstrações de dados que constituem uma maneira natural de permitir o acesso mutuamente exclusivo a dados compartilhados entre tarefas. Eles são incluídos em diversas linguagens de programação. A sincronização de cooperação em linguagens com monitores deve ser fornecida com alguma forma de semáforos.

A Ada oferece construções complexas, mas eficientes, baseadas no modelo de passagem de mensagens, para concorrência. As unidades concorrentes básicas são tarefas que se comunicam entre si pelo mecanismo de *rendezvous*, que é uma passagem de mensagens síncronas. Um *rendezvous* é a ação de uma tarefa que aceita uma mensagem enviada por outra. A Ada inclui tanto um método simples como um complicado de controlar as ocorrências de *rendezvous* entre as tarefas.

A Ada 95 inclui capacidades adicionais para suporte de concorrência, principalmente os objetos protegidos e a passagem de mensagem assíncrona.

O Java fornece unidades concorrentes de uma maneira simples, mas efetiva. Qualquer classe que herde de `Thread` ou implemente `Runnable` pode sobrepor a um método herdado chamado `run` e ter o método do código executado concorrentemente com outros desses métodos e com o programa principal. A sincronização de competição é especificada definindo-se métodos que acessem dados compartilhados para serem sincronizados. Pequenas seções de código também podem ser sincronizadas. A sincronização de cooperação é levada a efeito usando-se os métodos `wait` e `notify`.

O High-Performance FORTRAN inclui instruções para especificar como os dados devem ser distribuídos nas unidades de memória conectadas a múltiplos processadores. Também estão incluídas instruções para especificar coleções de instruções que podem ser executadas concorrentemente.

## *NOTAS BIBLIOGRÁFICAS*

O tema geral da concorrência é extensamente discutido em Andrews e Schneider (1983), Hot et al. (1978) e Ben-Ari (1982).

O conceito de monitor foi desenvolvido e sua implementação no Concurrent Pascal, descrita por Brinch Hansen (1977).

O primeiro desenvolvimento do modelo de passagem de mensagens de controle de unidades concorrentes é discutido por Hoare (1978) e Brinch Hansen (1978). Uma discussão em profundidade do desenvolvimento do modelo de tarefas Ada pode ser encontrada em Ichbiah et al. (1979). A Ada 95 é descrita detalhadamente em AARM (1995). O High-Performance FORTRAN é descrito em ACM (1993b).

## *QUESTÕES DE REVISÃO*

1. Quais são os três níveis de concorrência nos programas?
2. Qual nível de concorrência de programa é melhor suportado por computadores SIMD?
3. Qual nível de concorrência de programa é melhor suportado por computadores MIMD?
4. O que é a linha de controle de um programa?
5. Defina tarefa, tarefa disjunta, sincronização, sincronização de competição e de cooperação, vivência e enlace mortal.
6. Quais tipos de tarefa não exigem nenhum tipo de sincronização?
7. Quais são as questões de projeto referentes ao suporte de linguagem para concorrência?
8. Descreva as ações das operações de espera (`wait`) e de liberação (`release`) para semáforos.
9. O que é um semáforo binário? O que é um de contagem?

10. Quais são os principais problemas decorrentes do uso de semáforos para fornecer sincronização?
11. Qual vantagem os monitores têm sobre os semáforos?
12. Defina *rendezvous*, cláusula **accept**, cláusula **entry**, tarefa atuante, tarefa servidora, cláusula **accept** estendida, cláusula **accept** aberta, cláusula **accept** fechada e tarefa concluída.
13. Qual é mais geral: a concorrência por monitores ou a concorrência por passagem de mensagens?
14. As tarefas Ada são criadas estática ou dinamicamente?
15. Para que propósito serve uma cláusula **accept** estendida?
16. Como a sincronização de cooperação é fornecida para tarefas Ada?
17. Qual é a vantagem dos objetos protegidos da Ada 95 em relação às tarefas para oferecer acesso a objetos de dados compartilhados?
18. Descreva a cláusula **select** assíncrona da Ada 95.
19. Especificamente, qual unidade de programa Java pode rodar concorrentemente com o método principal em um programa aplicativo?
20. O que o método **sleep** Java faz?
21. O que o método **yield** Java faz?
22. Quais são as duas construções Java que podem ser sincronizadas?
23. Quais métodos Java são usados para suportar sincronização de cooperação?
24. Explique porque o Java inclui a interface **Runnable**.
25. Qual é o objetivo das instruções de especificação do High-Performance FORTRAN?
26. Qual é a finalidade da instrução **PORALL** do High-Performance FORTRAN?

## PROBLEMAS

1. Explique claramente por que a sincronização de competição não é um problema em um ambiente de programação que tem controle de unidade simétrico, mas sem concorrência.
2. Qual é a melhor ação que um sistema pode desenvolver quando um enlace mortal é detectado?
3. Escreva uma tarefa Ada para implementar semáforos gerais.
4. Escreva uma tarefa Ada para gerenciar um retentor compartilhado como o de nosso exemplo, mas use a tarefa de semáforo do Problema 3.
5. Espera ocupada (*busy waiting*) é um método por meio do qual uma tarefa espera que determinado evento ocorra verificando continuamente se esse evento ocorre. Qual é o problema com essa abordagem?
6. No exemplo de produtor-consumidor da Seção 13.3, suponhamos que substituímos incorretamente a **release(acesso)** do processo consumidor por **wait(acesso)**. Qual seria o resultado desse erro na execução do sistema?
7. De um livro sobre programação em linguagem de montagem VAX, determine quais instruções a estrutura VAX inclui para suportar a construção de semáforos.
8. De um livro sobre programação em linguagem de montagem para um computador que usa um processador Intel Pentium, determine quais instruções são fornecidas para suportar a construção de semáforos.
9. Suponhamos que duas tarefas, A e B, devam usar a variável compartilhada **TAM\_RET**. A tarefa A adiciona 2 a **TAM\_RET**, e a tarefa B subtrai 1 dela. Suponhamos que tais operações aritméticas sejam feitas pelo processo em três etapas: buscar o valor atual, realizar a operação aritmética e colocar de volta o novo valor. Na ausência de sincronização de competição, quais seqüências de eventos são possíveis e quais valores resultam dessas operações? Suponhamos que o valor inicial de **TAM\_RET** seja 6.
10. Compare o mecanismo de sincronização de competição do Java com o da Ada.
11. Compare o mecanismo de sincronização de cooperação do Java com o da Ada.
12. O que acontece se um procedimento monitor chamar outro procedimento no mesmo monitor?

## Capítulo 14

# Manipulação de Exceções



### **Edsger Dijkstra**

Edsger Dijkstra, ganhador do ACM Turing Award em 1972, ocupou a Schlumberger Chair\* em Ciência da Computação na Universidade do Texas, em Austin, no período 1984-1999. Ele foi membro da equipe que abriu caminho para "The Multiprogramming System", que foi o primeiro sistema operacional mundial com processos concorrentes. Essa estrutura tornou viáveis provas de ausência de perigo de enlace mortal e de outras propriedades de exatidão. Em 1976, Dijkstra escreveu *A Discipline of Programming* e, em 1982, escreveu *Selected Writings on Computing: A personal perspective*. Faleceu em agosto de 2002.

- 14.1** Introdução à Manipulação de Exceções
- 14.2** Manipulação de Exceções em PL/I
- 14.3** Manipulação de Exceções em Ada
- 14.4** Manipulação de Exceções em C++
- 14.5** Manipulação de Exceções em Java

\*N. de T. Schlumberger Chair: The Schlumberger Centennial Chair in Computer Sciences é uma das diversas cátedras universitárias mantidas por meio de dotação orçamentária que o Departamento de Ciência da Computação da Universidade do Texas — Austin — tem para reconhecer membros eminentes do corpo docente. Além da honraria, o departamento oferece um salário adicional em dinheiro e fundos discricionários (isso é, disponíveis para serem usados quando necessário). A dotação que custeia esta cátedra foi concedida em parte pela Schlumberger Corporation. Possuir uma cátedra (Chair) não é o mesmo que ser presidente do departamento, uma responsabilidade administrativa que gira entre os membros "sênior" do corpo docente.

Este capítulo descreve, primeiro, os conceitos fundamentais da manipulação de exceções, inclusive as detectáveis por hardware e por software, os seus manipuladores de exceção e o seu levantamento. Depois, serão apresentadas e discutidas as questões de projeto relativas aos manipuladores de exceção, incluindo a vinculação de exceções a manipuladores de exceção, continuação, manipuladores padrão e desativação de execuções. O restante do capítulo descreverá e avaliará as facilidades de manipulação de exceções de quatro linguagens de programação: PL/I, Ada, C++ e Java.

## 14.1 Introdução à Manipulação de Exceções

---

O hardware da maioria dos computadores é capaz de detectar certas condições de erro em tempo de execução, como, por exemplo, estouro de número real. Muitas linguagens de programação são projetadas e implementadas de uma maneira tal que o programa usuário não pode detectar, nem lidar com esses erros. Nessas linguagens, a ocorrência de erros simplesmente faz com que o programa seja finalizado, e o controle, transferido para o sistema operacional. A reação típica do sistema operacional a um erro é imprimir uma mensagem de diagnóstico, a qual pode ser muito significativa ou altamente crítica, e finalizar o programa depois.

No caso de operações de entrada e saída, entretanto, a situação, às vezes, é diferente. Por exemplo, uma instrução READ FORTRAN pode interceptar erros de entrada e condições de fim de arquivo, ambos detectados pelo hardware do dispositivo de entrada. Em ambos os casos, a instrução READ pode especificar o rótulo de alguma instrução no programa usuário que lida com a condição. No caso do fim de arquivo, é evidente que a condição nem sempre deve ser considerada um erro. Na maioria dos casos, nada mais é do que um sinal de que um tipo de processamento foi concluído e um novo deve se iniciar. Apesar da diferença óbvia entre o fim de arquivo e eventos sempre considerados erros, como um processo de entrada falho, o FORTRAN manipula ambas as situações com o mesmo mecanismo. Considere a seguinte instrução READ FORTRAN:

```
READ(UNIT=5, FMT=1000, ERR=100, END=999) PESO
```

A cláusula ERR especifica que o controle deve ser transferido para a instrução rotulada 100 se ocorrer um erro na operação de leitura. A cláusula END especifica que o controle deve ser transferido para a instrução rotulada 999 se a operação de leitura encontrar o final do arquivo. Desse modo, o FORTRAN usa desvios simples tanto para erros de entrada como para fim de arquivo.

Há uma categoria de erros sérios não-detectáveis por hardware, mas que poderiam ser detectados pelo código gerado pelo compilador. Por exemplo, os erros de faixa de subscrito de matriz quase nunca são detectados por hardware (houve poucos computadores que detectavam erros de faixa de subscrito em hardware), mas eles levam a erros fatais que, muitas vezes, somente são detectados mais tarde na execução do programa.

A detecção de erros de faixa de subscrito, às vezes, é exigida pelo projeto da linguagem. Por exemplo, os compilares Ada e Java precisam gerar código para verificar a correção de toda expressão de subscrito. No C, as faixas de subscrito não são verificadas porque não se acreditava (*e não se acredita*) que o seu custo valesse o benefício de detectar erros. Em alguns compiladores para algumas linguagens, a verificação de subscrito pode ser selecionada se for desejada pelo programa ou no comando que executa o compilador.

Muitas linguagens contemporâneas oferecem mecanismos que podem entrar em ação quando algumas condições detectáveis por hardware e certas detectáveis por software ocorrem. Elas também permitem que o programador defina outros eventos incomuns e use os mesmos mecanismos para lidar com eles quando surgirem. Tais mecanismos são chamados coletivamente de manipulação de exceções.

Talvez o mais importante motivo pelo qual algumas linguagens não incluem manipulação de exceções seja a complexidade que ela adiciona à linguagem.

#### 14.1.1 Conceitos Básicos

Denominamos como exceção os erros detectados por hardware (por exemplo, erros de leitura de disco) e condições incomuns (por exemplo, fim de arquivo, os quais também são detectados por hardware). Ampliamos o conceito de exceção para incluir erros ou condições incomuns detectáveis por software. Consequentemente, definimos **exceção** como qualquer evento, errôneo ou não, que seja detectável por hardware ou software e que possa exigir processamento especial.

O processamento especial que pode ser exigido pela detecção de uma exceção é chamado **manipulação da exceção**. Esse processamento é feito por uma unidade de código chamada **manipulador de exceção**. Uma exceção é **gerada** quando ocorre seu evento associado. Seus manipuladores normalmente são diferentes para diversos tipos de exceção. A detecção de fim de arquivo quase sempre exige alguma ação de programa específica. Mas, evidentemente, essa ação também não seria apropriada para uma exceção de estouro de número real. Em alguns casos, a única ação poderia ser a geração de uma mensagem de erro e uma finalização ordenada do programa.

Em algumas situações, pode ser desejável ignorar certas exceções durante algum tempo. Isso seria feito desativando-a. Uma exceção desativada poderia ser ativada novamente mais tarde.

A ausência de facilidades distintas ou específicas de manipulação de erros em uma linguagem não impediria a manipulação de exceções definidas pelo usuário, detectadas por software. Essa exceção detectada dentro de uma unidade de programa, muitas vezes, é manipulada pelo chamador ou pelo invocador. Um possível projeto é enviar um parâmetro auxiliar, usado como uma variável de *status*. A esta última é atribuído um valor na unidade chamada, de acordo com a correção e/ou com a normalidade de sua computação. Imediatamente depois da unidade chamada, o chamador testa a variável de *status*. Se o valor indicar que ocorreu uma exceção, o manipulador, que pode residir na unidade que faz a chamada, pode ser ativado. Muitas das funções de biblioteca da linguagem C usam uma variante dessa abordagem. Os valores de retorno são usados como indicadores de erro.

Outra possibilidade é passar um rótulo como parâmetro ao subprograma. Logicamente, isso somente é possível em linguagens que permitem o uso de rótulos como parâmetro. A passagem de um rótulo permite que a unidade chamada retorne a um ponto diferente no chamador se uma exceção tiver ocorrido. Como na primeira alternativa, o manipulador freqüentemente é um segmento do código da unidade que faz a chamada. Eis um uso comum dos parâmetros rótulo no FORTRAN.

Uma terceira possibilidade é ter o manipulador como um subprograma separado passado como um parâmetro à unidade chamada. Nesse caso, o subprograma manipulador é fornecido pelo chamador, mas a unidade chamada invoca o manipulador quando uma exceção é gerada. Um problema com tal abordagem é que é exigido o envio de um subprograma manipulador em toda chamada, seja isso desejável ou não. Além disso, para lidar com di-

versos tipos diferentes de exceções, seria necessário passar diversas rotinas manipuladoras diferentes, complicando o código.

Se for desejável manipular uma exceção na unidade em que ela é detectada, o manipulador será simplesmente um segmento de código nessa unidade.

Existem algumas vantagens definitivas de ter-se manipulação de exceções incorporada a uma linguagem. Primeiro, sem exceção, manipular o código necessário para detectar condições de erro pode embaralhar consideravelmente um programa. Por exemplo, suponhamos que um subprograma inclua expressões com 10 operações de divisão e que cada uma pudesse ter um divisor zero. Sem manipulação de exceções incorporada, cada uma dessas operações precisaria ser precedida por uma construção de seleção para detectar o possível erro de divisão por zero. A presença da manipulação de exceções na linguagem poderia permitir que o compilador inserisse essas verificações no código quando solicitado pelo programa.

Outra vantagem do suporte de linguagem para manipulação de exceções resulta da propagação de exceções, que permite a uma exceção levantada em uma unidade de programa ser manipulada em alguma outra unidade em seu ancestral dinâmico ou estático. Isso faz com que um único manipulador de exceções seja usado por um grande número de unidades de programa diferentes. A reutilização pode resultar em uma significativa economia em termos de custo de desenvolvimento e de tamanho de programa.

Uma linguagem que suporta manipulação de exceções encoraja seus usuários a considerar todos os eventos que podem ocorrer durante a execução do programa e como eles podem ser manipulados. Isso é bem melhor do que não considerar essas possibilidades e esperar que nada saia errado. Tal vantagem relaciona-se com exigir que uma construção seletora múltipla inclua ações para todos os valores possíveis da expressão de controle, como é exigido pela Ada.

Por fim, há programas nos quais o ato de lidar com situações não-errôneas, mas incomuns, pode ser simplificado com a manipulação de exceções, pois a estrutura de programa pode tornar-se demasiadamente convoluta sem ela.

#### 14.1.2 Questões de Projeto

Exploraremos agora algumas das questões de projeto referentes a um sistema de manipulação de exceções quando ele faz parte de uma linguagem de programação. Esse sistema poderia permitir tanto exceções incorporadas e definidas pelo usuário como seus manipuladores. Considere o seguinte subprograma esquemático que inclui um mecanismo de manipulação de exceções:

```
void exemplo() {
    ...
    media = soma / total;
    ...
    return;
/* Manipuladores de exceção */
when zero_divide {
    media = 0;
    printf("Erro-divisor (total) é zero\n");
}
...
}
```

A exceção da divisão por zero é interceptada na função, a qual transfere o controle para o manipulador apropriado, que será, então, executado.

A primeira questão de projeto referente a manipuladores de exceção definidos pelo usuário é sua forma. Eis, essencialmente, uma escolha entre manipuladores que são unidades de programa completas ou manipuladores que são segmentos de código. No último caso, eles podem ser incorporados às unidades que geram as exceções que devem manipular, como no exemplo dado, ou podem ser incorporados a uma unidade diferente, como, por exemplo, a unidade que chamou aquela na qual a exceção é gerada.

Se o manipulador for uma unidade separada e a linguagem usar escopo estático, ele poderá estar no mesmo escopo que o código que pode fazer com que ele seja gerado. Isso simplifica as comunicações entre as duas unidades. Se o manipulador for uma unidade fora da unidade que pode gerar sua exceção associada, a comunicação poderá ocorrer por meio de parâmetros.

Outra questão de projeto importante referente à manipulação de exceções é como uma ocorrência de exceção é vinculada a um manipulador de exceção. Essa questão ocorre em dois níveis diferentes. No nível da unidade, há a questão de como as mesmas exceções geradas em pontos diferentes de uma unidade podem ser vinculadas a diferentes manipuladores dentro desta. Por exemplo, no subprograma dado, há um manipulador para uma exceção de divisão por zero em uma instrução particular (a mostrada). Entretanto, suponhamos que a função inclua diversas outras expressões com operadores de divisão. Para esses operadores, tal manipulador, provavelmente, não é apropriado. Desse modo, deveria ser possível vincular as exceções que podem ser geradas por instruções particulares a manipuladores particulares, ainda que a mesma exceção possa ser gerada por muitas instruções diferentes.

Em um nível mais elevado, a questão da vinculação surge quando não há nenhum manipulador de exceções local à unidade em que a exceção é levantada. Nesse caso, o projetista deve decidir se propagará esta exceção para alguma outra unidade e, se assim for, para onde. Como essa propagação desenvolve-se e até onde ela vai tem um impacto importante na capacidade de escrita de manipuladores de exceção. Por exemplo, se eles precisarem ser locais, muitos manipuladores deverão ser escritos, o que complica a escrita e a leitura do programa. Por outro lado, se exceções forem propagadas, um único manipulador poderá manipular a mesma exceção levantada em diversas unidades de programa, o que pode exigir que ele seja mais geral do que o preferível.

Outro fator importante é se a vinculação de exceções a manipuladores é estática ou dinâmica; ou seja, se ela depende do projeto sintático do programa ou de sua sequência de execução. Como acontece em outras construções de linguagem, a vinculação estática de exceções é mais fácil de entender e de implementar do que a dinâmica.

Depois que um manipulador de exceções executa, duas coisas podem ocorrer: o controle pode transferir-se para algum outro lugar no programa fora do código do manipulador ou a execução do programa pode simplesmente ser finalizada. Denominamos essa questão de continuação do controle depois da execução do manipulador ou simplesmente de **continuação**. Evidentemente, a finalização é a opção mais simples, e, em muitas condições de exceção de erro, é a melhor. Porém, em outras situações, especialmente naquelas associadas com eventos incomuns, mas não-errôneos, a escolha de prosseguir a execução é a melhor. Nesses casos, algumas convenções devem ser escolhidas para determinar onde a execução deve prosseguir. Poderia ser a instrução que gerou a exceção, a seguinte à instrução que gerou a exceção ou possivelmente alguma outra unidade. A escolha de retornar à instrução que gerou a exceção pode parecer boa, mas, no caso de uma exceção de erro, ela é útil somente se o manipulador for, de alguma maneira, capaz de modificar os valores ou

as operações que fizeram com que a exceção fosse gerada. Caso contrário, este processo irá repetir-se. A modificação exigida por uma exceção de erro, às vezes, é muito difícil. Mesmo quando possível, entretanto, pode não ser uma prática segura. Ela permite ao programa remover os sintomas de um problema sem remover a causa.

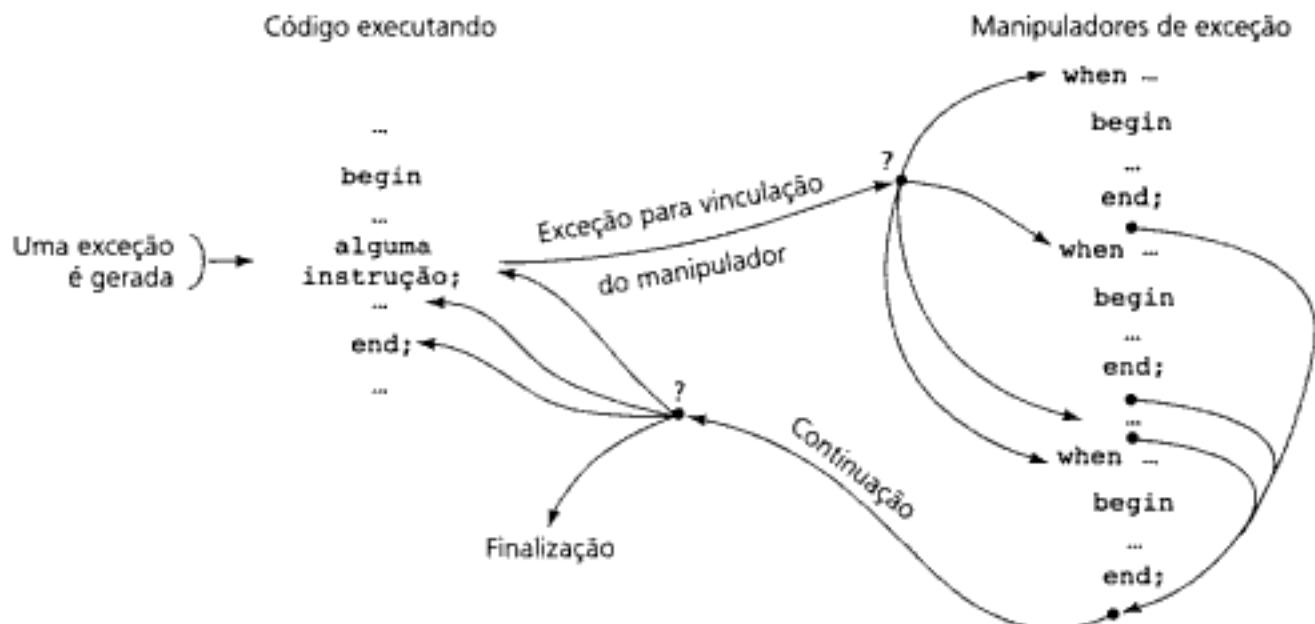
As duas questões da vinculação de exceções a manipuladores e da continuação são ilustradas na Figura 14.1.

Outra questão de projeto é a seguinte: se for permitido que os usuários definam exceções, como elas serão especificadas? A resposta usual é exigir que elas sejam declaradas nas partes de especificação das unidades de programa em que podem ser geradas. O escopo de uma exceção declarada usualmente é o mesmo da unidade de programa que contém a declaração.

No caso em que a linguagem fornece exceções incorporadas, surgem diversas outras questões de projeto. Por exemplo, o sistema da linguagem deve, em tempo de execução, oferecer manipuladores-padrão para as exceções incorporadas ou deve-se exigir que os usuários escrevam manipuladores para todas as exceções? Outra questão é se as exceções incorporadas podem ser geradas explicitamente pelo programa usuário. Isso pode ser conveniente se houver situações detectáveis por software nas quais o usuário gostaria de usar um manipulador incorporado.

Outra questão é se os erros detectáveis por hardware devem ser tratados como exceções possíveis de ser manipuladas por programas usuários. Se não, todas as exceções obviamente são detectáveis por software. Uma questão relacionada é se deveria haver alguma exceção incorporada.

Finalmente, há a questão de se as exceções, quer incorporadas ou definidas pelo usuário, podem ser desativadas temporariamente ou permanentemente. Tal questão é um tanto filosófica, especialmente no caso das condições de erro incorporadas. Por exemplo, suponhamos que uma linguagem tenha uma exceção incorporada gerada quando ocorre um erro de faixa de subscripto. Muitos acreditam que estes erros sempre devem ser detectados e, portanto, não deve ser possível que o programa desative a sua detecção. Outros argumentam que a verificação de faixas de subscripto é muito custosa para a produção de software,



**FIGURA 14.1** Fluxo de controle da manipulação de exceções.

na qual, presume-se, o código seja suficientemente isento de erros a ponto de erros de faixa não deverem ocorrer.

As questões de projeto de manipulação de exceções podem ser resumidas da seguinte maneira:

- Como e onde os manipuladores de exceção são especificados e qual é seu escopo?
- Como a ocorrência de uma exceção é vinculada a um manipulador de exceções?
- Onde a exceção continua, se for o caso, depois que um manipulador de exceções conclui sua execução? (Esta é a questão da continuação.)
- Como as exceções definidas pelo usuário são especificadas?
- Deve haver manipuladores de exceção padrão para programas que não fornecem seus próprios manipuladores?
- Exceções incorporadas podem ser explicitamente geradas?
- Os erros detectáveis por hardware são tratados como exceções que podem ser manipuladas?
- Há exceções incorporadas?
- Deve ser possível desativar exceções?

#### 14.1.3 História

A PL/I(ANSI) lançou o conceito de permitir que programas usuários sejam diretamente envolvidos na manipulação de exceções. A linguagem permite ao usuário escrever manipuladores de exceção para uma longa lista delas definidas pela linguagem. Além disso, a PL/I introduziu o conceito de exceções definidas pelo usuário, as quais permitem aos programas criarem exceções detectadas por software. Essas exceções usam os mesmos mecanismos para as execuções incorporadas.

Desde que a PL/I foi projetada, uma substancial quantidade de trabalho contribuiu para projetar métodos alternativos de manipulação de exceções. Em especial, a CLU (Liskov et al., 1984), a Mesa (Mitchel et al., 1979), a Ada, a COMMON LISP (Steele, 1984), a ML (Milner et al., 1990), o C++, o Modula-3 (Cardelli et al., 1989), a Eiffel e o Java incluem facilidades de manipulação de exceção.

Agora estamos preparados para examinar as facilidades de manipulação de exceção de quatro dessas linguagens de programação.

## 14.2 Manipulação de Exceções em PL/I

Ainda em um outro esforço pioneiro, os projetistas da PL/I envolveram-se com o problema de oferecer aos usuários os primeiros mecanismos lingüísticos para manipulação de exceções. Como era do estilo deles em outras áreas, ofereceram facilidades muito poderosas e altamente flexíveis. Mas, como acontece com outras construções PL/I, suas facilidades de manipulação de exceções são difíceis de entender, de implementar e de usar corretamente.

A linguagem oferece exceções incorporadas e permite aos usuários definir as suas.

Hidden page

tar sua execução. O projeto PL/I para continuação, muitas vezes, é confuso tanto para os leitores como para os escritores do programa.

#### 14.2.4 Outras Opções de Projeto

As exceções definidas pelo usuário são criadas em programas PL/I usando uma declaração simples com a forma

```
CONDITION nome_da_exceção
```

Todas as exceções incorporadas têm manipuladores próprios. Estes manipuladores podem sofrer preempção<sup>7</sup> por manipuladores de exceção definidos pelo usuário. As exceções definidas pelo usuário devem ser geradas explicitamente, o que é feito com uma instrução da forma

```
SIGNAL condição (nome_da_exceção)
```

Qualquer condição pode ser gerada explicitamente com uma instrução SIGNAL, não obstante as exceções incorporadas normalmente serem geradas implicitamente pelas condições de hardware e de software. Uma SIGNAL de uma exceção atualmente desativada não faz nada.

A PL/I define 22 exceções incorporadas que variam de erros aritméticos, como ZERODIVIDE, a erros de programação, como SUBSCRIPTRANGE. As exceções incorporadas são divididas em três categorias: (1) as sempre ativadas, (2) as ativadas quando não especificado, mas que podem ser desativadas pelo código-usuário e (3) as desativadas quando não especificado, mas que podem ser ativadas pelo código-usuário.

O processo de ativar e de desativar condições é realizado prefixando-se uma instrução, um bloco ou um procedimento com o nome ou com os nomes da exceção, como em

```
(SUBSCRIPTRANGE, NOOVERFLOW) :
  BEGIN;
  ...
  END;
```

Nesse caso, a exceção SUBSCRIPTRANGE é ativada, e a exceção OVERFLOW é desativada. (Os valores-padrão são os opostos, NOSUBSCRIPTRANGE e OVERFLOW.) O prefixo NO pode ser anexado a qualquer exceção que não seja permanentemente ativada (para desativá-la).

#### 14.2.5 Um Exemplo

O exemplo de programa seguinte ilustra dois usos comuns, mas simples, de manipuladores de exceção em PL/I. O programa computa e imprime uma distribuição de graus de entrada, usando um vetor de contadores. Há 10 categorias de graus (0-9, 10-19, ..., 90-100). Este mesmo é usado para computar índices em um vetor de contadores, um para cada categoria de grau. Graus de entrada inválidos são detectados, analisando os erros de indexação

<sup>7</sup>N. de R.T. Preempção indica interrupção, finalização. É um termo bastante usado na forma "preemptivo". O Windows, por exemplo, é um sistema operacional preemptivo, ou seja, ele pode interromper a execução de um processo mesmo que ele não tenha concluído ou esteja bloqueado.

Hidden page

### 14.2.6 Avaliação

A PL/I oferece uma poderosa facilidade para detecção e para manipulação de exceções. Seu elevado nível de flexibilidade, entretanto, não é sem custo. O principal exemplo disso é a vinculação dinâmica de exceções a manipuladores, que causa um problema de capacidade de escrita e de legibilidade relacionado aos problemas do escopo dinâmico. De fato, é o mesmo problema: o escopo do manipulador de exceções é dinâmico; sendo assim, é impossível determinar, a partir de uma listagem de programa, qual vinculação está em vigor em determinado ponto do programa. Devido à vinculação dinâmica, é fácil ter um manipulador usado de maneira não-intencional para uma exceção, de fato, longe dela mesma e também completamente impróprio para ela nessa situação. Por exemplo, considere o seguinte segmento de código:

```
(SUBSCRIPTRANGE):
    BEGIN;
    ...
    ON SUBSCRIPTRANGE
        BEGIN;
        PUT LIST('ERRO - SUBSCRITO INVÁLIDO NO VETOR SUBSUM');
        GO TO CONSERTE;
        END;
    ...
    ON SUBSCRIPTRANGE
        BEGIN;
        PUT LIST('ERRO - SUBSCRITO INVÁLIDO NO VETOR BLK');
        GO TO SAIR;
        END;
    ...
ROTULO1:;
    ...
    BLK(I, J, K) = SOMA;
    ...
END;
```

Se acontecer do código entre os dois manipuladores para a exceção SUBSCRIPTRANGE incluir uma GO TO ROTULO1, o primeiro manipulador seria executado se a exceção fosse levantada pela atribuição a BLK. Isso ativaría o manipulador errado, causando grande confusão para o usuário (ele afirmaria incorretamente que o erro fora com o vetor SUBSUM, em vez de com o BLK).

Outro problema sério é apresentado pela flexibilidade das regras de continuação das exceções PL/I. Elas são difíceis de implementar, prejudiciais à legibilidade da mesma maneira que os gotos, e também são difíceis de aprender a usar efetivamente.

Uma vez que os mecanismos da PL/I para manipulação de exceções eram considerados demasiadamente complexos, eles não foram copiados por outros projetistas de linguagens. Um modelo mais restrito foi proposto em 1975 por Goodenough (1975), em que as exceções são estaticamente vinculadas a manipuladores de exceção. Um modelo ainda mais restrito foi projetado para a linguagem CLU em meados da década de 70 (Liskov et al., 1984). Linguagens posteriores basearam seus projetos de manipulação de exceções, pelo menos em parte, no da CLU.

## 14.3 Manipulação de Exceções em Ada

A manipulação de exceções na Ada é uma ferramenta poderosa para construir sistemas de software mais confiáveis. Ela inclui as partes boas do projeto de manipulação de execuções tanto da PL/I como da CLU.

### 14.3.1 Manipuladores de Exceção

Os manipuladores de exceção da Ada normalmente são locais para o código no qual a exceção pode ser gerada. Uma vez que isso dá a eles o mesmo ambiente de referenciamento, parâmetros para manipuladores não são necessários e não são permitidos.

Os manipuladores de exceção têm a forma geral

```
when opção_de_exceção { | opção_de_exceção} => seqüência_de_instruções
```

em que as chaves são metassímbolos significando que os seus conteúdos podem ser deixados de fora ou repetidos qualquer número de vezes. A **opção\_de\_exceção** tem a forma

```
nome_da_exceção | others
```

O nome da exceção indica a exceção ou as exceções particulares que esse manipulador pretende manipular. A seqüência de instruções é o corpo do manipulador. A palavra reservada **others** indica que ele pretende manipular quaisquer exceções não-nomeadas em algum outro manipulador local.

Manipuladores de exceção podem ser incluídos em blocos ou nos corpos de subprogramas, de pacotes ou tarefas. Independentemente do bloco ou da unidade em que eles aparecem, os manipuladores são reunidos em uma cláusula **exception**, a qual deve ser colocada no final do bloco ou da unidade. Por exemplo, a forma usual de uma exception é mostrada no exemplo seguinte:

```
begin
  -- o bloco ou corpo da unidade --
exception
  when nome_da_exceção_1 =>
    -- primeiro manipulador --
  when nome_da_exceção_2 =>
    -- segundo manipulador --
    -- outros manipuladores --
end;
```

Qualquer instrução legítima no bloco ou na unidade em que o manipulador aparece também é legal no manipulador.

### 14.3.2 Vinculando Exceções a Manipuladores

Quando o bloco ou a unidade que gera uma exceção inclui um manipulador para essa exceção, ela é estaticamente vinculada a esse manipulador. Se uma exceção for gerada em um bloco ou em uma unidade que não tem um manipulador para essa exceção particular, ela será propagada para algum outro bloco ou unidade. A maneira pela qual as exceções são propagadas depende da entidade do programa na qual a exceção ocorre.

Quando uma exceção é gerada em um procedimento, quer seja na elaboração de suas declarações ou na execução de seu corpo, e o procedimento não tem nenhum manipulador para ela, a exceção é implicitamente propagada para a unidade de programa chamadora no ponto da chamada. Essa política reflete a filosofia de projeto segundo a qual a propagação da exceção deve seguir de volta pelo caminho de controle (ancestrais dinâmicos), não pelos ancestrais estáticos.

Se a unidade chamadora para a qual a exceção foi propagada também não tiver nenhum manipulador para ela, ela será novamente propagada para o chamador dessa unidade. Isso prosseguirá, se necessário, até o programa principal. Se uma exceção for propagada para o programa principal, e um manipulador ainda não tiver sido encontrado, o programa será finalizado.

No universo da manipulação de exceções, um bloco Ada é considerado um procedimento sem parâmetros que é “chamado” por seu bloco-pai quando o controle de execução atinge a primeira instrução do bloco. Quando uma exceção é gerada em um bloco, ou em suas declarações ou em suas instruções executáveis, e ele não tem nenhum manipulador para ela, a exceção é propagada para o escopo envolvente maior, que é o código que a “chamou”. O ponto para que a exceção é propagada está logo depois do fim do bloco no qual ela ocorreu, o ponto de “retorno”.

Quando uma exceção é gerada em um corpo de pacote e este não tem nenhum manipulador para a exceção, ela é propagada para a seção de declaração da unidade que contém a declaração do pacote. Se acontecer do pacote ser uma unidade de biblioteca (compilada separadamente), o programa será finalizado.

Se ocorrer uma exceção no nível mais externo em um corpo de tarefa e esta tiver um manipulador para a exceção, o manipulador será executado e a tarefa será marcada como concluída. Se a tarefa não tiver um manipulador para a exceção, ela simplesmente será marcada como concluída; a exceção não será propagada. O mecanismo de controle de uma tarefa é muito complexo para prestar-se a uma resposta razoável e simples à questão que diz respeito aonde suas exceções não manipuladas devem ser propagadas.

Exceções também ocorrem durante a elaboração das seções declarativas de subprogramas, de blocos, de pacotes e de tarefas. Por exemplo, suponhamos que uma função seja chamada para inicializar uma variável em sua instrução de declaração, como no seguinte exemplo:

```
procedure RIO is
    FLUXO_ATUAL : FLOAT := PEGA_FLUXO;
    ...
begin
    ...
end RIO;
```

Suponhamos que PEGA\_FLUXO seja uma função sem nenhum parâmetro. Se PEGA\_FLUXO gerar e propagar uma exceção ao seu chamador, a exceção será novamente levantada nesta declaração. A alocação de armazenamento durante a elaboração da declaração também poderá gerar uma exceção.

Quando exceções são geradas durante a elaboração da declaração de procedimentos, de pacotes e de blocos, elas são propagadas exatamente como se a exceção fosse gerada na seção de código associada. No caso de uma tarefa, há a marcação como concluída, nenhuma elaboração adicional desenvolve-se, e a exceção incorporada, TASKING\_ERROR, é gerada no ponto de ativação da tarefa.

### 14.3.3 Continuação

O bloco ou a unidade que gera uma exceção, juntamente com todas as unidades às quais a exceção foi propagada, mas que não a manipulou, é sempre finalizado. O controle nunca retorna implicitamente ao bloco ou à unidade que gera a exceção depois que ela é manipulada. O controle simplesmente prossegue depois da cláusula exception, a qual está sempre no final de um bloco ou de uma unidade. Isso causa um retorno imediato a um nível mais alto de controle.

Ao decidir onde uma execução prosseguiria depois que a execução do manipulador de exceções fosse concluída em uma unidade de programa, a equipe de projeto da Ada tinha poucas opções, porque a especificação de requisitos para a Ada (Departamento de Defesa, 1980a) afirma claramente que unidades de programa que geram exceções não podem ser continuadas ou retomadas. Porém, no caso de um bloco, uma instrução pode ser novamente tentada depois que ela gerar uma exceção e esta ser manipulada. Por exemplo, suponhamos uma instrução que possa gerar uma exceção e que um manipulador para ela esteja contido em um bloco, que, por sua vez, está contido em um laço. O exemplo seguinte, que recebe do teclado quatro valores inteiros na faixa desejada, ilustra esse tipo de estrutura:

```

...
type TIPO_IDADE is 0..125;
type TIPO_LISTA_IDADE is array (1..4) of TIPO_IDADE;
package IDADE_ES is new INTEGER_IO (TIPO_IDADE);
use IDADE_ES;
LISTA_IDADE : TIPO_LISTA_IDADE;
...
begin
for CONT_IDADE in 1..4 loop
    loop -- laço para repetição quando ocorrerem exceções
    EXCEPT_BLK:
        begin -- composto para encapsular manipulação de exceções
            PUT_LINE("Entre com um número inteiro na faixa 0..125");
            GET(LISTA_IDADE(CONT_IDADE));
            exit;
        exception
            when DATA_ERROR => -- A cadeia de entrada não é um número
                PUT_LINE("Valor numérico ilegal");
                PUT_LINE("Por favor, tente novamente");
            when CONSTRAINT_ERROR => -- A entrada é < 0 ou > 125
                PUT_LINE("O número de entrada está fora da faixa");
                PUT_LINE("Por favor, tente novamente");
        end EXCEPT_BLK;
    end loop;    -- fim do laço infinito para repetir a entrada
                 -- quando houver uma exceção
end loop;  -- fim do laço para CONT_IDADE em 1..4
...

```

O controle permanece no laço interno, que contém somente o bloco, até que um número de entrada válido seja recebido.

#### 14.3.4 Outras Opções de Projeto

A Ada inclui cinco exceções incorporadas,

```
CONSTRAINT_ERROR
NUMERIC_ERROR
PROGRAM_ERROR
STORAGE_ERROR
TASKING_ERROR
```

Cada uma delas é, de fato, uma categoria de exceções. Por exemplo, a exceção **CONSTRAINT\_ERROR** é gerada quando um subscrito de vetor está fora da faixa, quando há um erro de faixa em uma variável numérica que restringe a faixa, quando uma referência é feita a um campo de registro ausente em uma união discriminada e em algumas situações adicionais.

As exceções definidas pelo usuário têm a seguinte forma de declaração:

```
lista_com_nome_da_exceção : exception
```

Elas são tratadas exatamente como exceções incorporadas, exceto que devem ser geradas explicitamente.

Há manipuladores padrão para as exceções incorporadas, todos os quais resultam em finalização do programa.

Exceções são explicitamente geradas com a instrução **raise**, a qual tem a forma geral

```
raise [nome_da_exceção]
```

O único lugar em que uma instrução **raise** pode aparecer sem nomear uma exceção é dentro de um manipulador de exceções. Nesse caso, ela gerará novamente a mesma exceção que causou a do manipulador. Isso tem o efeito de propagá-la de acordo com as regras de propagação declaradas anteriormente. Uma **raise** em um manipulador de exceções é útil quando se deseja imprimir uma mensagem de erro em que uma exceção é gerada, mas se deseja manipulá-la em outro lugar.

Um **pragma Ada** é uma diretriz para o compilador. Certas verificações em tempo de execução que fazem parte das exceções incorporadas podem ser desativadas em programas Ada por meio do uso do **pragma SUPPRESS**, cuja forma simples é

```
pragma SUPPRESS(nome_da_verificação)
```

O **pragma SUPPRESS** pode aparecer somente em seções de declaração. Quando ele aparece, a verificação especificada pode ser suspensa no bloco, ou na unidade de programa associado do qual a seção de declaração faz parte. Gerações explícitas não são afetadas por **SUPPRESS**. Ainda que não se exija, a maioria dos compiladores Ada implementa o **pragma SUPPRESS**.

Exemplos de verificações que podem ser suprimidas são os seguintes: **INDEX\_CHECK** e **RANGE\_CHECK** especificam duas das verificações que normalmente são feitas em um programa Ada. **INDEX\_CHECK** refere-se a uma verificação de faixa de subscrito de matriz. **RANGE\_CHECK** refere-se à verificação de coisas como a faixa de um valor atribuído a uma variável de subtipo. Se **INDEX\_CHECK** ou **RANGE\_CHECK** for violado, **CONSTRAINT\_ERROR** será gerado. **DIVISION\_CHECK** e **OVERFLOW\_CHECK** são verificações suprimíveis associadas a **NUMERIC\_ERROR**. O **pragma** seguinte desativa a verificação de faixa de subscrito:

```
pragma SUPPRESS(INDEX_CHECK);
```

Há uma opção de **SUPPRESS** que permite à verificação nomeada ser ainda mais restrita para objetos, para tipos, para subtipos e para unidades de programa particulares.

#### 14.3.5 Um Exemplo

O exemplo seguinte tem o mesmo intento e uso da manipulação de exceções que o programa PL/I mostrado anteriormente neste capítulo. Ele produz uma distribuição de graus de entrada, usando um vetor de contadores para dez categorias de graus. Graus ilegais são detectados, verificando-se subscritos inválidos usados ao incrementar o contador selecionado.

```

with TEXT_IO; use TEXT_IO;
procedure DISTRIBUICAO_GRAU is
    package TEXTO_INTEIRO_ES is new INTEGER_IO(INTEGER);
    use TEXTO_INTEIRO_ES;
    FREQ: array (1..10) of INTEGER := (others => 0);
    NOVO_GRAU,
    INDEX,
    LIMITE_1,
    LIMITE_2 : INTEGER;
begin
    loop
        GET(NOVO_GRAU);
        INDEX := NOVO_GRAU / 10 + 1;
        begin -- Um bloco para o manipulador CONSTRAINT_ERROR
            FREQ(INDEX) := FREQ(INDEX) + 1;
        exception
            when CONSTRAINT_ERROR =>
                if NOVO_GRAU = 100 then
                    FREQ(10) := FREQ(10) + 1;
                else
                    PUT("ERRO -- novo grau: ");
                    PUT(NOVO_GRAU);
                    PUT(" está fora da faixa");
                    NEW_LINE;
                end if;
        end; -- fim do bloco para o manipulador CONSTRAINT_ERROR
    end loop;
exception -- Este manipulador inclui todas as computações finais
    when END_OF_FILE =>
        PUT("Frequência dos Limites");
        NEW_LINE; NEW_LINE;
        for INDICE in 0..9 loop
            LIMITE_1 := 10 * INDICE;
            LIMITE_2 := LIMITE_1 + 9;
            if INDICE = 9 then
                LIMITE_2 := 100;
            end if;
            PUT(LIMITE_1);

```

```

    PUT(LIMITE_2);
    PUT(FREQ(INDICE + 1));
    NEW_LINE;
end loop; -- para INDICE em 0..9...
end DISTRIBUICAO_GRAU;

```

Note que o código para manipular graus de entrada inválidos está em seu próprio bloco local. Isso permite que o programa prossiga depois que essas exceções são manipuladas, como em nosso exemplo anterior que lê valores do teclado.

#### 14.3.6 Avaliação

Como acontece em outras construções de linguagem, o projeto da Ada para manipulação de exceções representa certo consenso, pelo menos na época de seu projeto (final da década de 70 e início da década de 80), das idéias sobre o assunto. Ele é, claramente, um significativo avanço em relação à manipulação de exceções da PL/I. Durante algum tempo, a Ada foi a única linguagem popular que incluía manipulação de exceções. A manipulação de exceções em C++ e em Java mudou isso.

### 14.4 Manipulação de Exceções em C++

A manipulação de exceções do C++ foi aceita pelo comitê de padronização ANSI C++ em 1990 e, subsequentemente, abriu caminho para as implementações C++. O projeto baseia-se, parcialmente, na manipulação de exceções da CLU, da Ada e da ML.

#### 14.4.1 Manipuladores de Exceção

Na Seção 14.3, vimos que a Ada usa unidades ou blocos de programa para especificar o escopo de manipuladores de exceção. O C++ usa uma construção especial introduzida com a palavra reservada **try** para tal propósito. Uma construção **try** inclui uma instrução composta chamada cláusula **try** e uma lista de manipuladores de exceção. A instrução composta define o escopo dos manipuladores seguintes. A forma geral da construção é

```

try {
    /** O código que é esperado para gerar uma exceção
}
catch(parâmetro formal) {
    /** Um corpo de manipulador
}
...
catch(parâmetro formal) {
    // ** Um corpo de manipulador
}

```

Cada uma das funções **catch** são manipuladores de exceção. Uma **catch** pode ter somente um único parâmetro formal, similar a um parâmetro formal em uma definição de função

do C++, incluindo a possibilidade de ser somente reticências (...). O parâmetro formal pode ser um especificador de tipo simples, como, por exemplo, um **float**, como em um protótipo de função. Quando informações sobre a exceção são passadas ao manipulador, o parâmetro inclui um nome de tipo e um nome de variável usados para tal propósito. Por exemplo, o usuário pode definir uma classe como uma exceção e incluir quantos membros de dados forem necessários. Caso contrário, o parâmetro formal poderá ser apenas um nome de tipo, cuja única finalidade é tornar o manipulador unicamente identificável. Um manipulador com um parâmetro formal de reticências (...) é o manipulador “pega-tudo”; ele é ativado para qualquer exceção gerada se nenhum manipulador anterior for escolhido. O processo pelo qual as exceções geradas são conectadas a manipuladores será discutido na Seção 14.4.2.

No C++, os manipuladores de exceção podem incluir qualquer código C++.

#### 14.4.2 Vinculando Exceções a Manipuladores

Exceções C++ são geradas somente pela instrução explícita **throw**, cuja forma geral é

**throw** [expressão];

Os colchetes são metassímbolos usados para especificar que a expressão é opcional. Uma **throw** sem um operando pode aparecer somente em um manipulador. Quando aparece nele, gera novamente a exceção, a qual é manipulada em outro lugar. Isso é exatamente igual ao uso que a Ada faz da instrução **raise**, sem um nome de exceção.

A palavra **throw** foi escolhida porque tanto **signal** como **raise** são funções na biblioteca de padrões ANSI C.

O tipo da expressão **throw** seleciona o parâmetro particular, o qual, obviamente, deve ter um parâmetro formal de tipo “coincidente”. Nesse caso, coincidente significa o seguinte: um manipulador com um parâmetro formal do tipo **T**, **const T**, **T&** (uma referência a um objeto do tipo **T**) ou **const T&** coincide uma **throw** com uma expressão do tipo **T**. No caso em que este é uma classe, coincide um manipulador cujo parâmetro é do tipo **T** ou qualquer classe que seja um ancestral de **T**. Há situações mais complicadas em que uma expressão **throw** coincide com um parâmetro formal, mas elas não serão descritas aqui.

Uma exceção gerada em uma construção **try** provoca uma finalização imediata na exceção do código nesta **try**. A busca de um manipulador coincidente inicia-se com os manipuladores imediatamente seguintes à construção **try**. O processo de busca é feito seqüencialmente nos manipuladores até que uma coincidência seja encontrada. Isso significa que qualquer outro coincidente que precede o manipulador coincidente exato fará com que este último não seja usado. Se houver uma **catch** com um parâmetro formal de reticências, ela coincidirá com qualquer **throw**, de modo que não é útil colocá-la em qualquer lugar, a não ser no fim da lista de manipuladores.

As exceções são manipuladas de maneira local somente se for encontrado um manipulador local. Se não for encontrada nenhuma coincidência de manipulador, a expressão será propagada para o chamador da função na qual ela foi gerada. Se nenhum manipulador coincidente for encontrado no programa, este será finalizado.

### 14.4.3 Continuação

Depois que um manipulador concluiu sua execução, o controle flui para a primeira instrução depois da construção **try** (a instrução imediatamente depois do último manipulador na seqüência dos manipuladores da qual ele é um elemento). Um manipulador pode gerar novamente uma exceção, usando uma **throw** sem uma expressão, em cujo caso ela é propagada para o chamador.

### 14.4.4 Outras Opções de Projeto

Em termos das questões de projeto resumidas na Seção 14.1.2, a manipulação de exceções do C++ é simples. Há somente exceções definidas pelo usuário, e elas não são especificadas (ainda que possam ser declaradas como novas classes). Não há manipuladores padrão, porque as exceções detectadas pelo sistema não podem ser manipuladas. Elas não podem ser desativadas.

Uma função C++ pode listar os tipos de exceções (os tipos das expressões **throw**) que ela poderia gerar. Isso é feito anexando-se a palavra reservada **throw**, seguida de uma lista entre parênteses destes tipos, ao cabeçalho da função. Por exemplo,

```
int fun() throw (int, char *) { ... }
```

especifica que a função **fun** poderia gerar exceções do tipo **int** e **char \***, mas não outras. Se os tipos na cláusula **throw** forem classes, a função poderá gerar qualquer exceção que seja derivada das classes listadas. Se um cabeçalho de função tiver uma cláusula **throw** e gerar uma exceção que não está listada na cláusula e que não deriva de uma classe listada ali, isso causará um erro fatal. A lista de tipos pode estar vazia, significando que a função não gerará nenhuma exceção. Se não houver nenhuma especificação **throw** no cabeçalho, a função poderá gerar qualquer exceção. A lista não faz parte do tipo da função.

Quando uma exceção finaliza uma construção **try**, todas as variáveis dinâmicas na pilha e no monte alocadas pelo código executado na construção antes que a exceção ocorresse são desalocadas. Portanto, o manipulador jamais pode acessar essas variáveis.

### 14.4.5 Um Exemplo

O exemplo seguinte, mais uma vez, tem o mesmo intento e uso da manipulação de exceções que o programa PL/I mostrado na Seção 14.2.5. Ele produz uma distribuição de graus de entrada usando um vetor de contadores para 10 categorias. Graus ilegais são detectados verificando-se subscritos inválidos usados ao incrementar o contador selecionado.

```
#include <iostream.h>
void principal() { /* Qualquer exceção pode ser gerada
    int novo_grau,
        indice,
        limite_1,
        limite_2,
        freq[10] = {0,0,0,0,0,0,0,0,0,0};
    short int condicao_eof;
    try {
        while (1) {
```

```

        if (!cin >> novo_grau) /* Quando cin detectar eof,
            throw condicao_eof; /* gerar a condição eof
        indice = novo_grau / 10;
        {try {
            if (indice < 0 || indice > 9)
                throw(novo_grau);
            freq[indice]++;
        } /* fim do composto try interno
        catch(int grau) { /* Manipulador para erros de índice
            if (grau == 100)
                freq[9]++;
            else
                cout << "Erro -- novo grau: " << grau
                    << " está fora da faixa" << endl;
        } /* fim de catch(int grau)
        } /* fim do bloco para o par interno try-catch
    } /* fim de while (1)
} /* fim do composto externo try
catch(short int) { /* Manipulador para eof
    cout << "Freqüência dos Limites" << endl;
    for (indice = 0; indice < 10; indice++) {
        limite_1 = 10 * indice;
        limite_2 = limite_1 + 9;
        if (indice == 9)
            limite_2 = 100;
        cout << limite_1 << limite_2 << freq[indice] << endl;
    } /* fim de for (indice == 9)
} /* fim de catch (short int)
} /* fim de principal

```

Esse programa pretende ilustrar a mecânica da manipulação de exceções em C++. Porém, ambos os usos de exceções no exemplo são melhor manipulados por outros meios. A condição fim-de-arquivo é mais fácil de manipular simplesmente controlando-se o laço `while` com a expressão `cin` como expressão de controle. Além disso, a exceção de faixa de índice usualmente é manipulada em C++ sobrecarregando-se a operação de indexação, a qual poderia, então, gerar a exceção, em vez da detecção direta da operação de indexação com a construção de seleção usada em nosso exemplo.

#### 14.4.6 Avaliação

Sob alguns aspectos, o mecanismo de manipulação de exceções do C++ é similar ao da Ada: a vinculação de exceções a manipuladores é estática, e as exceções não-manipuladas são propagadas para o chamador da função. Mas, sob outros aspectos, o projeto do C++ é bem diferente: não há nenhuma exceção detectável por hardware incorporada que possa ser manipulada pelo usuário, e as exceções não são nomeadas. Isso leva à estranha situação de exceções serem conectadas a manipuladores por um tipo de parâmetro no qual o parâmetro formal poderia estar ausente. O tipo do parâmetro formal de um manipulador determina a condição sob a qual ele é chamado, mas pode não ter nada a ver com a natureza da exceção gerada. Portanto, o uso de tipos predefinidos certamente não promove a legibilidade.

de. É muito melhor definir classes com nomes significativos em uma hierarquia significativa que possam ser usadas para definir exceções.

## 14.5 Manipulação de Exceções em Java

No Capítulo 13, o exemplo de programa Java inclui o uso de manipulação de exceções com poucas explicações. Esta seção descreve os detalhes das capacidades de manipulação de exceções do Java.

A manipulação de exceções do Java baseia-se na do C++, mas foi projetada para estar mais dentro do paradigma de linguagem orientada a objeto.

### 14.5.1 Classes de Exceções

Todas as exceções Java são objetos de classes descendentes da classe `Throwable`. O Java inclui duas classes de exceções definidas pelo sistema que são subclasses de `Throwable`, `Error` e `Exception`. A classe `Error` e suas descendentes estão relacionadas a erros gerados pelo interpretador Java, como, por exemplo, o esgotamento de memória do monte. Essas exceções jamais são geradas por programas usuários e nunca devem ser manipuladas aí. Há duas descendentes diretas de `Exception`, `RuntimeException` e `IOException` definidas pelo sistema. Como seu nome indica, `IOException` é gerada quando ocorre um erro em uma operação de entrada ou saída, todas definidas como métodos nas várias classes definidas no pacote `java.io`.

Há classes definidas pelo sistema que são descendentes de `RuntimeException`. Na maioria dos casos, `RuntimeException` é gerada quando um programa usuário causa um erro. Por exemplo, `ArrayIndexOutOfBoundsException`, que é definido em `java.util`, e `NullPointerException` são exceções comumente geradas que descendem de `RuntimeException`.

Programas usuários podem definir suas próprias classes de exceção. A convenção em Java é que as exceções definidas pelo usuário são subclasses de `Exception`.

### 14.5.2 Manipuladores de Exceção

Os manipuladores de exceção do Java têm a mesma forma que os do C++, exceto pelo fato de que o parâmetro de toda `catch` deve estar presente e sua classe deve ser uma descendente da classe predefinida, `Throwable`.

A sintaxe da construção `try` em Java é exatamente como a do C++.

### 14.5.3 Vinculando Exceções a Manipuladores

Gerar uma exceção é muito simples. Uma instância da classe `exception` é dada como operando da instrução `throw`. Por exemplo, suponhamos que definimos uma exceção chamada `MinhaExcecao` como

```
class MinhaExcecao extends Exception {
```

```

public MinhaExcecao() {}
public MinhaExcecao(String message) {
    super (message)
}
}

```

Esta exceção pode ser gerada com

```
throw new MinhaExcecao();
```

A criação da instância da exceção para a `throw` poderia ser feita separadamente da instrução `throw`, como em

```

MinhaExcecao meuObjetoExcecao = new MinhaExcecao();
...
throw meuObjetoExcecao;

```

Um dos dois construtores que incluímos em nossa nova classe não tem nenhum parâmetro, e o outro tem um de objeto `String` que envia à superclasse (`Exception`) que, por sua vez, exibe-o. Desse modo, nossa nova exceção poderia ser gerada com

```

throw new MinhaExcecao
("uma mensagem para especificar a localização do erro");

```

A vinculação de exceções a manipuladores em Java é menos complexa do que no C++. Se uma exceção for gerada na instrução composta de uma construção `try`, ela será vinculada ao primeiro manipulador (função `catch`), imediatamente depois da cláusula `try`, cujo parâmetro é da mesma classe que o objeto gerado ou um ancestral dele. Se um manipulador coincidente for encontrado, a `throw` será vinculada a ele e ele será executado.

As exceções podem ser manipuladas e depois regeradas, incluindo uma instrução `throw` sem um operando no final do manipulador. A exceção recém-gerada não será manipulada na mesma `try` em que foi gerada originalmente; sendo assim, o laço não preocupa. Tal regeração normalmente é feita quando alguma ação local é útil, mas uma manipulação adicional por meio de uma cláusula `try` envolvente ou de um chamador é necessária. Uma instrução `throw` em um manipulador também poderia gerar alguma exceção diferente daquela que transferiu o controle para o manipulador; uma exceção particular poderia fazer com que outra fosse gerada.

#### 14.5.4 Continuação

Quando um manipulador é encontrado na seqüência de manipuladores em uma construção `try`, ele é executado; a execução do programa prossegue com a instrução que vem depois da construção `try`. Se nada for encontrado, os manipuladores das `try` envolventes serão procurados, sendo o mais interno em primeiro lugar. Se nenhum manipulador for encontrado nesse processo, a exceção será propagada para o chamador do método. Se a chamada do método tiver sido em uma cláusula `try`, a busca pelo manipulador prosseguirá na coleção anexada de manipuladores da cláusula. A propagação prosseguirá até que o chamador original seja encontrado, o qual, no caso de um programa aplicativo, é `main`. Se nenhum manipulador coincidente for encontrado em algum lugar, o programa será finalizado. Em muitos casos, os manipuladores de exceção incluem uma instrução `return` para finalizar o método no qual a exceção ocorreu.

Para assegurar que as exceções que podem ser geradas em uma cláusula `try` sejam sempre manipuladas em um método, pode-se escrever um manipulador especial que combine todas as exceções derivadas de `Exception` simplesmente definindo-o com um tipo de parâmetro `Exception`, como em

```
catch (Exception objetoGenerico) {  
    ...  
}
```

Uma vez que o nome da classe sempre coincide consigo mesmo ou com qualquer classe ancestral, qualquer derivada de `Exception` coincide com `Exception`. Obviamente, esse manipulador de exceções sempre deve ser colocado no final da lista de manipuladores, porque ele bloquearia o uso de qualquer manipulador que venha depois na construção `try` em que aparece. Isso porque a busca de um manipulador coincidente é sequencial e encerra-se quando uma coincidência é encontrada.

O parâmetro de objeto para um manipulador de exceção não é totalmente inútil, como poderia parecer até aqui nesta discussão. Durante a execução do programa, o sistema em tempo de execução Java armazena o nome da classe de todo objeto do programa. O método `getClass` pode ser usado para obter-se um objeto que armazena o nome da classe, o qual pode ser obtido com o método `getName`. Assim, podemos recuperar o nome da classe do parâmetro real a partir da instrução `throw` que causou a execução do manipulador. Para o manipulador acima, isso é feito com

```
objetoGenerico.getClass().getName()
```

A mensagem associada com o objeto de parâmetro, criado pelo construtor, pode ser obtida com

```
objetoGenerico.getMessage()
```

#### 14.5.5 Outras Opções de Projeto

A cláusula `throws` do Java tem uma aparência e uma colocação (no programa) similar à da especificação `throw` do C++. Porém, a semântica da `throws` difere completamente da semântica da cláusula `throw` do C++.

A ocorrência de um nome de classe `Exception` na cláusula `throw` de um método Java especifica que aquela ou quaisquer de suas descendentes podem ser geradas pelo método. Por exemplo, quando um método especifica que ele pode gerar `IOException`, significa que pode gerar um objeto `IOException` ou um objeto de qualquer uma de suas classes descendentes, como, por exemplo, `EOFException`.

As exceções da classe `Error` e `RuntimeException` e suas descendentes são chamadas **exceções não-verificadas**. Todas as outras são chamadas **exceções verificadas**. As primeiras nunca são uma preocupação do compilador. Porém, ele assegura que todas as exceções verificadas que um método pode gerar estejam listadas em sua cláusula `throws` ou manipuladas no método. A razão pela qual as exceções das classes `Error` e `RuntimeException` e suas descendentes não são verificadas é que qualquer método poderia gerá-las.

Um método não pode declarar mais exceções em sua cláusula `throws` do que o método ao qual ele se sobrepõe, embora possa declarar um número menor. Desse modo, se um método não tiver nenhuma cláusula `throws`, nenhum outro método que o sobreponha poderá ter tal cláusula. Um método pode gerar qualquer exceção listada em sua cláusula

Hidden page

```

    } /** fim da cláusula try interna
    catch (ArrayIndexOutOfBoundsException) {
        if (NovoGrau == 100)
            freq [9]++;
        else
            System.out.println("Erro - novo grau: " + NovoGrau +
                " está fora da faixa");
    } /** fim de catch (ArrayIndex...
    } /** fim de while (true) ...
} /** fim da cláusula try externa
catch(Excecaofimdados) {
    System.out.println ("\nFrequência dos limites\n");
    for (indice = 0; indice < 10; indice++) {
        limite_1 = 10 * indice;
        limite_2 = limite_1 + 9;
        if (indice == 9)
            limite_2 = 100;
        System.out.println(" " + limite_1 + " - " +
            limite_2 + " " + freq [indice]);
    } /** fim de for (indice = 0; ...
} /** fim de catch (Excecaofimdados ...
} /** fim do método Constroilista

```

A exceção de fim de dados `Excecaofimdados`, é definida no programa. Seu construtor exibe uma mensagem quando um objeto da classe é criado. Seu manipulador produz os dados de saída do método. A `ArrayIndexOutOfBoundsException` é predefinida e gerada pelo interpretador. Em ambos os casos, o manipulador não inclui um nome de objeto em seu parâmetro. Em nenhum dos casos, o nome serviria a qualquer propósito. Note que todos os manipuladores recebem objetos como parâmetros, que, freqüentemente, não são úteis.

#### 14.5.7 A Cláusula `finally`

Há algumas situações em que um processo deve ser executado independentemente de se a cláusula `try` gera ou não uma exceção e de se esta exceção é pega ou não em um método. Um exemplo dessa situação é um arquivo que deve ser fechado. Outro, é se o método tem algum recurso externo que deve ser liberado nele, independentemente de como a execução do mesmo é finalizada. A cláusula `finally` foi projetada para esses tipos de necessidades. Ela é colocada no fim da lista de manipuladores, logo depois de uma construção `try`. Em geral, a construção inteira assemelha-se a

```

try {
    ...
}
catch (...) {
    ...
}
... /** Mais manipuladores
finally {
}

```

A semântica dessa construção é a seguinte: se a cláusula **try** não gerar nenhuma exceção, a **finally** será executada antes que a execução prossiga depois da construção **try**. Se a **try** gerar uma exceção e ela for pega por um manipulador seguinte, a **finally** será executada depois do manipulador. Se a cláusula **try** gerar uma exceção, mas ela não for pega por um manipulador da construção **try**, a **finally** será executada antes que a exceção seja propagada.

Uma construção **try** sem nenhum manipulador de exceções pode ser seguida de uma cláusula **finally**. Isso somente faz sentido, é claro, se a instrução composta tiver uma instrução **break**, **continue** ou **return**. Nesses casos, sua finalidade é a mesma de quando ela é usada com manipulação de exceções. Por exemplo, poderíamos ter algo como o seguinte:

```
try {
    for (indice = 0; indice < 100; indice++) {
        ...
        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 a instrução **return** finalizar o laço ou dele se encerrar normalmente.

#### 14.5.8 Avaliação

Os mecanismos Java para manipulação de exceções oferecem uma melhoria em relação à versão C++ sobre a qual eles se basearam.

Primeiro, um programa C++ pode gerar qualquer tipo definido no programa ou pelo sistema. No Java, somente objetos que são instâncias de **Throwable** ou de alguma classe que descenda dela podem ser gerados. Isso separa os objetos que podem ser gerados de todos os outros objetos (e de não-objetos) que residem em um programa. Que significação pode ser dada a uma exceção que faz um valor **int** ser gerado?

Segundo, uma unidade de programa C++ que não inclui uma cláusula **throws** pode gerar qualquer exceção, a qual nada diz ao leitor. Um método Java sem uma cláusula **throws** não pode gerar nenhuma exceção verificada que ele não manipula. Portanto, o leitor de um método Java sabe, em função de seu cabeçalho, quais exceções ele poderia gerar, mas não manipular.

Terceiro, a adição da cláusula **finally** é uma grande conveniência em certas situações. Ela permite que tipos de ações de limpeza se desenvolvam, independentemente de como uma instrução composta foi finalizada.

Finalmente, o sistema Java, em tempo de execução, gera implicitamente uma variedade de exceções, como, por exemplo, para índices de vetor fora de faixa e para acessos a

Hidden page

**QUESTÕES DE REVISÃO**

1. Defina exceção, manipulador de exceções, levantamento de uma exceção, desativação de uma exceção e exceção incorporada.
2. Quais são as questões de projeto relativas à manipulação de exceções?
3. O que se quer dizer por uma exceção estar vinculada a um manipulador de exceções?
4. Qual é o problema com a vinculação de exceções a manipuladores da PL/I?
5. Quais são os quadros possíveis para exceções em Ada?
6. Onde as exceções não-manipuladas são propagadas em Ada se elas forem geradas em um subprograma? Em um bloco? Em um corpo de pacote? Em uma tarefa?
7. Onde a execução prossegue depois que uma exceção é manipulada em Ada?
8. Como uma exceção pode ser explicitamente gerada em Ada?
9. Como uma exceção definida pelo usuário é definida em Ada?
10. Como uma exceção pode ser suprimida em Ada?
11. Qual é o nome de todos os manipuladores de exceção do C++?
12. Como as exceções podem ser explicitamente geradas em C++?
13. Como as exceções são vinculadas a manipuladores em C++?
14. Como um manipulador de exceções pode ser escrito em C++ de maneira que manipule qualquer exceção?
15. Para onde vai o controle de execução quando um manipulador de exceções do C++ encerra sua execução?
16. O C++ inclui exceções incorporadas?
17. Qual é a classe-raiz de todas as classes de exceção Java?
18. Qual é a classe-pai da maioria das classes de exceção Java definidas pelo usuário?
19. Como um manipulador de exceções pode ser escrito em Java de maneira que manipule qualquer exceção?
20. Qual é a diferença entre uma especificação `throw C++` e uma cláusula `throws` Java?
21. Qual é a diferença entre exceções verificadas e não-verificadas em Java?
22. Como se pode desativar uma exceção Java?
23. Qual é o propósito da cláusula Java `finally`?

**PROBLEMAS**

1. Quais erros ou condições que, se for o caso, programas Pascal, em tempo de execução, podem detectar ou manipular?
2. Em livros didáticos sobre as linguagens de programação PL/I e Ada, pesquise os conjuntos respectivos de exceções incorporadas. Faça uma avaliação comparativa das duas, considerando tanto sua integridade como sua flexibilidade.
3. Escreva um segmento de código Ada que recupere uma chamada a um procedimento, `tape_read`, que leia dados de entrada de uma unidade de fita e possa gerar a exceção `tape_read_error`.
4. Em (AARM, 1995), determine como são manipuladas as exceções que se desenvolvem durante o *rendezvous*.
5. Em um livro didático sobre o COBOL, determine como é feita a manipulação de exceções em seus programas.
6. Em linguagens sem facilidades de manipulação de exceções, é comum que a maioria dos subprogramas inclua um parâmetro “erro”, que pode ser fixado em algum valor que represente “OK” ou algum outro valor que represente “erro de procedimento”. Quais vantagens uma facilidade de manipulação de exceções lingüística como a Ada tem sobre esse método?
7. Em uma linguagem sem facilidades de manipulação de exceções, poderíamos enviar um procedimento de manipulação de erros como um parâmetro a cada procedimento que possa detectar erros que devem ser manipulados. Quais desvantagens há nesse método?
8. Compare os métodos sugeridos nos Problemas 6 e 7. Qual deles você acha que é o melhor e por quê?
9. Compare as facilidades de manipulação de exceções do C++ com as da Ada. Qual projeto, em sua opinião, é o mais flexível? Qual deles possibilita escrever programas mais confiáveis?

10. Suponhamos que você esteja escrevendo um procedimento Ada com três métodos alternativos para cumprir suas exigências. Escreva uma versão esquemática deste procedimento de maneira que, se a primeira alternativa gerar alguma exceção, a segunda seja experimentada, e se também gerar uma exceção, a terceira seja executada. Escreva o código como se os três métodos fossem procedimentos chamados ALT1, ALT2 e ALT3.
11. Escreva um programa Ada que leia uma lista de valores inteiros na faixa de -100 a 100 pelo teclado e compute a soma dos quadrados dos valores lidos. Este programa deve usar manipulação de exceções para assegurar que os valores introduzidos estejam dentro da faixa e sejam números inteiros legais, para manipular o erro da soma dos quadrados que se tornar maior do que aquilo que uma variável INTEGER padrão pode armazenar, e detectar o fim de arquivo e usá-lo para acarretar a saída do resultado. No caso de estouro da soma, uma mensagem de erro deve ser impressa e o programa finalizado.
12. Escreva um programa C++ para a especificação do Problema 11.
13. Escreva um programa Java para a especificação do Problema 11.
14. Escreva uma comparação detalhada das capacidades de manipulação de exceções do C++ e do Java.
15. Considere o seguinte programa Java esquemático:

```

class Grande {
    int i;
    float f;
    void fun1() throws {int} {
        ...
        try {
            ...
            throw i;
            ...
            throw f;
            ...
        }
        catch(float) { ... }
        ...
    }
    class Pequena {
        int j;
        float g;
        void fun2() throws {float} {
            ...
            try {
                ...
                try {
                    Grande.fun1();
                    ...
                    throw j;
                    ...
                    throw g;
                    ...
                }
                catch (int) { ... }
                ...
            }
            catch (float) { ... }
        }
    }
}

```

Em cada uma das quatro instruções **throw**, onde a exceção é manipulada? Note que fun1 é chamada de fun2 na classe Pequena.

Hidden page

## Capítulo 15

# Linguagens de Programação Funcionais



- 15.1** Introdução
- 15.2** Funções Matemáticas
- 15.3** Fundamentos das Linguagens de Programação Funcionais
- 15.4** A Primeira Linguagem de Programação Funcional: LISP
- 15.5** Uma Introdução à Scheme
- 15.6** COMMON LISP
- 15.7** ML
- 15.8** Haskell
- 15.9** Aplicações das Linguagens Funcionais
- 15.10** Uma Comparação entre as Linguagens Funcionais e Imperativas

### **John McCarthy**

John McCarthy e Marvin Minsky formaram o *Artificial Intelligence Project* do MIT em 1958. Em 1958–1959, McCarthy projetou o LISP, que se tornou operacional em 1959. Serviu, também, na equipe de projeto do ALGOL.

Este capítulo apresenta a programação funcional e algumas das linguagens de programação projetadas para essa abordagem ao desenvolvimento de software. Uma vez que elas se baseiam em funções matemáticas, iniciamos nosso trabalho revisando as idéias fundamentais das mesmas, incluindo a notação funcional lambda de Church. Incluímos, também, nesta seção, uma breve discussão a respeito das formas funcionais e alguns exemplos das mais comuns. Em seguida, será introduzida a idéia de linguagem de programação funcional seguida de uma olhada na primeira linguagem funcional, o LISP, e suas estruturas de dados de lista e de sintaxe funcional, que se baseiam na notação lambda. A seção seguinte, bastante extensa, é dedicada a uma introdução à Scheme, incluindo algumas de suas funções primitivas, de suas formas especiais, de suas formas funcionais e de alguns exemplos de funções simples escritas em Scheme. Descreveremos, então, brevemente, alguns recursos imperativos da Scheme. Em seguida, apresentaremos breves introduções à COMMON LISP, à ML e à Haskell para mostrar algumas idéias de projeto diferentes (da Scheme) para linguagens de programação funcionais. Segue-se uma seção que descreve algumas das aplicações das linguagens de programação funcionais. Por fim, apresentaremos uma breve comparação entre as linguagens funcionais e imperativas.

## 15.1 Introdução

---

Os 14 primeiros capítulos deste livro preocuparam-se, principalmente, com as linguagens de programação imperativas e orientadas a objeto. Com exceção da Smalltalk, as linguagens orientadas a objeto discutidas têm formas semelhantes às imperativas.

O elevado grau de similaridade entre as linguagens imperativas provém, em parte, de uma de suas bases comuns de projeto: a arquitetura de von Neumann, conforme discutimos no Capítulo 1. Podemos pensar nas linguagens imperativas coletivamente como uma progressão de desenvolvimentos para melhorar o modelo básico, o FORTRAN 1. Todas foram projetadas para um uso eficiente de computadores com a arquitetura de von Neumann. Não obstante o estilo imperativo de programação ter sido considerado aceitável pela maioria dos programadores, o fato dele recorrer fortemente à arquitetura subjacente é visto, por alguns, como uma restrição desnecessária ao processo de desenvolvimento de software.

Existem outras bases de projeto de linguagem, muitas das quais orientadas mais a paradigmas e a metodologias de programação particulares do que à execução eficiente em uma arquitetura de computador particular. Até agora, entretanto, a reduzida eficiência ao executar programas escritos nessas linguagens tem impedido que elas se tornem tão populares quanto as linguagens imperativas.

O paradigma de programação funcional, baseado em funções matemáticas, é a base de projeto de um dos mais importantes estilos de linguagens não-imperativas. Esse estilo de programação é suportado pelas linguagens de programação funcionais ou aplicativas.

O LISP iniciou-se como uma linguagem puramente funcional, mas logo adquiriu alguns recursos imperativos importantes que aumentaram sua eficiência de execução. Ele ainda é a mais importante das linguagens funcionais, pelo menos em termos de ser a única que conseguiu obter um uso generalizado. A Scheme é um dialeto pequeno e de escopo estático do LISP. A COMMON LISP é um amálgama de diversos dialetos da década de 80 do LISP. A ML é uma linguagem funcional fortemente tipificada com uma sintaxe mais funcional do que o LISP ou do que a Scheme. A Haskell baseia-se parcialmente na ML, mas é uma linguagem puramente funcional.

Um objetivo deste capítulo é introduzir o conceito, mas não o processo, da programação funcional. Descreveremos também diversas maneiras pelas quais uma linguagem pode ser projetada para oferecer facilidades convenientes para programação funcional. Nossa abordagem é discutir as funções matemáticas e a programação funcional e, depois, introduzir um subconjunto da Scheme puramente funcional, para ilustrarmos o seu estilo. Incluímos um material suficiente sobre a Scheme para permitir que o leitor escreva alguns programas simples, mas interessantes. É difícil de adquirir uma compreensão real do que é programação funcional sem alguma experiência, de fato, em programação; assim, isso é fortemente encorajado.

## 15.2 Funções Matemáticas

Uma função matemática é uma **correspondência** de membros de um conjunto, chamado **conjunto domínio**, com outro, o **conjunto imagem**. Uma definição de função especifica ambos explicita ou implicitamente, juntamente com a correspondência, descrita por uma expressão ou, em alguns casos, por uma tabela. Funções freqüentemente são aplicadas a um elemento particular do conjunto domínio. Note que este pode ser o produto vetorial de diversos conjuntos. Uma função produz ou retorna um elemento do conjunto imagem.

Uma das características fundamentais das funções matemáticas é que a ordem de avaliação de suas expressões de correspondência é controlada por recursão e por expressões condicionais, não pela seqüência ou pela repetição iterativa comuns nas linguagens de programação imperativas.

Outra característica importante das funções matemáticas é que, uma vez que elas não têm efeitos colaterais, sempre definem o mesmo valor, dado o mesmo conjunto de argumentos. Os efeitos colaterais nas linguagens de programação estão ligados a variáveis que modelam as localizações da memória. Uma função matemática define um valor, em vez de especificar uma seqüência de operações sobre valores na memória para produzir um valor. Não existem variáveis no sentido das linguagens imperativas, de modo que não pode haver nenhum efeito colateral.

### 15.2.1 Funções Simples

As definições de funções freqüentemente são escritas como um nome de função, seguido de uma lista de parâmetros entre parênteses, seguida da expressão de correspondência. Por exemplo,

$$\text{cubo}(x) = x * x * x, \text{ em que } x \text{ é um número real}$$

Nessa definição, os conjuntos domínio e imagem são os números reais. O símbolo  $=$  é usado significando “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 nesse aspecto que os parâmetros das funções matemáticas diferem das variáveis nas linguagens imperativas.

As aplicações de funções são especificadas dispondendo-se em pares o nome da função com um elemento particular do conjunto domínio. O elemento da imagem é obtido avaliando-se a expressão de correspondência da função com o elemento do domínio substituído

nas ocorrências do parâmetro. Por exemplo, cubo (2,0) produz o valor 8,0. Novamente, é importante notar que, durante a avaliação, a correspondência de uma função não contém nenhum parâmetro desvinculado; um parâmetro vinculado é o nome de um valor particular. Toda ocorrência de um parâmetro é vinculada a um valor do conjunto domínio e considerada uma constante durante a avaliação.

Os primeiros trabalhos teóricos sobre funções separavam a tarefa de definir uma função da tarefa de nomeá-la. A notação lambda, conforme foi idealizada por Alonzo Church (Church, 1941), fornece um meio para definir funções sem nome. Uma **expressão lambda** especifica o parâmetro e a correspondência biunívoca de uma função. Ela é a própria função. Por exemplo, considere

$$\lambda(x)x * x * x$$

conforme afirmamos acima, antes da avaliação, um parâmetro representa qualquer membro do conjunto domínio, mas, durante a avaliação, é vinculado a um particular. Quando uma expressão lambda é avaliada para determinado parâmetro, diz-se que ela deve ser aplicada nele. A mecânica dessa aplicação é a mesma que para qualquer avaliação de função. A aplicação da expressão lambda acima é denotada como no seguinte exemplo:

$$(\lambda(x)x * x * x)(2)$$

que resulta no valor 8.

As expressões lambda, à semelhança de outras definições de função, podem ter mais de um parâmetro.

### 15.2.2 Formas Funcionais

Uma função de ordem superior ou **forma funcional** é aquela que toma funções como parâmetros, produz uma função como seu resultado, ou ambos. Um tipo comum de forma funcional é a **composição de funções**, que tem dois parâmetros funcionais e produz uma função cujo valor é a primeira função paramétrica real aplicada ao resultado da segunda. A composição de funções é escrita como uma expressão, usando  $\circ$  como operador, como em

$$h \equiv f \circ g$$

Por exemplo, se

$$f(x) \equiv x + 2$$

$$g(x) \equiv 3 * x$$

$h$  é definido como

$$h(x) \equiv f(g(x)), \text{ ou } h(x) \equiv (3 * x) + 2$$

A **construção** é uma forma funcional que toma uma lista de funções como parâmetros. Quando aplicada a um argumento, ela aplica cada um de seus parâmetros funcionais a esse argumento e coleta os resultados em uma lista ou seqüência. Uma construção é sintaticamente denotada colocando-se as funções entre colchetes, como em  $[f, g]$ . Considere o seguinte exemplo:

Digamos que

$$g(x) \equiv x * x$$

$$h(x) \equiv 2 * x$$

$$i(x) \equiv x / 2$$

então

$$[g, h, i](4) \text{ produz } (16, 8, 2)$$

**Apply-to-all** (Aplica-se a tudo) é uma forma funcional que toma uma única função como parâmetro. Se aplicada a uma lista de argumentos, *apply-to-all* aplica seu parâmetro funcional a cada um dos valores no argumento da lista e coleta os resultados em uma lista ou em uma seqüência. *Apply-to-all* é denotada por  $\alpha$ . Considere o seguinte exemplo:

Digamos que

$$h(x) = x * x$$

então

$$\alpha(h, (2, 3, 4)) \text{ produz } (4, 9, 16)$$

Há muitas outras formas funcionais, mas esses exemplos devem ilustrar suas características.

### 15.3 Fundamentos das Linguagens de Programação Funcionais

O objetivo do projeto de uma linguagem de programação funcional é imitar as funções matemáticas no maior grau possível. Isso resulta em uma abordagem à solução de problemas que difere fundamentalmente dos métodos usados com as linguagens imperativas. Em uma linguagem imperativa, uma expressão é avaliada e o resultado é armazenado em uma localização da memória, representada como uma variável em um programa. A atenção necessária para as células de memória resulta em uma metodologia de programação de nível relativamente baixo. Um programa em uma linguagem de montagem, muitas vezes, precisa armazenar os resultados de avaliações parciais de expressões. Por exemplo, para avaliar

$$(x + y)/(a - b)$$

o valor de  $(x + y)$  é computado primeiro. Ele deve ser armazenado enquanto  $(a - b)$  é avaliado. Para ajudar a aliviar esse problema, o compilador manipula o armazenamento de resultados intermediários de avaliações de expressões em linguagens de alto nível. O armazenamento de resultados intermediários ainda é necessário, mas os detalhes são ocultados do programador.

Uma linguagem de programação puramente funcional não usa variáveis ou instruções de atribuição. Isso libera o programador de preocupar-se com as células da memória do computador no qual o programa é executado. Sem variáveis, construções iterativas não são possíveis, porque elas são controladas por variáveis. A repetição deve ser feita por meio de recursão, não por meio de laços. Programas são definições de funções e de especificações de aplicação destas, e as execuções consistem em avaliá-las. Sem variáveis, a execução de um programa puramente funcional não tem nenhum estado em termos de semântica operacional e denotacional. A execução de uma função sempre produz o mesmo resultado quando dados os mesmos parâmetros. Isso se chama **transparência referencial**. Ela torna a semântica de linguagens puramente funcionais bem mais simples do que a semântica das imperativas e das funcionais que incluem recursos imperativos.

Uma linguagem funcional oferece um conjunto de funções primitivas, um conjunto de formas funcionais para construir funções complexas a partir das primitivas, uma operação de aplicação de funções e alguma estrutura ou estruturas para representar dados. Tais estruturas são usadas para representar os parâmetros e os valores computados pelas funções. Uma linguagem funcional bem definida requer somente um pequeno número de primitivas.

Ainda que as linguagens funcionais freqüentemente sejam implementadas com interpretadores, elas também podem ser compiladas.

As linguagens imperativas usualmente fornecem apenas um limitado suporte para programação funcional. A maioria, por exemplo, inclui algum tipo de definição de função e facilidades de representação. O inconveniente mais sério no uso de uma linguagem imperativa para fazer programação funcional é que as funções nas linguagens imperativas têm restrições sobre os tipos de valores que podem ser retornados. Em muitas linguagens, como o FORTRAN e o Pascal, somente valores de tipo escalar podem ser retornados. O mais importante é que elas não podem retornar uma função. Essas restrições limitam os tipos de formas funcionais que podem ser oferecidos. Outro problema sério com as funções das linguagens imperativas é a possibilidade de efeitos colaterais funcionais.

## 15.4 A Primeira Linguagem de Programação Funcional: LISP

---

Desenvolveu-se um grande número de linguagens de programação funcionais. A mais antiga e de uso generalizado é o LISP. Estudar as linguagens funcionais pelo LISP é bastante semelhante a estudar as imperativas a partir do FORTRAN: o LISP foi a primeira funcional, mas alguns acreditam, agora, que, não obstante ela ter se desenvolvido firmemente ao longo dos últimos 30 anos, não mais representa os conceitos de projeto mais recentes para esta sua definição. Além disso, com exceção da primeira versão, todos os dialetos LISP incluem características de linguagem imperativa, como, por exemplo, variáveis, instruções de atribuição e iteração ao estilo imperativo (variáveis no estilo imperativo são usadas para nomear células de memória, cujos valores podem modificar-se muitas vezes durante a execução do programa). Apesar disso e de sua forma bastante estranha, as descendentes do LISP original representam bem os conceitos fundamentais da programação funcional e, portanto, merecem ser estudadas.

### 15.4.1 Tipos e Estruturas de Dados

Havia somente dois tipos de objetos de dados no LISP original: átomos e listas. Eles não são tipos no sentido das linguagens imperativas. De fato, o LISP original era uma linguagem sem tipos. Os átomos, sob a forma de identificadores, são os símbolos do LISP. As constantes numéricas também são consideradas átomos.

Lembre-se do Capítulo 2 que o LISP originalmente usava listas como sua estrutura de dados porque elas eram consideradas uma parte essencial do processamento de listas. Da maneira como se desenvolveu por fim, entretanto, o LISP raramente requer as operações de inserção e de exclusão.

As listas são especificadas delimitando-se seus elementos entre parênteses. Os elementos de listas simples restringem-se a átomos, como em

(A B C D)

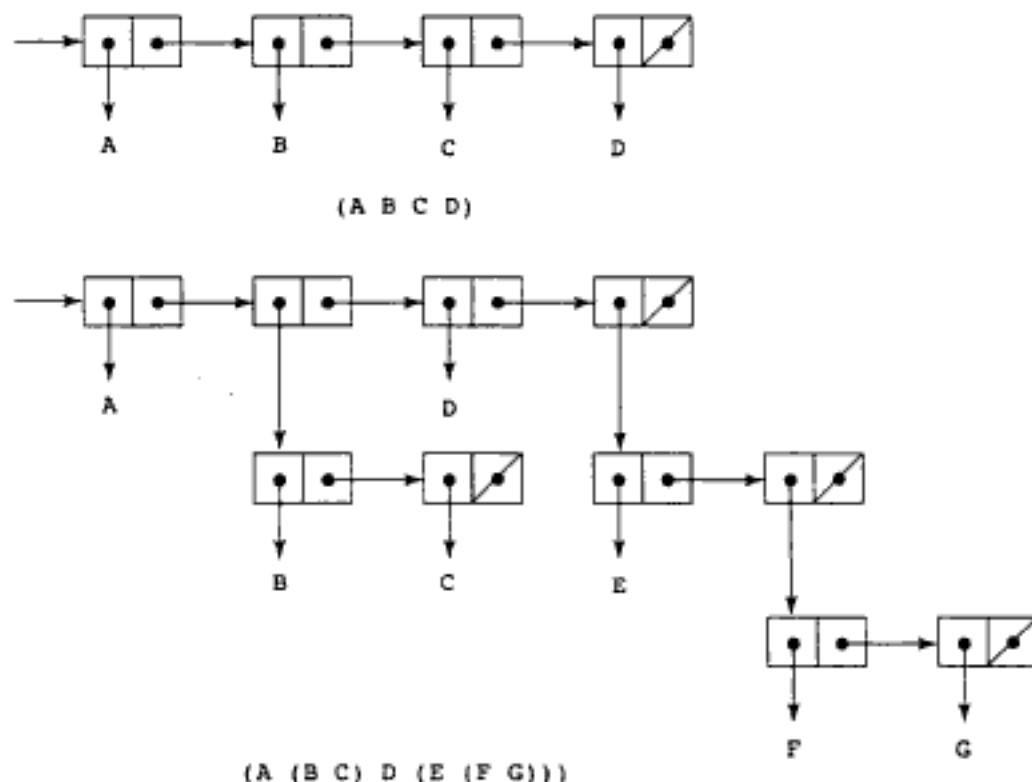
Estruturas de listas aninhadas também são especificadas por parênteses. Por exemplo, a lista

(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 seu segundo elemento a sublista (F G).

Internamente, as listas são armazenadas como estruturas de listas encadeadas simples, nas quais cada vértice tem dois ponteiros e representa um elemento. Um vértice de um átomo tem seu primeiro ponteiro apontando para alguma representação daquele, como, por exemplo, seu símbolo ou seu valor numérico. Um vértice de um elemento de sublista tem seu primeiro ponteiro apontando para o primeiro vértice da sublista. Em ambos os casos, o segundo ponteiro de um vértice aponta para o elemento seguinte da lista, que é referenciada por um ponteiro para o seu primeiro elemento.

A representação interna de nossos dois exemplos de lista é mostrada na Figura 15.1. Note que os elementos de uma lista são mostrados horizontalmente. O último não tem nenhum sucessor, de modo que seu vínculo é NIL. As sublistas são mostradas com a mesma estrutura.



**FIGURA 15.1** Representação interna de duas listas LISP.

### 15.4.2 O Primeiro Interpretador LISP

A intenção original era ter uma notação para programas LISP que estivesse o mais próximo possível do FORTRAN, com adições quando necessário. Tal notação foi chamada de notação M, de metanotação. Precisava haver um compilador que traduzisse os programas escritos em M para programas em código de máquina semanticamente equivalentes para o IBM 704.

No inicio do desenvolvimento do LISP, McCarthy decidiu redigir um documento que promovesse o processamento de lista como uma abordagem ao processamento simbólico geral. McCarthy acreditava que o processamento de lista podia ser usado para estudar a computabilidade, a qual, na época, usualmente era estudada usando-se máquinas Turing. McCarthy achava que o processamento de listas simbólicas era um modelo mais natural de computação do que as máquinas de Turing. Uma das exigências mais comuns do estudo da computação é que se deve ser capaz de provar certas características de computabilidade na classe inteira de quaisquer modelos computacionais usados. No caso do modelo de máquina Turing, pode-se construí-la com uma característica universal que pode imitar as operações de qualquer outra máquina de Turing. A partir desse conceito surgiu a idéia de construir uma função LISP universal que pudesse avaliar qualquer outra função em LISP.

A primeira exigência para a função LISP universal era uma notação que permitisse expressar as funções da mesma maneira que os dados eram expressos. A notação de lista entre parênteses descrita na Seção 15.4.1 já havia sido adotada para dados LISP, de modo que se decidiu inventar convenções para definição e chamada de função que poderiam ser expressas na notação de lista. As chamadas a função eram especificadas sob a forma de lista prefixada chamada Cambridge Polish, como no seguinte:

(nome\_da\_função argumento\_1 ... argumento\_n)

Por exemplo, se + for uma função que leva dois parâmetros numéricos,

(+ 5 7)

avalia-se como 12.

A notação lambda descrita na Seção 15.2.1 foi escolhida para especificar definições de funções. Ela teve de ser modificada, entretanto, para permitir a vinculação de funções a nomes, para que as funções pudessem ser referenciadas por outras funções e por elas próprias. Essa vinculação de nomes era especificada por meio de uma lista que consistia no nome da função e em uma outra contendo a expressão lambda, como em

(nome\_da\_função (LAMBDA (arg\_1... arg\_n) expressão))

Se você ainda não teve nenhum contato com a programação funcional, pode parecer estranho até mesmo considerar uma função sem-nome. Porém, elas, às vezes, são úteis na programação funcional (bem como na Matemática). Por exemplo, considere uma função cuja ação seja produzir uma função para aplicação imediata a uma lista de parâmetros. A função produzida não tem necessidade de um nome, porque ela é aplicada somente no ponto de sua construção. Esse exemplo é apresentado na Seção 15.5.6.

As funções LISP especificadas nessa nova notação eram chamadas expressões S, de simbólicas. Por fim, todas as estruturas LISP, tanto dados como código, eram expressões S. Esta última pode ser uma lista ou um átomo. Freqüentemente nos referiremos às S simplesmente como expressões.

McCarthy desenvolveu, com sucesso, uma função universal que poderia avaliar qualquer outra. Ela foi chamada de EVAL e estava na forma de uma expressão. Duas das pessoas

do projeto de IA\*, Stephen B. Russell e Daniel J. Edwards, notaram que uma implementação da EVAL poderia servir como um interpretador do LISP e construíram prontamente essa implementação (McCarthy et al., 1965).

Houve diversos resultados importantes nessa rápida, fácil e inesperada implementação. Primeiro, todos os sistemas LISP copiaram a EVAL e, portanto, eram interpretativos. Segundo, a definição da notação M, de programação planejada para o LISP, jamais foi concluída ou implementada, de modo que as expressões S tornaram-se a única notação do LISP. O uso da mesma notação para dados e para código tem importantes consequências, uma das quais será discutida na Seção 15.5.8. Terceiro, grande parte do projeto da linguagem original foi efetivamente congelado, mantendo certos recursos estranhos na linguagem, como, por exemplo, a forma de expressão condicional e o uso do zero tanto para endereço nulo como para o falso-lógico.

Outro recurso dos primeiros sistemas LISP, aparentemente acidental, era o uso do escopo dinâmico. As funções eram avaliadas nos ambientes de seus chamadores. Na época, ninguém sabia muita coisa sobre a definição de escopos e é duvidoso que se desse muita importância à opção. O escopo dinâmico era usado para a maioria dos dialetos do LISP antes de 1975. Os dialetos contemporâneos usam escopo estático ou permitem que o programador escolha entre este e o dinâmico.

## 15.5 Uma Introdução à Scheme

Nesta seção, descreveremos uma parte da Scheme (Dybvig, 1996). Nós a escolhemos por sua relativa simplicidade e por sua popularidade nos colégios e nas universidades; interpretadores da Scheme estão prontamente disponíveis para uma ampla variedade de computadores. Esta seção descreve sua versão 4.

### 15.5.1 Origens da Scheme

A linguagem Scheme, um dialeto do LISP, surgiu no MIT em meados da década de 70 (Sussman e Steele, 1975). Ela é caracterizada pelo pequeno tamanho, pelo uso exclusivo que faz do escopo estático e pelo tratamento que dá às funções como entidades de primeira classe. Como esta última, as funções Scheme podem ser os valores de expressões e elementos de listas, e podem ser atribuídas a variáveis e passadas como parâmetros. As primeiras versões do LISP não forneciam todas estas capacidades.

Como uma linguagem pequena com sintaxe e semântica simples, a Scheme adapta-se bem a aplicações educacionais, como, por exemplo, a cursos de programação funcional e também a introduções gerais à programação.

Note que a maioria das funções escrita em Scheme das seções seguintes exigiria que somente pequenas modificações fossem feitas nas funções LISP.

\*N. de R.T. IA é um acrônimo de Inteligência Artificial.

### 15.5.2 Funções Primitivas

Os nomes em Scheme podem consistir em letras, em dígitos e em caracteres especiais, exceto parênteses; não fazem distinção entre maiúsculas e minúsculas, mas não devem iniciar por um dígito.

O interpretador Scheme é um laço infinito de leitura-avaliação-escrita. Ele lê repetidamente uma expressão digitada pelo usuário (sob a forma de uma lista), interpreta-a e exibe o valor resultante. As literais avaliam a si mesmas. Desse modo, se você digitar um número para o interpretador, ele simplesmente o exibirá. Expressões que chamam funções primitivas são avaliadas da seguinte maneira: primeiro, cada um dos parâmetros da expressão é avaliado, sem nenhuma ordem particular. Depois, a função primitiva é aplicada aos valores dos parâmetros, e o valor resultante é exibido.

A Scheme inclui funções primitivas para as operações aritméticas básicas. São elas, o `+`, `-`, `*` e `/`, para adicionar, subtrair, multiplicar e dividir. `*` e `+` podem ter zero ou mais parâmetros. Se a `*` não for dado nenhum parâmetro, ele retornará 1; se a `+` não for dado nenhum parâmetro, ele retornará 0. `+` soma todos os seus parâmetros. `*` multiplica todos os seus. `/` e `-` podem ter dois ou mais parâmetros. No caso da subtração, todos, a não ser o primeiro parâmetro, são subtraídos do primeiro. A divisão é semelhante à subtração. Eis alguns exemplos:

Expressão	Valor
42	42
( <code>*</code> 3 7)	21
( <code>+</code> 5 7 8)	20
( <code>-</code> 5 6)	-1
( <code>-</code> 15 7 2)	6
( <code>-</code> 24 ( <code>*</code> 4 3))	12

`SQRT` retorna a raiz quadrada de seu parâmetro numérico, se o valor do parâmetro não for negativo.

A primitiva Scheme que descreveremos na seqüência é uma função utilitária exigida pela natureza da operação de aplicação de função Scheme `EVAL`. Esta última é a base de toda avaliação de função na Scheme, seja primitiva ou não. Ela é chamada para manipular a parte-avaliação da ação ler-avaliar-escrever do interpretador Scheme. Quando aplicada a uma função primitiva, `EVAL` avalia primeiro os parâmetros da função dada. Essa ação é necessária quando os parâmetros reais de uma chamada a função, por sua vez, são chamadas à função, o que freqüentemente é verdadeiro. Em algumas chamadas, entretanto, os parâmetros são elementos de dados, átomos ou listas, em vez de referências a função. Quando um parâmetro não é uma referência a função, obviamente ele não deve ser avaliado.

Por exemplo, suponhamos que temos uma função com dois parâmetros, um átomo e uma lista, cujo propósito seja determinar se o átomo dado está nela. Nem o átomo, nem a lista, devem ser avaliados; eles são dados literais que devem ser examinados. Para evitar avaliar um parâmetro, ele é fornecido, primeiro, como um parâmetro para a função primitiva `QUOTE`, a qual simplesmente o retorna sem modificações. Os exemplos seguintes ilustram `QUOTE`:

```
(QUOTE A) retorna A
(QUOTE (A B C)) retorna (A B C)
```

Hidden page

Os resultados das operações CONS são mostrados na Figura 15.2. Note que CONS é, em certo sentido, o inverso de CAR e de CDR. CAR e CDR desmontam uma lista, CONS constrói uma nova a partir de partes das que foram dadas. Os dois parâmetros para CONS tornam-se o CAR e o CDR da nova lista. Assim, se lis for uma lista,

`(CONS (CAR lis) (CDR lis))`

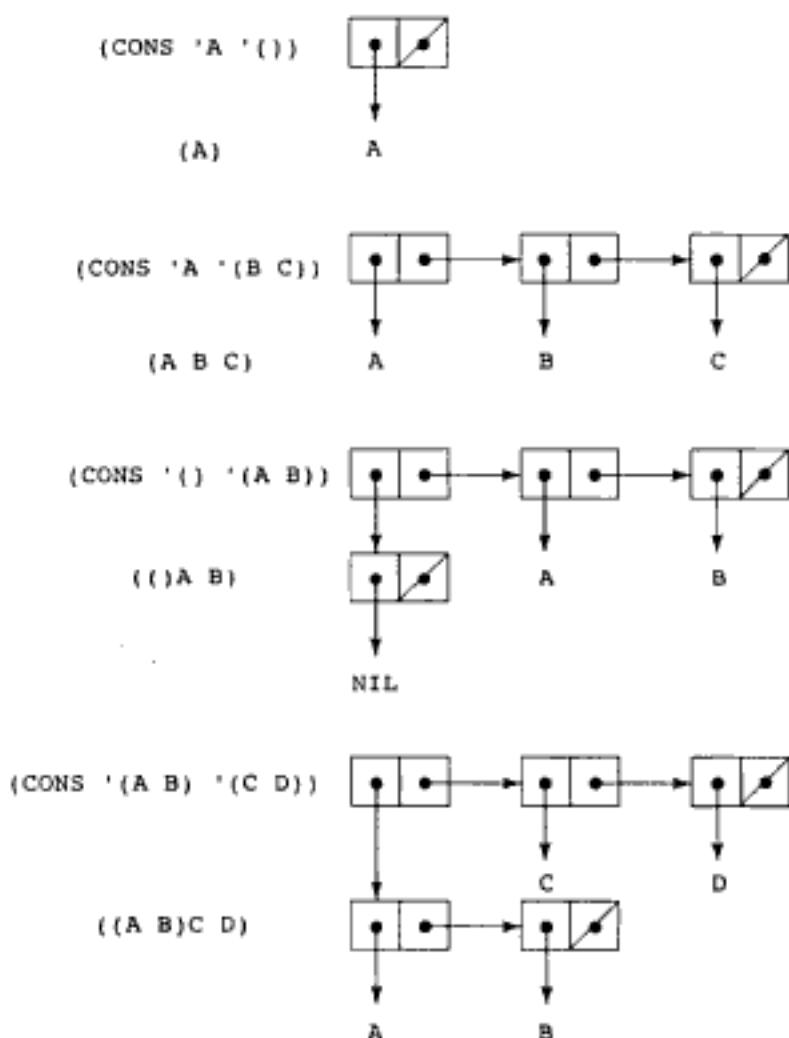
será a função de identidade.

LIST é uma função que constrói listas a partir de um número variável de parâmetros. Ela é uma versão abreviada de funções CONS aninhadas, como é ilustrado em

`(LIST 'maçã 'laranja 'uva)`

que equivale a

`(CONS 'maçã (CONS 'laranja (CONS 'uva '() )))`



**FIGURA 14.2** O resultado de diversas operações CONS.

Hidden page

Assim que o interpretador avaliar essa expressão, ela poderá ser usada, como em  
 (quadrado 5)

a qual exibe 25.

A semântica da forma especial `DEFINE` quando usada para definir uma função é a seguinte: a parte dos parâmetros do primeiro parâmetro e o segundo parâmetro inteiro são considerados juntos como uma expressão lambda. O nome do primeiro parâmetro é vinculado à expressão lambda.

Para ilustrar a diferença entre funções primitivas e a forma especial `DEFINE`, consideremos

```
(DEFINE x 10)
```

Se `DEFINE` fosse uma função primitiva, a primeira ação de `EVAL` sobre a expressão seria avaliar os dois parâmetros de `DEFINE`. Se `x` ainda não estivesse vinculado a um valor, isso seria um erro.

Como outro exemplo de função simples, consideremos

```
(DEFINE (segundo lst) (CAR (CDR lst)))
```

Nesse caso, o nome `segundo` é vinculado à expressão lambda

```
((LAMBDA (lst) (CAR (CDR lst))))
```

Assim que a função for avaliada, ela poderá ser usada, como em

```
(segundo '(A B C))
```

que retorna `B`.

A seguir, um exemplo mais simples de função. Ela calcula o tamanho da hipotenusa de um triângulo retângulo, dados os tamanhos dos outros lados.

```
(DEFINE (hipotenusa lado1 lado2)
  (SQRT (+(quadrado lado1) (quadrado lado2))))
)
```

Note que esta função usa a função `quadrado`, que foi definida anteriormente.

#### 15.5.4 Funções de Predicados

Três funções de predicado importantes entre as funções da Scheme são `EQ?`, `NULL?` e `LIST?`. Note que todas as funções de predicado predefinidas têm nomes que terminam com um ponto de interrogação. Uma função de predicado é aquela que retorna um valor booleano (verdadeiro ou falso). Em Scheme, os dois valores booleanos são `#T` e `#F`. O interpretador Scheme retorna a lista vazia, `()`, em vez de `#F`. Qualquer lista não-nula retornada por uma função de predicado é interpretada como `#T`.

A função `EQ?` assume dois parâmetros simbólicos. Ela retorna `#T` se ambos forem átomos e se os dois forem iguais; caso contrário, ela retornará `()`. Considere os seguintes exemplos:

```
(EQ? 'A 'A) retorna #T
(EQ? 'A 'B) retorna ()
```

```
(EQ? 'A '(A B)) retorna ()
(EQ? '(A B) '(A B)) retorna () ou #T
```

Como o último caso indica, o resultado de comparar listas com `EQ?` depende da implementação — algumas produzem `#T` e outras `()`. A razão para essa diferença é que `EQ?`, muitas vezes, é implementada como uma comparação de ponteiros (os dois apontam para o mesmo lugar?), e duas listas exatamente iguais freqüentemente não são duplicadas na memória. No momento que o sistema Scheme cria uma lista, ele confere se ela já existe. Se já existir, a nova lista nada mais será do que um novo ponteiro para a existente. Nesses casos, as duas serão consideradas iguais por `EQ?`. Porém, em alguns casos, pode ser difícil detectar a presença de uma lista idêntica, assim uma nova lista será criada. Nesse cenário, `EQ?` produzirá `()`.

Note que `EQ?` funciona para átomos simbólicos, mas não necessariamente para átomos numéricos. Seguem-se os predicados para comparar átomos numéricos. Conforme discutimos acima, `EQ?` também não funciona confiavelmente para parâmetros de lista.

A função de predicado `LIST?` retorna `#T` se seu argumento único for uma lista e, caso contrário, retorna `()`, como nos seguintes exemplos:

```
(LIST? '(X Y)) retorna #T
(LIST? 'X) retorna ()
(LIST? '()) retorna #T
```

A função `NULL?` testa seu parâmetro para determinar se ele é uma lista vazia e retorna `#T` se for. Considere os seguintes exemplos:

```
(NULL? '(A B)) retorna ()
(NULL? '()) retorna #T
(NULL? 'A) retorna ()
(NULL? '(( ))) retorna ()
```

O último caso é `()` porque o parâmetro não é uma lista vazia. Ao contrário, é uma lista que contém um único elemento, uma lista vazia.

A Scheme inclui uma coleção de funções de predicado para dados numéricos. Entre elas estão as seguintes:

<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?

Afirmamos anteriormente que `EQ?` funciona para átomos simbólicos, mas não necessariamente para átomos numéricos. O predicado `=` funciona para átomos numéricos, mas não para átomos simbólicos.

Às vezes, é conveniente ser capaz de testar dois átomos quanto à igualdade quando não se sabe se eles são simbólicos ou numéricos. Para esse propósito, a Scheme tem um

Hidden page

Hidden page

rados. Um processo similar pode ser realizado usando-se a recursão. A função pode comparar o átomo dado com o CAR da lista. Se eles coincidirem, o valor #T será retornado. Se não, o átomo somente poderá ser encontrado no restante da lista, de maneira que a função deverá chamar a si mesma, tendo o CDR daquela como parâmetro, e retornar o resultado dessa chamada recursiva. Nesse processo, há duas maneiras de sair da recursão: ou a lista está vazia em alguma chamada e () é retornado, ou uma coincidência foi encontrada e #T é retornado.

Ao todo, há três casos que devem ser manipulados na função: uma lista de entrada vazia, uma coincidência entre o átomo e o CAR da lista ou uma não-coincidência entre o átomo e o CAR da lista que causa a chamada recursiva. Esses três são exatamente os mesmos parâmetros para COND, sendo o último o caso padrão disparado por um predicado ELSE. Segue-se a função completa:

```
(DEFINE (membro atm lis)
  (COND
    ((NULL? lis) '())
    ((EQ? atm (CAR lis)) #T)
    (ELSE (membro atm (CDR lis))))
  ))
```

Essa forma é típica das funções de processamento de lista Scheme simples. Nessas funções, os dados das listas são processados, um elemento de cada vez. Os elementos individuais podem ser obtidos por CAR e o processo prossegue usando-se a recursão no CDR da lista.

Note que o teste nulo (*null*) deve preceder o de igualdade (*equal*), porque o CAR de uma lista vazia é um erro.

Como outro exemplo, considere o problema de determinar se duas listas dadas são iguais. Se elas forem simples, a solução será relativamente fácil, ainda que algumas técnicas pouco familiares estejam envolvidas. Uma função de predicado para comparar listas simples é mostrada aqui:

```
(DEFINE (igualsimples lis1 lis2)
  (COND
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) '())
    ((EQ? (CAR lis1) (CAR lis2))
      (igualsimples (CDR lis1) (CDR lis2)))
    (ELSE '()))
  ))
```

O primeiro caso, manipulado pelo primeiro parâmetro de COND, serve para quando o primeiro parâmetro é uma lista vazia. Isso pode ocorrer em uma chamada externa se o primeiro parâmetro de lista estiver inicialmente vazio. Uma vez que uma chamada recursiva usa os CDRs das duas listas de parâmetros como os seus, a primeira pode estar vazia nessa chamada se tiver tido todos os seus elementos removidos pelas chamadas recursivas anteriores. Quando isso ocorrer, deve-se verificar se a segunda lista também está vazia. Se estiver, elas serão iguais (ou inicialmente, ou quando os CARs eram iguais em todas as chamadas recursivas anteriores), e NULL? retornará corretamente #T. Se a segunda não estiver vazia, ela será maior do que a primeira e () deve ser retornado por NULL?. Lembre-se de que qualquer lista não-vazia retornada por uma função de predicado será interpretada como #T.

O caso seguinte trata da segunda lista estar vazia quando a primeira não estiver. Tal situação ocorre quando a primeira é maior do que a segunda. Somente a segunda lista deve ser testada, porque o primeiro caso cobre todas as instâncias da primeira lista estar vazia.

Hidden page

Hidden page

LET cria um novo escopo estático local de uma maneira muito semelhante ao `declare` da Ada. Novas variáveis podem ser criadas, usadas e depois descartadas quando o fim do novo escopo for alcançado. Os componentes nomeados de LET são como instruções de atribuição, mas podem ser usados somente no novo escopo de LET. Além disso, eles não podem ser novamente vinculados aos novos valores de LET.

LET é, de fato, apenas uma abreviação de uma expressão LAMBDA. As duas expressões seguintes 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 pelo parâmetro da expressão LAMBDA.

### 15.5.7 Formas Funcionais

Esta seção descreve duas formas funcionais matemáticas comuns fornecidas pela Scheme: composição e *apply-to-all*.

#### 15.5.7.1 Composição Funcional

Composição funcional é a única forma funcional primitiva oferecida pelo LISP original. Todos os dialetos subsequentes do LISP, inclusive a Scheme, também a oferecem. Ela é a essência de como a EVAL funciona. Todas as listas não-apostrofadas são interpretadas como chamadas a função, o que exige que seus parâmetros sejam avaliados primeiro. Isso se aplica recursivamente à menor lista de qualquer expressão, precisamente o que significa composição funcional. Os exemplos seguintes ilustram-na:

```
(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 de função nas chamadas internas não estão entre aspas porque devem ser avaliados, em vez de tratados como dados literais.

#### 15.5.7.2 Uma Forma Funcional Apply-to-All

As formas funcionais mais comuns oferecidas nas linguagens de programação funcionais são variações das formas funcionais *apply-to-all* matemáticas. A forma mais simples delas é mapcar, a qual tem dois parâmetros, uma função e uma lista. mapcar aplica a função dada a cada elemento da lista, e retorna uma lista dos resultados dessas aplicações. Uma definição Scheme de mapcar é a seguinte:

```
(DEFINE (mapcar fun lis)
  (COND
    ((NULL? lis) '())
    (ELSE (CONS (fun (CAR lis)) (mapcar fun (CDR lis))))
```

Hidden page

faz com que somador construa a lista

```
(+ 3 4 6)
```

A lista é, então, submetida a EVAL, que invoca + e retorna o resultado 13.

Em todas as versões anteriores da Scheme, a função EVAL avalia sua expressão no escopo mais externo do programa. A versão mais recente da Scheme, a 4, exige um segundo parâmetro para EVAL, que especifica o escopo no qual a expressão deve ser avaliada. Em nome da simplicidade, deixamos o parâmetro do escopo fora de nosso exemplo, e não discutimos seus nomes aqui.

### 15.5.9 Recursos Imperativos da Scheme

A Scheme, igual a outros dialetos contemporâneos do LISP, inclui diversos recursos empregados das linguagens imperativas. Por exemplo, nomes podem ser vinculados a valores e essas vinculações podem ser mudadas mais tarde. Isso é feito com a função SET!, como no seguinte:

```
(SET! pi 3.141593)
```

A função SET! retorna o valor que ela vincula.

Em uma versão puramente funcional do LISP, listas não podem ser mudadas. Elas podem ser separadas com CAR e CDR, mas nenhuma mudança é possível porque isso exigiria um recurso de linguagem imperativa — um efeito colateral — de uma chamada a função. A Scheme inclui duas funções que criam esses efeitos colaterais: SET-CAR! e SET-CDR!. Considere os seguintes exemplos:

```
(DEFINE lis (LIST 'A 'B))
(SET-CAR! lis 'C)
(SET-CDR! lis '(D))
```

SET-CAR! muda a lista vinculada a lis de (A B) para (C B). A SET-CDR! muda a lista (C B) para (C D).

Os recursos imperativos da Scheme acima descritos foram colocados nela em nome da eficiência, mas esses desvios da programação funcional também têm seus custos. A depuração e a manutenção dos programas tornam-se difíceis devido à possibilidade de apelidos e porque os efeitos colaterais permitem que chamadas a função idênticas produzam resultados diferentes em diferentes momentos. Por exemplo, considere o seguinte:

```
(DEFINE cont 0)
(DEFINE (cont_inc numero)
  (SET! cont (+ cont numero)))
)
```

Ainda que as duas chamadas seguintes a cont\_inc sejam idênticas, elas produzem resultados diferentes.

```
(cont_inc 1)
0
(cont_inc 1)
1
```

## 15.6 COMMON LISP

A COMMON LISP (Steele, 1984) surgiu de um esforço para combinar os recursos de diversos dialetos do LISP do início da década de 80, incluindo a Scheme, em uma única linguagem. Sendo uma combinação, ela é muito grande e complexa. Sua base, entretanto, é o LISP original, de modo que sua sintaxe, suas funções primitivas e sua natureza fundamental vêm dessa linguagem.

Reconhecendo a flexibilidade ocasional proporcionada pelo escopo dinâmico, bem como a simplicidade do escopo estático, a COMMON LISP permite ambos. O escopo padrão para variáveis é o estático, mas, ao declarar uma variável como “especial”, ela se torna de escopo dinâmico.

A lista de recursos da COMMON LISP é longa: um grande número de tipos de dados e de estruturas, incluindo registros, vetores, números complexos e cadeias de caracteres; operações de entrada e saída poderosas; uma forma de pacotes para modularizar coleções de funções e dados, e também oferecer controle de acesso; os recursos imperativos da Scheme, especificamente funções que fazem o que a `SET!`, `SET-CAR!` e `SET-CDR!` da Scheme fazem, mais o que é próprio dela.

A COMMON LISP, juntamente com a maioria dos dialetos do LISP (exceto a Scheme), inclui uma função chamada `PROG`, que permite o seqüenciamento de instruções, como é comum nas linguagens imperativas. Rótulos e as duas funções, `GO` e `RETURN`, são incluídos para proporcionar controle de iteração. `GO` é usada para transferir o controle para um rótulo dentro do escopo de `PROG`. `RETURN` é um meio de sair de `PROG`. A forma geral de `PROG` é

```
(PROG (variáveis locais)
      expressão_1
      ...
      expressão_n
    )
```

As variáveis locais são inicializadas em `NIL`, têm o escopo da `PROG` e existem somente durante a sua execução. Se houver nomes globais iguais aos locais, os globais não são afetados (e ocultos) em `PROG`. As expressões em `PROG` são átomos tratadas como rótulos. `GO` transfere o controle para seu parâmetro, que deve ser um rótulo dentro da lista de expressão `PROG`. `RETURN` tem um parâmetro, que passa a ser o valor de `PROG`.

Note que `PROG` é incluída nas versões contemporâneas do LISP somente para oferecer compatibilidade retrógrada com dialetos mais antigos. A COMMON LISP tem construções melhores para oferecer as capacidades de `PROG`. Por exemplo, a COMMON LISP tem construções `DOTIMES` e `DOLIST` para iteração e `PROG1`, `PROG2` e `PROGN` para construir seqüências.

`SETQ` é a função COMMON LISP que corresponde à `SET!` da Scheme e `DEFUN` é sua versão de `DEFINE`. Considere a seguinte versão iterativa da função de pertinência à lista. A versão iterativa segue-se uma recursiva similar à que apareceu na Seção 15.5.6.

```
(DEFUN membro_iterativo (atom lis)
  (PROG ()
    laco_1
    (COND
```

```

        ((NULL lis) (RETURN NIL))
        ((EQUAL atm (CAR lis)) (RETURN T))
    )
    (SETQ lis (CDR lis))
    (GO laco_1)
))

(DEFUN membro_recursivo (atm lis)
  (COND
    ((NULL lis) NIL)
    ((EQUAL atm (CAR lis)) T)
    (T (membro_recursivo atm (CDR lis))))
))

```

Note que T é a versão COMMON LISP do valor booleano verdadeiro, NIL é o valor booleano falso, ATOM é um predicado que determina se seu parâmetro é um átomo e uma lista nula pode ser considerada tanto uma lista como um átomo.

Como outro exemplo, considere as seguintes funções iterativas e recursivas que computam o tamanho de uma lista:

```

(DEFUN comprimento_iterativo (lis)
  (PROG (soma)
    (SETQ soma 0)
    novamente
    (COND
      ((ATOM lis (RETURN soma)))
    )
    (SETQ soma (+ 1 soma))
    (SETQ lis (CDR lis))
    (GO novamente)
))

(DEFUN comprimento_recursivo (lis)
  (COND
    ((NULL lis) 0)
    (T (+ 1 (comprimento_recursivo (CDR lis)))))
  )
)

```

Em certo sentido, a Scheme e a COMMON LISP são opostas. A primeira é bem menor e um pouco mais limpa, em parte devido ao uso exclusivo que faz do escopo estático. A COMMON LISP pretendia ser comercial e obteve sucesso como uma linguagem popularizada por aplicações de IA. A Scheme, por outro lado, é mais freqüentemente usada em cursos sobre programação funcional. Ela também tem mais probabilidade de ser estudada como uma linguagem funcional por causa de seu tamanho relativamente pequeno. Um critério de projeto importante da COMMON LISP que fez com que ela se tornasse muito grande é o desejo de torná-la compatível com diversos dialetos anteriores do LISP.

## 15.7 ML

A ML (Milner et al., 1990) é uma linguagem de programação funcional de escopo estático, como a Scheme. Ela difere do LISP e de seus dialetos, inclusive da Scheme, de muitas maneiras significativas. Usa uma sintaxe mais semelhante à do Pascal do que à do LISP. A ML tem declarações de tipo e usa inferência de tipos (o que significa que variáveis não precisam ser declaradas) e é fortemente tipificada. O tipo de cada variável e de expressão pode ser determinado durante a compilação. Isto contrasta fortemente com a Scheme, que é essencialmente sem tipo. A ML tem manipulação de exceções e uma facilidade modular para implementar tipos de dados abstratos. Uma breve história do desenvolvimento e dos principais recursos da ML é apresentada no Capítulo 2. O Capítulo 5 inclui uma introdução à idéia de inferência de tipos conforme ela aparece na ML.

Na ML, nomes podem ser vinculados a valores, tendo as instruções de declaração de valor a forma

```
val novo_nome = expressão
```

Por exemplo,

```
val distancia = tempo * velocidade;
```

Não fique com a idéia de que essa instrução é exatamente como as de atribuição das linguagens imperativas, porque não é. A instrução **val** vincula um nome a um valor, mas o nome não pode ser revinculado a um novo valor mais tarde. Bem, em certo sentido, pode. Realmente, se você revincular um nome com uma segunda instrução **val**, isso causará uma nova entrada no ambiente não relacionada à versão anterior do nome. De fato, seu tipo não precisa ser o mesmo. As instruções **val** não têm efeitos colaterais. Elas simplesmente adicionam um nome ao ambiente atual e vinculam-no a um valor, como a forma especial **LET** do LISP. O uso normal da **val** é 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 * raio * raio
end;
```

A ML tem listas e operações de lista, ainda que a aparência delas não seja como a do LISP. A ML também tem tipos enumerados, vetores e tuplas, que são registros.

As declarações de função na ML aparecem na forma geral

```
fun nome_da_função (parâmetros_formais) = expressão_do_corpo_da_função;
```

Por exemplo,

```
fun quadrado (x : int) = x * x;
```

Note que

```
fun quadrado (x) = x * x;
```

é ilegal, porque o tipo de  $x$  não pode ser determinado pelo compilador. Assim, funções de operadores aritméticos não podem ser polimórficas. O mesmo é verdadeiro em relação a funções que usam operadores relacionais, exceto  $=$  e  $\neq$ , e operadores booleanos. Porém, funções que usam somente operações de lista,  $=$ ,  $\neq$  e operadores de tupla (para formar tuplas e para seleção de componentes) podem ser polimórficas.

A construção de fluxo de controle de seleção ML é, de fato, uma expressão condicional com a forma

```
if E then expressão_then else expressão_else
```

$E$  deve ser avaliada em um valor booleano. Somente uma das duas outras expressões são avaliadas.

Não existe nenhuma coerção de tipos na ML; os tipos dos operandos de um operador ou de atribuição simplesmente devem coincidir para evitar erros de sintaxe.

## 15.8 Haskell

A Haskell (Thompson, 1996) é similar à ML em razão de usar uma sintaxe semelhante; tem escopo estático, é fortemente tipificada e usa o mesmo método de inferência de tipos. Ela difere da ML em termos de ser puramente funcional; não tem variáveis e nenhuma instrução de atribuição. De fato, a Haskell não permite nenhum efeito colateral e não inclui nenhum recurso imperativo de qualquer tipo. Isso a separa de quase todas as outras linguagens de programação. Duas outras características separam a Haskell da ML. Primeiro, ela usa uma técnica de avaliação diferente, chamada **avaliação preguiçosa**, na qual não se avalia nenhuma subexpressão até que seu valor seja reconhecido como necessário. Segundo, a Haskell tem um método para definir listas que permite infinitas listas. Essas são chamadas **abrangências de lista**. Alguns dos recursos da Haskell originaram-se na linguagem Miranda (Turner, 1986).

O código dessa seção foi escrito na versão 1.4 da Haskell.

Considere a seguinte definição da função factorial. Note que a sintaxe da definição e da aplicação de função define-se no nome desta simplesmente escrito próximo aos parâmetros.

```
fat 0 = 1
fat n = n * fat (n - 1)
```

Essa definição mostra que as definições de função podem incluir mais de uma linha, que definem versões de funções para diferentes formas de parâmetros reais. O valor da expressão de função apropriado (lado direito) é escolhido pelo padrão que coincide os parâmetros reais com os formais. Parâmetros formais constantes obviamente coincidem consigo mesmos nos parâmetros reais. Um nome em um padrão de parâmetro formal coincide com um parâmetro real não coincidente com um padrão constante. O valor do parâmetro real coincidido é, então, usado como o nome do valor na expressão correspondente no lado direito. A função acima definida é parcial porque não pode determinar um valor para parâmetros negativos.

Usando coincidência de padrões, podemos definir uma função para computar o enésimo número Fibonacci com o seguinte:

```

fib 0 = 1
fib 1 = 1
fib (n + 2) = fib (n + 1) + fib n

```

Guardas podem ser adicionadas às linhas de uma definição de função para especificar as circunstâncias sob as quais ela pode ser aplicada. Por exemplo,

```

fat n
| n == 0 = 1
| n > 0 = n * fat(n - 1)

```

Essa é uma definição mais precisa de fatorial do que a nossa anterior, porque ela restringe a faixa de valores de parâmetros reais àqueles para os quais funciona. A coincidência de padrões falharia, é claro, nesse uso, porque o padrão de parâmetros é *n* para ambas as expressões de valor. Tal forma de definição de função chama-se *expressão condicional*.

Uma **otherwise** pode aparecer como a última condição em uma expressão condicional, com a semântica óbvia. Por exemplo,

```

fun n
| n < 10      = 0
| n > 100     = 2
| otherwise   = 1

```

As listas são escritas entre colchetes, como em

```
cores = [ "azul", "verde", "vermelho", "amarelo" ]
```

A Haskell inclui uma coleção de operadores de lista. Por exemplo, as listas podem ser concatenadas com **++**, o **:** serve como uma versão infixada de **CONS** e **..** é usado para especificar séries aritméticas. Por exemplo,

```

5:[2, 7, 9] resulta em [5, 2, 7, 9]
[1, 3..11] resulta em [1, 3, 5, 7, 9, 11]
[1, 3, 5] ++ [2, 4, 6] resulta em [1, 3, 5, 2, 4, 6]

```

Dois exemplos de funções que operam em listas são

```

soma [] = 0
soma (a:x) = a + soma x

produto [] = 1
produto (a:x) = a * produto x

```

Em ambos, **a:x** especifica a lista com um **CAR**, ou cabeça, como **a** e um **CDR**, ou cauda, como **x**. **soma** retorna a soma dos elementos de uma lista dada. **produto** retorna o produto dos elementos de uma lista dada. Tanto **soma** como **produto** são funções Haskell padrão.

Usando **produto**, uma função fatorial pode ser escrita na forma mais simples

```
fat n = produto [1..n]
```

A função **length** retorna um número de elementos de uma lista dada. Por exemplo,

```
length(cores) retorna 4
```

Em Haskell, uma cláusula **where** é semelhante à **let** e à **val** da ML, exceto que as vinculações são dadas depois das expressões que as usam. Por exemplo, poderíamos escrever

```
raizes a b c =
```

```
[menos_b_sobre_2a - parte_quadrática_sobre_2a,
 menos_b_sobre_2a + parte_quadrática_sobre_2a]
where
    menos_b_sobre_2a = - b / (2.0 * a)
    parte_quadrática_sobre_2a =
        sqrt(b ^ 2 - 4.0 * a * c) / (2.0 * a)
```

As abrangências das listas constituem um método para descrever listas que representam conjuntos. A sintaxe da abrangência de lista é a mesma que, muitas vezes, é usada para descrever conjuntos em 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. Lê-se: "uma lista de todos os  $n \cdot n \cdot n$ , tal que  $n$  é tomado da faixa entre 1 e 50". Nesse caso, o qualificador está na forma de um **gerador**. Ele gera os números de 1 a 50. Em outros casos, os qualificadores estão na forma de expressões booleanas e são chamados de **testes**. Tal notação pode ser usada para descrever algoritmos para fazer muitas coisas, como localizar permutações de listas e classificá-las. Por exemplo, considere a seguinte função, a qual, quando dado um número **n**, retorna uma lista de todos os seus fatores:

fatores n = [ i | i ← [1..n div 2], n mod i == 0 ]

Em seguida, considere a concisão da Haskell, mostrada na seguinte implementação do algoritmo quicksort:

```
sort [] = []
sort (a:x) = sort [b | b <- x, b ≤ a]
             ++
             [a]
             ++
             sort [b | b <- x, b > a]
```

Essa definição de quicksort é significativamente mais curta do que o mesmo algoritmo codificado em uma linguagem imperativa.

Retornaremos, agora, ao tópico da avaliação preguiçosa. Lembre-se de que, na Scheme, os parâmetros para uma função são plenamente avaliados antes que ela seja chamada. Avaliação preguiçosa significa que os parâmetros para funções são avaliados somente quando isso é necessário. Assim, se uma função tiver dois parâmetros, mas em uma execução particular desta o primeiro não é usado, o parâmetro real passado para essa execução não será avaliado. Além disso, se somente uma parte de um parâmetro real precisar ser avaliada para a execução da função, o resto também não será avaliado. Finalmente, os parâmetros reais são avaliados somente uma vez, se forem.

O fato de uma linguagem usar a avaliação preguiçosa abre algumas possibilidades interessantes. Uma delas é a definição de estruturas de dados infinitas. Por exemplo, considere o seguinte:

```
positivos = [0...]
pares = [2, 4...]
quadrados = [n * n | n ← {0...}]
```

Evidentemente, nenhum computador pode representar de fato todos os números dessas listas, mas isso não impede sua aplicação se for usada a avaliação preguiçosa. Por exemplo,

Hidden page

Dentro da IA, um grande número de áreas foi desenvolvido, principalmente pelo uso do LISP. Ainda que outros tipos de linguagens pudessem ser usados — principalmente linguagens de programação lógica — a maioria dos sistemas especialistas existentes, por exemplo, foi desenvolvida em LISP. Ele também domina nas áreas de representação do conhecimento, de aprendizagem de máquina, de processamento de linguagem natural, de sistemas de treinamento inteligente e de modelagem da fala e da visão.

Fora da IA, o LISP também foi bem-sucedido. Por exemplo, o editor de texto EMACS é escrito em LISP, assim como o sistema de matemática simbólica, MACSYMA, que faz cálculo simbólico, entre outras coisas. A máquina LISP é um computador pessoal cujo software de sistemas inteiro é escrito em LISP. Ele também foi usado de maneira bem-sucedida para construir sistemas experimentais em uma variedade de áreas de aplicação.

A Scheme é bastante usada para ensinar programação funcional. Ela também é usada em algumas universidades para ministrar cursos de programação introdutórios. O uso da ML e da Haskell tem restringido-se, em sua maior parte, a laboratórios de pesquisa e universidades.

## 15.10 Uma Comparação entre as Linguagens Funcionais e Imperativas

Cabe, agora, uma breve discussão das vantagens — algumas amplamente aceitas e algumas somente amplamente conjecturadas — da programação funcional e de suas linguagens.

É natural compararmos a programação funcional com a programação em linguagens imperativas. Uma vez que as imperativas baseiam-se diretamente na arquitetura de von Neumann, os programadores que as usam devem lidar com a administração de variáveis e com a atribuição de valores a elas. O resultado disso é o aumento da eficiência de execução, mas a laboriosa construção de programas. Em uma linguagem funcional, o programador não precisa preocupar-se com variáveis, porque células de memória não precisam ser abstraídas na linguagem. Um resultado disso é a queda na eficiência de execução. Outro resultado, entretanto, é um nível de programação mais elevado, que deve exigir menos mão-de-obra do que programar em uma linguagem imperativa. Muitos acreditam que isso acontece e que constitui uma vantagem definitiva da programação funcional.

As linguagens funcionais podem ter uma estrutura sintática muito simples. A estrutura de lista do LISP é um exemplo. A sintaxe das linguagens imperativas é muito mais complexa. A semântica das funcionais também pode ser simples em comparação com a das imperativas.

A execução concorrente nas linguagens imperativas é difícil de projetar e de usar. Por exemplo, considere o modelo de tarefas da Ada, no qual a cooperação entre tarefas concorrentes cabe ao programador. Programas funcionais podem ser executados convertendo-os primeiro em grafos que poderão, então, ser executados por um processo de redução de grafo, o que pode ser feito com um bocado de concorrência que não foi especificada pelo programador. A representação por meio de grafos naturalmente expõe muitas oportunidades para a execução concorrente. A sincronização de cooperação neste processo não é uma preocupação do programador. Uma descrição adicional deste processo está além do escopo deste livro.

Em uma linguagem imperativa, o programador deve fazer uma divisão estática do programa em suas partes concorrentes, então escritas como tarefas. Este pode ser um processo complicado. Os programas em linguagens funcionais podem ser divididos dinamica-

Hidden page

## NOTAS BIBLIOGRÁFICAS

A primeira versão publicada do LISP pode ser encontrada em McCarthy (1960). Uma versão popular de meados da década de 60 até o final da década de 70 é descrita em McCarthy et al. (1965) e Weissman (1967). A COMMON LISP é descrita em Steele (1984). A linguagem Scheme, juntamente com algumas de suas inovações e vantagens, é discutida em Rees e Clinger (1986). Dybvig (1996) é uma boa fonte de informação sobre programação em Scheme. A ML é definida em Milner et al. (1990). Ullman (1994) é um excelente texto didático introdutório para a ML. A programação em Haskell é apresentada em Thompson (1996).

Uma discussão rigorosa da programação funcional em geral pode ser encontrada em Henderson (1980). O processo de implementar linguagens funcionais através de redução de grafos é discutido detalhadamente em Peyton Jones (1987).

## QUESTÕES DE REVISÃO

1. Defina forma funcional e transparência referencial.
2. Quais tipos de dados faziam parte do LISP original?
3. Qual é a diferença entre EQ?, EQV? e =?
4. Quais são as diferenças entre o método de avaliação usado para a forma especial da Scheme, DEFINE, e a usada para suas funções primitivas?
5. Quais são as duas formas de DEFINE?
6. Descreva a semântica de COND.
7. Descreva a semântica de LET.
8. Por que recursos imperativos foram adicionados à maioria dos dialetos do LISP?
9. Sob quais aspectos a COMMON LISP e a Scheme são opostas?
10. Qual regra de escopo é usada na Scheme? Na COMMON LISP? Na ML? Na Haskell?
11. Quais são as três características que deixam a ML muito diferente da Scheme?
12. O que é inferência de tipo, quando usada na ML (veja o Capítulo 5)?
13. Quais são os três recursos da Haskell que a tornam muito diferente da Scheme?
14. O que significa avaliação preguiçosa?

## PROBLEMAS

1. Escreva uma função Scheme que retorne seu parâmetro de lista simples em ordem reversa.
2. Escreva uma função de predicado em Scheme que teste a igualdade estrutural de duas listas dadas. Ambas são estruturalmente iguais se tiverem a mesma estrutura de lista, não obstante seus átomos poderem ser diferentes.
3. Escreva uma função Scheme que retorne a união de dois parâmetros de lista simples que representam conjuntos.
4. Escreva uma função Scheme com dois parâmetros, um átomo e uma lista, que retorne a lista com todas as ocorrências, não importa quão profundas, do átomo dado excluído. A lista retornada não pode conter nada em lugar dos átomos excluídos.
5. Escreva uma função Scheme que tome uma lista como parâmetro e retorne-a com o segundo elemento de nível máximo removido. Se a lista dada não tiver dois elementos, a função deve retornar () .
6. Leia o documento de John Backus sobre a FP (Backus, 1978) e compare os recursos da Scheme discutidos neste capítulo com os recursos correspondentes da FP.
7. Encontre definições das funções Scheme EVAL e APPLY, e explique suas ações.
8. Um dos mais modernos e completos ambientes de programação para qualquer linguagem é o sistema INTERLISP para LISP, conforme é descrito em "The INTERLISP Programming Environment", de Teitelman e Masinter (IEEE Computer, Vol. 14, No. 4, abril de 1981). Leia esse artigo cuidadosamente e compare a dificuldade de escrever programas LISP em seu sis-

tema com aquele que usa o INTERLISP (supondo que você normalmente não use o INTERLISP).

9. Consulte um livro sobre programação LISP e determine quais argumentos sustentam a inclusão do recurso PROG no LISP.
10. Uma linguagem funcional poderia usar alguma estrutura de dados que não seja uma lista. Por exemplo, ela poderia usar seqüências de símbolos. Quais primitivas essa linguagem teria em lugar das primitivas CAR, CDR e CONS da Scheme?
11. O que a seguinte função Scheme faz?

```
(define (y s lis)
  (cond
    ((null? lis) '())
    ((equal? s (car lis)) lis)
    (else (y s (cdr lis))))
  ))
```

12. O que a seguinte função 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)))))
  ))
```

## Capítulo 16

# Linguagens de Programação Lógicas



### **Robert Kowalski**

Robert Kowalski, da University of Edinburgh, é pesquisador em inteligência artificial. Kowalski, juntamente com Alain Colmerauer e Phillippe Roussel, da University of Aix-Marseille, desenvolveu a primeira linguagem de programação lógica, o Prolog.

- 16.1** Introdução
- 16.2** Uma Breve Introdução ao Cálculo de Predicados
- 16.3** Cálculo de Predicados e Demonstração 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 da Programação Lógica
- 16.9** Conclusões

Os objetivos deste capítulo são apresentar os conceitos da programação lógica e as suas linguagens, incluindo uma breve descrição de um subconjunto do Prolog. Iniciamos com uma introdução ao cálculo de predicado, a base para as linguagens de programação lógicas. Segue-se uma discussão de como ele pode ser usado para sistemas automáticos de demonstração de teoremas. Depois, apresentaremos uma visão geral da programação lógica. Em seguida, uma seção extensa apresenta os conceitos básicos do Prolog, incluindo cálculos aritméticos, processamento de lista e o uso de uma ferramenta de rastreamento que pode ser usada para depurar programas e também para ilustrar como o sistema Prolog funciona. As duas últimas seções descrevem alguns dos problemas do Prolog como uma linguagem lógica e algumas das áreas de aplicação nas quais ele tem sido usado.

## **16.1 Introdução**

---

O Capítulo 15 discutiu o paradigma da programação funcional, que é significativamente diferente das metodologias de desenvolvimento de software usadas com as linguagens imperativas. Neste capítulo, descreveremos outra metodologia de programação. Neste caso, a abordagem é expressar programas na forma de lógica simbólica e usar um processo de inferência lógica para produzir resultados. Os programas lógicos são declarativos em vez de baseados em procedimentos, o que significa que somente as especificações dos resultados desejados são declarados em vez de procedimentos detalhados para produzi-los.

A programação que usa uma forma de lógica simbólica como linguagem freqüentemente é chamada **programação lógica** e as linguagens baseadas na lógica simbólica são chamadas **linguagens de programação lógicas** ou **linguagens declarativas**. Optamos por descrever o Prolog porque é a única linguagem lógica de uso generalizado.

A sintaxe das linguagens de programação lógica são notavelmente diferentes da sintaxe das linguagens imperativas e das funcionais. A semântica dos programas lógicos também sustenta pouca semelhança com a dos programas em linguagem imperativa. Essas observações devem levar o leitor a ter certa curiosidade sobre a natureza da programação lógica e das linguagens declarativas.

## **16.2 Uma Breve Introdução ao Cálculo de Predicados**

---

Antes de podermos discutir a programação lógica, precisamos investigar brevemente sua base, a lógica formal. Esse não é nosso primeiro contato com a lógica formal neste livro; ela foi usada extensamente na semântica axiomática descrita no Capítulo 3.

Uma **proposição** pode ser imaginada como uma declaração lógica que pode ou não ser verdadeira. Ela consiste em objetos e nas relações de objetos entre si. A lógica formal foi desenvolvida para fornecer um método para descrever proposições, com a meta de permitir que essas proposições formalmente declaradas fossem verificadas quanto à validade.

A **lógica simbólica** pode ser usada para as três necessidades básicas da lógica formal: expressar proposições, expressar as relações entre essas e descrever como novas proposições podem ser inferidas de outras que se presumem verdadeiras.

Há uma estreita relação entre a lógica formal e a matemática. De fato, grande parte da matemática pode ser imaginada em termos de lógica. Os axiomas fundamentais da teoria dos números e dos conjuntos são o conjunto inicial de proposições presumidas como verdadeiras. Teoremas são as proposições adicionais que podem ser inferidas do conjunto inicial.

A forma particular de lógica simbólica usada para a programação lógica é o **cálculo de predicados**. Nas subseções seguintes, apresentaremos um breve exame dele. Nossa meta é lançar as bases para a discussão da programação lógica e do Prolog.

### 16.2.1 Proposições

Os objetos nas 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 que pode representar diferentes objetos em diferentes tempos, ainda que em um sentido bem mais próximo da matemática do que as variáveis em uma linguagem de programação imperativa.

As proposições mais simples, chamadas **proposições atômicas**, consistem em termos compostos. Um **termo composto** é um elemento de uma relação matemática, escrito em uma forma com a aparência de notação de função matemática. Lembre-se do Capítulo 15 que uma função matemática é uma correspondência, que pode ser representada como uma expressão ou como uma tabela ou lista de tuplas. Desse modo, termos compostos são elementos da definição tabular de uma função.

Um termo composto é formado por duas partes: um **functor**, o símbolo de função que nomeia a relação, e uma lista ordenada de parâmetros. 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 duas proposições

```
homem(jake)
gosta(bob, bife)
```

declarando que {jake} é uma 1-tupla na relação chamada homem, e que {bob, bife} é uma 2-tupla na relação chamada gosta. Se adicionássemos a proposição

```
homem(fred)
```

às duas proposições acima, a relação homem teria dois elementos distintos, {jake} e {fred}. Todos os termos simples dessas proposições — homem, jake, gosta, bob e bife — são constantes. Note que tais proposições não têm nenhuma semântica intrínseca. Elas significam qualquer coisa que quisermos que signifiquem. Por exemplo, o segundo exemplo acima pode significar que bob gosta de bife, ou que bife gosta de bob, ou que bob é de alguma maneira semelhante a um bife.

As proposições são declaradas em dois modos: um, em que a proposição é definida como verdadeira, e outro, em que a verdade da proposição é algo que precisa ser determinado. Em outras palavras, as proposições podem ser declaradas como fatos ou como consultas. Os exemplos de proposições acima poderiam ser ambos.

As proposições compostas têm duas ou mais proposições atômicas, ligadas por conectores lógicos ou operadores, da mesma maneira que as expressões lógicas compostas são construídas nas linguagens imperativas. Os nomes, os símbolos e os significados dos conectores lógicos do cálculo de predicados são os seguintes:

Nome	Símbolo	Exemplo	Significado
negação	$\neg$	$\neg a$	não $a$
conjunção	$\cap$	$a \cap b$	$a$ e $b$
disjunção	$\cup$	$a \cup b$	$a$ ou $b$
equivalência	$\equiv$	$a \equiv b$	$a$ é equivalente a $b$
implicação	$\supset$	$a \supset b$	$a$ implica $b$
	$\subset$	$a \subset b$	$b$ implica $a$

Apresentamos a seguir 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 elevada. Os operadores  $\cap$ ,  $\cup$  e  $\equiv$  têm, todos, maior precedência do que  $\supset$  e  $\subset$ . Assim, o segundo exemplo acima é equivalente a

$$(a \cap (\neg b)) \supset d$$

Variáveis podem aparecer em proposições, mas somente quando introduzidas por símbolos especiais, chamados quantificadores. O cálculo de predicados inclui dois quantificadores, conforme descreveremos abaixo, em que  $X$  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 $P$ seja verdadeiro

O ponto final (.) entre  $X$  e  $P$  simplesmente separa a variável da proposição. Por exemplo, considere o seguinte:

$$\begin{aligned} &\forall X.(\text{mulher}(X) \supset \text{humano}(X)) \\ &\exists X.(\text{mãe}(\text{mary}, X) \cap \text{homem}(X)) \end{aligned}$$

A primeira dessas proposições significa que para qualquer valor de  $X$ , se  $X$  for uma mulher,  $X$  é humano. A segunda significa que existe um valor de  $X$  tal que mary é a mãe de  $X$  e  $X$  é um homem; em outras palavras, mary tem um filho. O escopo dos quantificadores universais e existenciais são as proposições atômicas às quais eles são anexados. Ele pode ser estendido usando-se parênteses, como nas duas proposições compostas que acabamos de descrever. Desse modo, os quantificadores universais e existenciais têm precedência mais elevada do que qualquer um dos operadores.

### 16.2.2 Forma Clausal

Estamos discutindo o cálculo de predicados porque ele é a base para as linguagens de programação lógicas. Como acontece com outras, as linguagens lógicas são melhores em sua forma mais simples, significando que se deve minimizar a redundância.

Um problema com o cálculo de predicados conforme o descrevemos até aqui é que há muitas maneiras diferentes de declarar proposições com o mesmo significado; ou seja, há muita redundância. Isso não se constitui em um problema para os lógicos, mas se o cálculo de predicados precisar ser usado em um sistema automatizado (computadorizado), será um sério problema. Para simplificar as coisas, uma forma padrão de proposições é desejável. A clausal, uma forma relativamente simples de proposições, é uma dessas formas pa-

drão. Sem perda da generalidade, todas as proposições podem ser expressas na forma clausal. Uma proposição nessa forma tem a seguinte sintaxe geral:

$$B_1 \cup B_2 \cup \dots \cup B_m \subset A_1 \cap A_2 \cap \dots \cap A_n$$

na qual os  $A_s$  e os  $B_s$  são termos. O significado dessa proposição de forma clausal é o seguinte: se todos os  $A_s$  forem verdadeiros, pelo menos um  $B$  será verdadeiro. As principais características destas proposições são as seguintes: quantificadores existenciais não são necessários; os quantificadores universais ficam implícitos no uso de variáveis nas proposições atômicas; e nenhum outro operador além de conjunção e de disjunção é necessário. Além disso, estas duas precisam aparecer na ordem mostrada na forma clausal geral: a disjunção no lado esquerdo e a conjunção no lado direito. Todas as proposições de cálculo de predicados podem ser convertidas algorítmicamente para a forma clausal. Nilsson (1971) apresenta provas de que isso pode ser feito e oferece um algoritmo de conversão simples para tanto.

O lado direito de uma proposição na forma clausal é chamado **antecedente**. O esquerdo, **consequente**; isso porque é a consequência da verdade de seu antecedente. Como exemplos de proposições na forma clausal, considere o seguinte:

$$\text{gosta(bob, truta)} \subset \text{gosta(bob, peixe)} \cap \text{peixe(truta)}$$

$$\text{pai(louis, al)} \cup \text{pai(louis, violet)} \subset \text{pai(al, bob)} \cap \text{mãe(violet, bob)} \cap \text{avô(louis, bob)}$$

A versão em português da primeira dessas proposições declara que, se bob gosta de peixe e truta é um peixe, então bob gosta de truta. A segunda declara que, se al é pai de bob e violet é mãe de bob e louis é avô de bob, então louis é pai de al ou de violet.

### 16.3 Cálculo de Predicados e Demonstração de Teoremas

O cálculo de predicados oferece um método para expressar coleções de proposições. Um uso das coleções de proposições é determinar se quaisquer fatos interessantes ou úteis podem ser inferidos a partir das mesmas. Isso é exatamente análogo ao trabalho dos matemáticos, que lutam para descobrir novos teoremas que possam ser inferidos a partir de axiomas e de teoremas conhecidos.

Os primórdios da ciéncia da computação (os anos 50 e início da década de 60) pre-senciaram muito interesse em automatizar o processo de demonstração de teoremas. Um dos seus avanços mais significativos foi a descoberta do princípio da resolução por Alan Robinson, na *Syracuse University* (Robinson, 1965).

**Resolução** é uma regra de inferência que permite a computação das proposições inferidas a partir de proposições dadas, constituindo, assim, um método com potencial aplicação na demonstração automática de teoremas. A resolução foi idealizada para ser aplicada a proposições na forma clausal. O conceito de resolução é o seguinte: suponhamos que haja duas proposições com as formas

$$P_1 \subset P_2$$

$$Q_1 \subset Q_2$$

O significado delas é que  $P_2$  implica  $P_1$  e  $Q_2$  implica  $Q_1$ . Suponhamos ainda que  $P_1$  seja idêntico a  $Q_2$ , de modo que possamos renomear  $P_1$  e  $Q_2$  como  $T$ . Então, poderíamos rescrever as duas proposições como

$$T \subset P_2$$

$$Q_1 \subset T$$

Agora, uma vez que  $P_2$  implica  $T$  e  $T$  implica  $Q_1$ , é logicamente evidente que  $P_2$  implica  $Q_1$ , o que poderia ser escrito como

$$Q_1 \subset P_2$$

O processo de inferir tal proposição a partir das duas proposições originais é uma resolução.

Como outro exemplo, consideremos as duas proposições:

$$\text{mais velho(joanne, jake)} \subset \text{mãe(joanne, jake)}$$

$$\text{mais sábio(joanne, jake)} \subset \text{mais velho(joanne, jake)}$$

A partir dessas proposições, a seguinte pode ser construída usando-se a resolução:

$$\text{mais sábio(joanne, jake)} \subset \text{mãe(joanne, jake)}$$

A mecânica dessa construção de resolução é simples: os termos do lado esquerdo das duas proposições são unidos por um  $\cup$  para formar o lado esquerdo da nova proposição. Então, a mesma coisa é feita para obter o lado direito dela. Em seguida, o termo que aparece em ambos os seus lados é removido. O processo é exatamente o mesmo quando as proposições têm termos múltiplos em qualquer um ou em ambos os lados. O esquerdo contém inicialmente todos os termos dos lados correspondentes nas duas proposições dadas. O novo lado direito é similarmente construído. Então, o termo que aparece em ambos os lados da nova proposição é removido. Por exemplo, se tivermos

$$\text{pai(bob, jake)} \cup \text{mãe(bob, jake)} \subset \text{pais(bob, jake)}$$

$$\text{avô(bob, fred)} \subset \text{pai(bob, jake)} \cap \text{pai(jake, fred)}$$

a resolução diz que

$$\text{mãe(bob, jake)} \cup \text{avô(bob, fred)} \subset$$

$$\text{pais(bob, jake)} \cap \text{pai(jake, fred)}$$

tem todas, a não ser uma das proposições atômicas originais. A proposição atômica que permitiu a operação  $\text{pai(bob, jake)}$  no lado esquerdo da primeira e no lado direito da segunda é deixada de fora. Em português, diríamos

- se:* bob é um dos pais de jake, isso 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, o que 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, de fato, mais complexa do que os exemplos ilustram. Em particular, a presença de variáveis em proposições exige que a resolução encontre valores para essas variáveis permitindo que o processo de comparação seja bem-sucedido. Este processo de determinação de valores úteis é chamado **unificação**. A atribuição temporária de valores a variáveis para permitir a unificação é chamada **instanciação**.

É comum acontecer do processo de resolução instanciar uma variável para um valor, não conseguir concluir a comparação exigida, e depois ser necessário retroceder e instanciar a variável para um valor diferente. Discutiremos a unificação e a ação de retroceder mais extensamente no contexto do Prolog.

Uma propriedade crucialmente importante da resolução é sua capacidade de detectar qualquer inconsistência em um conjunto dado de proposições. Tal propriedade permite

que a resolução seja usada para demonstrar teoremas, o que pode ser feito da seguinte maneira: podemos imaginar uma prova de teorema em termos de cálculo de predicados como determinado conjunto de proposições pertinentes, com a negação do próprio teorema declarada como uma nova proposição. Este é negado a fim de que a resolução possa ser usada para prová-lo, descobrindo uma inconsistência. Esta é a prova pela contradição. Tipicamente, as proposições originais são chamadas **hipóteses**, e a negação do teorema é chamada **meta**.

Teoricamente, este é um processo válido e útil. O tempo necessário para a resolução, entretanto, pode ser um problema. Não obstante a resolução ser um processo finito quando o conjunto de proposições é finito, o tempo necessário para encontrar uma inconsistência em um grande banco de dados de proposições pode ser enorme.

A demonstração de teoremas é a base da programação lógica. Grande parte do que é computado pode ser expresso na forma de uma lista de fatos dados e de relações como hipóteses, e uma meta a ser inferida a partir das hipóteses, usando-se a resolução.

Quando proposições são usadas para resolução, somente um tipo restrito de forma clausal pode ser ativado, o que simplifica ainda mais o processo de resolução. Os tipos especiais de proposições, as **cláusulas de Horn**, [homenagem a Alfred Horn, que estudou as cláusulas nessa forma (Horn, 1951)], podem estar somente em duas formas: têm uma única proposição atômica no lado esquerdo ou têm o lado esquerdo vazio. O lado esquerdo de uma proposição na forma clausal, às vezes, é chamado cabeça, e as cláusulas de Horn com lados esquerdos vazios são chamadas cláusulas de Horn encabeçadas. Estas são usadas para declarar relações, como, por exemplo

gosta(bob, truta) ⊂ gosta(bob, peixe) ∩ peixe(truta)

As cláusulas de Horn com lados esquerdos vazios freqüentemente são usadas para declarar fatos e são chamadas de cláusulas de Horn sem-cabeça. Por exemplo,

pai(bob, jake)

A maioria das proposições, mas não todas, pode ser declarada como cláusulas de Horn.

## 16.4 Uma Visão Geral da Programação Lógica

As linguagens usadas para programação lógica são chamadas de declarativas porque os programas nelas escritos consistem em declarações em vez de atribuições e em instruções de fluxo de controle. Essas declarações são, de fato, instruções ou proposições, em lógica simbólica.

Uma das características essenciais das linguagens de programação lógicas é sua semântica, chamada de **semântica declarativa**. Seu conceito básico é que existe uma maneira simples de determinar o significado de cada instrução, e não depende de como esta 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 dada em uma linguagem de programação lógica pode ser determinado concisamente a partir da própria instrução. Em uma linguagem imperativa, a semântica de uma instrução de atribuição simples exige o exame de declarações locais, o conhecimento das regras de escopo da linguagem, e possivelmente até mesmo o exame de programas em outros arquivos apenas para determinar os tipos das variáveis na instrução de atribuição.

Então, supondo que a expressão da instrução de atribuição contenha variáveis, a execução do programa antes da instrução de atribuição deve ser rastreada para determinar os valores delas. A ação resultante da instrução, então, depende de seu contexto durante a execução. Comparando com o simples exame de uma única instrução, sem nenhuma necessidade de considerar o contexto textual ou as seqüências de execução, é claro que a semântica declarativa é bem mais simples do que a das linguagens imperativas. Desse modo, a semântica declarativa, muitas vezes, é declarada como uma das vantagens que as linguagens declarativas têm sobre as imperativas (Hogger, 1984, pp. 240-241).

A programação, tanto nas linguagens imperativas como nas funcionais, é principalmente baseada em procedimentos, cujo significado está no fato do programador saber o que deve ser realizado por um programa e instruir o computador a respeito de como a computação deve ser feita. Em outras palavras, o computador é tratado como um simples dispositivo que obedece a ordens. Tudo que é computado deve ter todos os detalhes dessa computação redigidos. Algumas pessoas acreditam que essa é a essência da dificuldade de programar computadores.

A programação em alguns tipos de linguagens não-imperativas, e em especial nas de programação lógicas, não é baseada em procedimentos. Os programas nessas linguagens não declaram exatamente como um resultado deve ser computado, mas, ao contrário, descrevem a sua forma. A diferença é que nós presumimos que o sistema de computador possa, de alguma maneira, determinar como o resultado deve ser obtido. O que é necessário para oferecer tal capacidade para as linguagens de programação lógicas é um meio conciso de abastecer o computador tanto com informações relevantes como com um método de inferência para computar resultados desejáveis. O cálculo de predicados fornece a forma básica de comunicação ao computador, e o método da prova, desenvolvido pela primeira vez por Robinson, fornece a técnica de inferência.

Um exemplo comumente usado para ilustrar a diferença entre sistemas baseados em procedimentos e não-baseados em procedimentos é o processo de reorganizar uma lista de dados em alguma ordem particular, de outra forma conhecida como classificação. Em uma linguagem como o C++, ela é feita explicando-se todos os detalhes de algum algoritmo de classificação em um programa C++ para um computador que possua o compilador C++. O computador, depois de traduzir o programa C++ para código de máquina ou para algum código intermediário interpretativo, segue as instruções e produz a lista classificada.

Em uma linguagem não-baseada em procedimentos, é necessário somente descrever as características da lista classificada: é uma de permutação da lista dada, tal que para cada par de elementos adjacentes uma determinada relação mantém-se entre os dois. Declarando isso formalmente: suponhamos que a lista a ser classificada está em um vetor chamado lista que tem uma faixa de subscrito  $1 \dots n$ . O conceito de classificar os elementos da lista dada, lista\_velha, e colocá-los em um vetor separado, lista\_nova, pode ser expresso da seguinte maneira:

$$\begin{aligned} \text{ordena}(\text{lista\_velha}, \text{lista\_nova}) &\subset \text{permuto}(\text{lista\_velha}, \text{lista\_nova}) \cap \text{ordenada}(\text{lista\_nova}) \\ \text{ordenada}(\text{lista}) &\subset \forall j \text{ tal que } 1 \leq j < n, \text{ lista}(j) \leq \text{lista}(j + 1) \end{aligned}$$

em que *permuto* é um predicado que retorna verdadeiro (*true*) se seu segundo parâmetro for uma permutação de seu primeiro.

A partir dessa descrição, o sistema de linguagem não-baseada em procedimentos poderia produzir a lista classificada. Isso faz com que as linguagens não-baseadas em procedimentos pareçam a mera produção de especificações concisas de exigências de software, o que é uma avaliação justa. Infelizmente, porém, não é assim tão simples. Programas lógicos que usam somente a resolução defrontam-se com sérios problemas de eficiência de máquina. Além disso, a melhor forma de uma linguagem lógica pode não ter sido ainda determi-

nada e bons métodos para criar programas em linguagens de programação lógicas para grandes problemas podem não ter sido ainda desenvolvidos.

## 16.5 As Origens do Prolog

Conforme foi declarado no Capítulo 2, Alain Colmerauer e Phillippe Roussel, da University of Aix-Marseille, com alguma assistência de Robert Kowalski, da University of Edinburgh, desenvolveram o projeto fundamental do Prolog. Colmerauer e Roussel estavam interessados no processamento da linguagem natural, e Kowalski, na demonstração automatizada de teoremas. A colaboração entre as duas universidades perdurou até meados da década de 70. A partir de então, a pesquisa sobre o desenvolvimento e uso da linguagem prosseguiu independentemente nesses dois lugares, resultando, entre outras coisas, em dois dialetos sintaticamente diferentes do Prolog.

O desenvolvimento do Prolog e outros esforços de pesquisa em programação lógica receberam limitada atenção fora de Edimburgo e de Marselha até o anúncio em 1981 de que o governo japonês estava lançando um grande projeto de pesquisa chamado *Fifth Generation Computing Systems* (FGCS) (Fuchi, 1981; Moto-oka, 1981). Um dos seus principais objetivos era desenvolver máquinas inteligentes, e o Prolog foi escolhido como base para o esforço. O anúncio do FGCS suscitou, nos pesquisadores e nos governos dos Estados Unidos e de diversos países europeus, um repentino e forte interesse na inteligência artificial e na programação lógica.

## 16.6 Os Elementos Básicos do Prolog

Agora, há um grande número de diferentes dialetos do Prolog que podem ser agrupados em diversas categorias: os que se desenvolveram a partir do grupo de Marselha, os que vieram do grupo de Edimburgo e uma coleção de dialetos desenvolvidos para microcomputadores, como, por exemplo, a micro-Prolog, descrita por Clark e McCabe (1984). As formas sintáticas desses são um tanto diferentes. Em vez de tentar descrever a sintaxe de diversos dialetos do Prolog ou de algum híbrido, escolhemos um dialeto particular, bastante usado, desenvolvido em Edimburgo. Essa forma da linguagem, às vezes, é chamada sintaxe de Edimburgo. Sua primeira implementação ocorreu em um DEC-System-10 (Warren et al., 1979).

### 16.6.1 Termos

Como acontece em outras linguagens, os programas Prolog consistem em coleções de instruções. Há somente alguns tipos de instruções no Prolog, mas elas podem ser muito complexas. Todas as instruções 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 número inteiro. Átomos são os valores simbólicos do Prolog semelhantes às suas contrapartes do LISP. Em especial, um átomo é uma cadeia de letras, de dígitos

e de grifos que se inicia com uma letra minúscula ou uma cadeia de quaisquer caracteres ASCII imprimíveis delimitados por apóstrofos.

Uma variável é qualquer cadeia de letras, de dígitos e de grifos que se inicia com uma letra maiúscula. As variáveis não são vinculadas a tipos por declarações. A vinculação de um valor e, dessa forma, de um tipo, a uma variável é chamada **instanciação**, que ocorre somente no processo de resolução. Uma variável à qual não foi atribuído um valor é não-instanciada. As instanciações duram somente o tempo que elas precisam para satisfazer uma meta completa, a qual envolve a prova ou a refutação de uma proposição. As variáveis Prolog são somente parentes distantes, tanto em termos de semântica como de uso, das variáveis das linguagens imperativas.

O último tipo de termo é chamado estrutura. As 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 usado para identificar a estrutura. A lista de parâmetros pode ser qualquer lista de átomos, de variáveis ou de outras estruturas. Conforme discutiremos extensamente na próxima seção, as estruturas são o meio de especificar fatos em Prolog. Elas podem ser imaginadas como objetos, em cujo caso permitem que fatos sejam declarados em termos de diversos átomos relacionados. Nesse sentido, as estruturas são relações, porque declaram relações entre termos. Uma estrutura é também um predicado quando seu contexto especificar que ela deve ser uma consulta (pergunta).

### 16.6.2 Instruções Relativas a Fatos

Nossa discussão a respeito das instruções Prolog inicia-se com as usadas para construir as hipóteses ou os bancos de dados de informações presumidas — as instruções a partir das quais a nova informação pode ser inferida.

O Prolog tem duas formas de instrução básicas que correspondem às cláusulas de Horn sem e com cabeça do cálculo de predicados. A forma mais simples de cláusula de Horn sem-cabeça no Prolog é uma estrutura simples, interpretada como uma asserção incondicional ou como um fato. Logicamente, fatos são simplesmente proposições consideradas verdadeiras.

Os exemplos seguintes ilustram os tipos de fatos que se pode ter em um programa Prolog. Note que toda instrução Prolog é encerrada com um ponto final (.) .

```

mulher(shelley).
homem(bill).
mulher(mary).
homem(jake).
pai(bill, jake).
pai(bill, shelley).
mãe(mary, jake).
mãe(mary, shelley).

```

Essas estruturas simples declaram certos fatos a respeito de *jake*, *shelley*, *bill* e de *mary*. Por exemplo, a primeira afirma que *shelley* é uma mulher. As quatro últimas ligam seus dois parâmetros com uma relação nomeada no átomo functor; por exemplo, a quinta proposição poderia ser interpretada como se *bill* fosse o pai de *jake*. Note que estas proposições Prolog, à semelhança daquelas do cálculo de predicados, não têm nenhu-

ma semântica intrínseca. Elas significam qualquer coisa que o programador queira que signifiquem. Por exemplo, a proposição

```
pai(bill, jake).
```

poderia significar que bill e jake têm o mesmo pai ou que jake é o pai de bill. O significado mais comum e direto, porém, é que bill é o pai de jake.

### 16.6.3 Instruções Relativas a Regras

A outra forma básica de instrução Prolog para construir o banco de dados é a cláusula de Horn com cabeça. Essa forma pode estar relacionada a um teorema de matemática conhecido, a partir do qual se pode tirar uma conclusão sobre se o conjunto de determinadas condições é satisfeita. O lado direito é o antecedente, ou a parte se (*if*), e o lado esquerdo é o consequente, ou a parte então (*then*). Se o antecedente de uma instrução Prolog for verdadeiro, então a parte consequente da instrução também deve ser verdadeira. Uma vez que são cláusulas de Horn, o consequente de uma instrução Prolog é um termo único, enquanto que o antecedente pode ser um termo único ou uma conjunção.

As **conjunções** contêm termos múltiplos separados por operações AND lógicas. Em Prolog, a operação AND é implícita. As estruturas que especificam proposições atômicas em uma conjunção são separadas por vírgulas, de modo que se poderia considerar as vírgulas como operadores AND. Como um exemplo de conjunção, considere o seguinte:

```
mulher (shelley), filho (shelley).
```

A forma geral da instrução de cláusula de Horn com cabeça é

```
consequência_1 :- expressão_antecedente
```

Lê-se: a “consequência\_1 poderá ser concluída se a expressão antecedente for verdadeira ou puder tornar-se verdadeira por meio de alguma instanciação de suas variáveis”. Por exemplo,

```
antepassado(mary, shelley) :- mãe(mary, shelley).
```

afirma que, se mary é a mãe de shelley, então mary é um antepassado de shelley. As cláusulas de Horn com cabeça são chamadas **regras** porque declaram regras de implicação entre proposições.

Como acontece com as proposições de forma clausal no cálculo de predicados, as instruções Prolog podem usar variáveis para generalizar seu significado. Lembre-se de que as variáveis na forma clausal fornecem um tipo de quantificador universal implícito. O exemplo seguinte demonstra o uso de variáveis em instruções Prolog:

```
pais(X, Y) :- mãe(X, Y).
pais(X, Y) :- pai(X, Y).
avô(X, Z) :- pais(X, Y), pais(Y, Z).
irmãos(X, Y) :- mãe(M, X), mãe(M, Y),
               pai(F, X), pai(F, Y).
```

Essas instruções fornecem regras de implicação entre algumas variáveis ou objetos universais. Nesse caso, estes são X, Y, Z, M e F. A primeira regra afirma que, se houver instâncias de X e Y tal que mãe(X, Y) seja verdadeiro, então para essas mesmas instâncias de X e Y, pais(X, Y) será verdadeiro.

Hidden page

Uma vez que o processo de provar uma submeta é feito por um processo de comparação de proposições, às vezes ele é chamado de *casamento*. Em alguns casos, provar uma submeta é chamado **satisfazê-la**.

Considere a seguinte consulta:

```
homem(bob).
```

Essa instrução de meta é o tipo mais simples. É relativamente fácil para a resolução determinar se ela é verdadeira ou falsa. Seu padrão é comparado com os fatos e com as regras do banco de dados. Se o banco de dados incluir o fato

```
homem(bob).
```

a prova será banal. Entretanto, se o banco de dados contiver o seguinte fato e regra de inferência,

```
pai(bob).  
homem(X) :- pai(X).
```

Seria necessário que o Prolog encontrasse essas duas instruções e usasse-as para inferir a verdade da meta. Isso necessitaria de unificação para instanciar X temporariamente para bob.

Considere agora a meta

```
homem(X).
```

Nesse caso, o Prolog deve comparar a meta com as proposições do banco de dados. A primeira proposição que ele encontra sob a forma da meta, tendo qualquer objeto como seu parâmetro, fará com que X seja instanciado para o valor desse objeto. X será, então, exibido como o resultado. Se não houver nenhuma proposição com a forma da meta, o sistema indicará, usando no, que esta não pode ser satisfeita.

Há duas abordagens opostas para tentar casar determinada meta com um fato no banco de dados. O sistema pode iniciar-se com os fatos e com as regras do banco de dados e tentar encontrar a sequência de coincidências que levam à meta. Esta abordagem chama-se **resolução baixo-cima** ou **encadeamento progressivo**. A alternativa é iniciar com a meta e tentar encontrar uma sequência de proposições coincidentes que levam a algum conjunto de fatos originais no banco de dados. Esta abordagem chama-se **resolução cima-baixo** ou **encadeamento retrógrado**. Em geral, este último funciona bem quando há um conjunto razoavelmente pequeno de respostas candidatas. A abordagem do encadeamento progressivo é melhor quando o número de respostas possivelmente corretas é grande; nessa situação, o encadeamento retrógrado exigiria um número muito grande de coincidências para chegar a uma resposta. As implementações Prolog usam o encadeamento retrógrado para resolução, presumivelmente porque seus projetistas acreditavam que ele era adequado para uma classe maior de problemas do que o progressivo.

Considere novamente a consulta:

```
homem(bob).
```

Suponhamos que o banco de dados contenha

```
pai(bob).  
homem(X) :- pai(X).
```

O encadeamento progressivo procuraria e encontraria a primeira proposição. A meta seria, então, inferida comparando-se a primeira proposição com o lado direito da segunda regra

(`pai(X)`) pela instanciação de `X` para `bob` e, depois, comparando-se o lado esquerdo da segunda proposição para a meta. O encadeamento retrógrado compararia primeiro a meta com o lado esquerdo da segunda proposição (`homem(X)`) pela instanciação de `X` para `bob`. Como seu último passo, ele compararia o lado direito da segunda proposição (agora `pai(bob)`) com a primeira.

A questão de projeto seguinte vem à tona quando a meta tiver mais de uma estrutura, como em nosso exemplo acima. A questão, então, é se a busca da solução é feita primeiramente pela profundidade (*depth-first*) ou pela largura (*breadth-first*). Uma busca feita **primeiramente pela profundidade** localiza uma seqüência completa de proposições — uma prova — referente à primeira submeta antes de trabalhar nas outras. Uma busca **primeiramente pela largura** trabalha em todas as submetas de determinada meta de maneira paralela. Os projetistas do Prolog optaram pela abordagem da profundidade porque ela pode ser feita com menos recursos do computador. A abordagem da largura é uma busca paralela que pode necessitar uma grande quantidade de memória.

O último recurso do mecanismo de resolução do Prolog a ser discutido é o *backtracking*. Quando uma meta com múltiplas submetas está sendo processada e o sistema não consegue mostrar a veracidade de uma das submetas, ele abandona a submeta que não conseguia provar. O sistema, então, reconsidera a submeta anterior, se houver uma, e tenta encontrar uma solução alternativa para ela. Este retroceder na meta para a reconsideração de uma submeta provada anteriormente é chamado **backtracking**. Uma nova solução é encontrada ao iniciar a procura onde a busca anterior dessa submeta interrompeu-se. Múltiplas soluções para uma submeta resultam de diferentes instanciações de suas variáveis. O *backtracking* pode exigir muito tempo e espaço, porque ele talvez tenha de encontrar todas as provas possíveis para cada submeta. Tais provas de submetas podem não estar organizadas para minimizar o tempo necessário para localizar aquela que resultará na prova final completa, o que exacerbará o problema.

Para solidificar sua compreensão do *backtracking*, considere o seguinte exemplo. Suponha que haja um conjunto de fatos e de regras em um banco de dados e que tenha sido apresentado ao Prolog a seguinte meta composta:

```
homem(X), pais(X, shelley).
```

Essa meta pergunta se há uma instanciação de `X` tal que `X` seja um `homem` e também um dos `pais` de `shelley`. O Prolog localiza o primeiro fato no banco de dados que tem `homem` como seu functor. Depois, ele instancia `X` para o parâmetro do fato encontrado, digamos, `mike`. Então, ele tenta provar que `pais(mike, shelley)` é verdadeiro. Se falhar, ele retrocederá (fará *backtracking*) à primeira submeta, `homem(X)` e tentará satisfazê-la novamente com alguma instanciação alternativa de `X`. O processo de resolução talvez tenha de localizar todo `homem` no banco de dados antes de encontrar aquele que satisfaça `pais de shelley`. Definitivamente, ele deve localizar todos os `homens` para provar que a meta não pode ser satisfeita. Note que nosso exemplo de meta poderia ser processado mais eficientemente se a ordem das duas submetas fosse invertida. Então, somente depois que a resolução tivesse encontrado um dos `pais de shelley` é que ela tentaria comparar essa pessoa com a submeta `homem`. Isso será mais eficiente se `shelley` tiver menos pais do que o número de `homens` no banco de dados, o que parece ser uma suposição justa. A Seção 16.7.1 discutirá um método de limitar o *backtracking* feito por um sistema Prolog.

As buscas em bancos de dados no Prolog sempre se desenvolvem na direção do primeiro ao último.

As duas subseções seguintes descrevem exemplos Prolog que ilustram adicionalmente o processo de resolução.

### 16.6.6 Aritmética Simples

O Prolog suporta variáveis de números inteiros e seus respectivos cálculos. Originalmente, os operadores aritméticos eram functores, de modo que a soma de 7 com a variável *x* era formada desta maneira:

```
+ (7, X)
```

O Prolog, agora, permite uma sintaxe mais abreviada para cálculos aritméticos com o operador *is*. Este toma uma expressão aritmética como seu operando direito e uma variável como seu operando esquerdo. Todas as variáveis da expressão já devem estar instanciadas, mas a variável do lado esquerdo não pode ser previamente instanciada. Por exemplo, em

```
A is B / 17 + C.
```

Se *B* e *C* forem instanciados, mas *A* não, essa cláusula fará com que *A* seja instanciado com o valor da expressão. Quando isso acontecer, a cláusula será satisfeita. Se *B* ou *C* não estiverem instanciados ou se *A* estiver instanciado, a cláusula não será satisfeita e nenhuma instânciação de *A* irá se desenvolver. A semântica de uma proposição *is* é consideravelmente diferente de uma instrução de atribuição em uma linguagem imperativa. Essa diferença pode levar a um cenário interessante. Uma vez que o operador *is* faz a cláusula na qual ele aparece se assemelhar a uma instrução de atribuição, um programador Prolog iniciante pode se sentir tentado a escrever uma instrução como

```
Soma is Soma + numero.
```

que jamais será útil, ou até mesmo válido, em Prolog. Se *Soma* não estiver instanciada, a referência a ela no lado direito estará indefinida, e a cláusula falhará. Se *Soma* já estiver instanciada, acontecerá o mesmo, porque o operando esquerdo não pode ter uma instânciação atual quando *is* é avaliada. Em qualquer um dos casos, a instanciação de *Soma* para o novo valor não acontecerá (se o valor de *Soma* + *numero* for exigido, ele poderá ser vinculado a algum novo nome).

O Prolog não tem instruções de atribuição no mesmo sentido que as linguagens imperativas. Elas simplesmente não são necessárias na maioria das programações para as quais o Prolog foi projetado. A utilidade das instruções de atribuição nas linguagens imperativas depende da capacidade do programador para controlar o fluxo de controle de execução do código onde a instrução de atribuição está embutida. Uma vez que esse tipo de controle nem sempre é possível no Prolog, essas instruções são bem menos úteis.

Como um exemplo simples do uso de computações numéricas em Prolog, considere o seguinte problema: suponhamos conhecer as velocidades médias de diversos automóveis em uma pista de corrida particular e a quantidade de tempo que eles permanecem na pista. Essa informação básica pode ser codificada como fatos, e a relação entre velocidade, tempo e distância pode ser escrita como uma regra, como no seguinte:

```
velocidade(ford, 100).
velocidade(chevy, 105).
velocidade(dodge, 95).
velocidade(volvo, 80).
tempo(ford, 20).
tempo(chevy, 21).
tempo(dodge, 24).
tempo(volvo, 24).
```

Hidden page

Hidden page

Hidden page

Se `Elemento_1` tiver sido instanciado com `picles` e `Lista_2` tiver sido instanciada com `[amendoim, ameixa seca, pipoca]`, a notação acima criará, para esta referência, a lista `[picles, amendoim, ameixa seca, pipoca]`.

Conforme declaramos acima, a notação de lista que inclui o símbolo `|` é universal: ela pode especificar uma construção de lista ou um desmantelamento de lista. Note ainda que os seguintes exemplos são equivalentes:

```
[damasco, pêssego, pera | []]
[damasco, pêssego | [pera]]
[damaco | [pêssego, pera]]
```

Ao lidar com listas, certas operações básicas freqüentemente são necessárias, como aquelas encontradas no LISP. Como um exemplo dessas operações no Prolog, examinemos uma definição de `append`, relacionada a uma dessas funções no LISP. Nesse exemplo, as diferenças e as semelhanças entre as linguagens funcionais e as declarativas podem ser vistas. Não precisamos especificar como o Prolog deve construir uma nova lista a partir de outras dadas; ao contrário, precisamos somente especificar as características da nova lista em termos das listas dadas.

Na aparência, a definição de `append` no Prolog é muito semelhante à versão LISP, e uma espécie de recursão na resolução é usada de uma maneira semelhante para produzir a nova lista. No caso do Prolog, a recursão é causada e controlada pelo processo de resolução.

Os dois primeiros parâmetros para a operação `append` no código seguinte são as duas listas a serem anexadas, e o terceiro parâmetro é a lista resultante:

```
append([], Lista, Lista).
append([Cabeça | Lista_1], Lista_2, [Cabeça | Lista_3]) :-  
    append(Lista_1, Lista_2, Lista_3).
```

A primeira proposição especifica que, quando a lista vazia é anexada a qualquer outra lista, esta última é o resultado. Essa instrução corresponde à etapa de finalização da recursão da função `append` LISP. Note que a proposição de finalização é colocada antes da de recursão. Isso é feito porque sabemos que o Prolog comparará as duas proposições em uma ordem, iniciando com a primeira (devido ao uso que ele faz da ordem `depth-first` — primeiramente pela profundidade).

A segunda proposição especifica diversas características da nova lista. Ela corresponde à etapa de recursão na função LISP. O predicado do lado esquerdo declara que o primeiro elemento da nova lista é o mesmo que o primeiro elemento da primeira lista dada, porque ambos têm o nome `Cabeça`. Quando `Cabeça` for instanciado para um valor, todas as suas ocorrências na meta são, com efeito, simultaneamente instanciadas para esse valor. O lado direito da segunda instrução especifica que a cauda da primeira (`Lista_1`) tem a segunda (`Lista_2`) anexada a ela para formar a cauda (`Lista_3`) da lista resultante.

Uma maneira de ler a segunda instrução de `append` é a seguinte: anexando a lista `[Cabeça | Lista_1]` a qualquer `Lista_2` produzirá `[Cabeça | Lista_3]`, mas somente se a `Lista_3` for formada anexando-se `Lista_1` a `Lista_2`. No LISP, isso seria

```
(CONS (CAR PRIMEIRO) (APPEND (CDR PRIMEIRO) SEGUNDO))
```

Tanto nas versões Prolog como no LISP, a lista resultante não é construída até que a recursão produza a condição de finalização; neste caso, a primeira lista deve ficar vazia. Então, a resultante é construída usando-se a própria função `append`; os elementos tomados da primeira lista são adicionados, em ordem inversa à segunda. A inversão é feita pelo desdobramento da recursão.

Para ilustrarmos como o processo `append` progride, consideremos o seguinte exemplo rastreado:

```
trace.
append([bob, jo], [jake, darcie], Familia).

(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])
Familia = [bob, jo, jake, darcie]
yes
```

As duas primeiras chamadas representam submetas e têm `Lista_1` não-vazia, de modo que criam as chamadas recursivas a partir do lado direito da segunda instrução. O lado esquerdo desta especifica efetivamente os argumentos para as chamadas recursivas ou metas, desmantelando, assim, a primeira lista, um elemento por etapa. Quando a primeira lista fica vazia em uma chamada (ou submeta), a instância atual do lado direito da segunda instrução obtém sucesso ao comparar a primeira instrução. O efeito disso é retornar como terceiro parâmetro o valor da lista vazia anexada à segunda lista de parâmetros original. Em saídas sucessivas, que representam coincidências bem-sucedidas, os elementos que foram removidos da primeira lista são anexados à lista resultante, `Familia`. Quando a saída (`exit`) da primeira lista é realizada, o processo é concluído e a resultante é exibida.

As proposições `append` também podem ser usadas para criar outras operações de lista, como, por exemplo, a seguinte, cujo efeito convidamos o leitor a determinar. Note que `list_op_2` pretende ser usado ao oferecer uma lista como seu primeiro parâmetro e uma variável como seu 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([Cabeça | Cauda], Lista) :- list_op_2(Cauda, Resultado),
                                         append(Resultado, [Cabeça],
                                               Lista).
```

Como o leitor talvez tenha sido capaz de determinar, `list_op_2` faz com que o sistema Prolog instancie seu segundo parâmetro com uma lista que tem os elementos da lista do primeiro parâmetro, mas em ordem inversa. Por exemplo, `((maçã, laranja, uvas), Q)` instancia `Q` com a lista `[uvas, laranja, maçã]`.

Novamente, não obstante as linguagens Prolog e LISP serem fundamentalmente diferentes, operações similares podem usar estas abordagens. No caso da operação inversa, tanto a `list_op_2` do Prolog como a função `reverse` do LISP incluem a condição de finalização de recursão, juntamente com o processo básico de anexar a inversão do CDR ou da cauda da lista, ao CAR, ou cabeça da lista, para criar a resultante.

O seguinte é um rastreamento deste processo, agora chamado `reverso`.

```
trace.
reverso([a, b, c], Q).
```

```

(1) 1 Call: reverso([a, b, c], _6)?
(2) 2 Call: reverso([b, c], _65636)?
(3) 3 Call: reverso([c], _65646)?
(4) 4 Call: reverso([], _65656)?
(4) 4 Exit: reverso([], [])
(5) 4 Call: append([], [c], _65646)?
(5) 4 Exit: append([], [c], [c])
(3) 3 Exit: reverso([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: reverso([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: reverso([a, b, c], [c, b, a])

```

`Q = [c, b, a]`

Suponhamos a necessidade de sermos capazes de definir se determinado símbolo está em uma lista dada. Uma descrição Prolog disso é

```

membro(Elemento, [Elemento | _]).
membro(Elemento, [_ | Lista]) :- membro(Elemento, Lista).

```

O sublinhado indica uma variável “anônima”, significando que não nos preocupamos com qual instanciação poderia ser obtida da unificação. A primeira instrução acima obtém sucesso se `Elemento` for a cabeça da lista, ou inicialmente, ou depois de diversas recursões pela segunda instrução. Esta obtém sucesso se `Elemento` estiver na cauda da lista. Considere os seguintes exemplos de rastreamento:

```

trace.
membro(a, [b, c, d]).
(1) 1 Call: membro(a, [b, c, d])?
(2) 2 Call: membro(a, [c, d])?
(3) 3 Call: membro(a, [d])?
(4) 4 Call: membro(a, [])?
(4) 4 Fail: membro(a, [])
(3) 3 Fail: membro(a, [d])
(2) 2 Fail: membro(a, [c, d])
(1) 1 Fail: membro(a, [b, c, d])
no

membro(a, [b, a, c]).
(1) 1 Call: membro(a, [b, a, c])?
(2) 2 Call: membro(a, [a, c])?
(2) 2 Exit: membro(a, [a, c])
(1) 1 Exit: membro(a, [b, a, c])
yes

```

## 16.7 Deficiências do Prolog

Surgem diversos problemas ao usar o Prolog como uma linguagem de programação lógica. Não obstante ser uma ferramenta útil, ele não é puro, nem perfeito.

### 16.7.1 Controle da Ordem de Resolução

O Prolog, por razões de eficiência, permite que o usuário controle a ordem de casamento de padrões durante a resolução. Em um ambiente de programação lógica puro, a ordem de coincidências tentadas, que se desenvolvem durante a resolução, não é determinística, e todas as coincidências poderiam ser tentadas concorrentemente. Porém, uma vez que o Prolog sempre compara na mesma ordem, partindo do início do banco de dados e na extremidade esquerda de uma meta dada, o usuário pode afetar profundamente a eficiência ao organizar as declarações do banco de dados para otimizar uma aplicação particular. Por exemplo, se o usuário tiver conhecimento de que certas regras têm muito mais probabilidade de ser bem-sucedidas do que outras durante uma “execução” particular, o programa pode ficar mais eficiente ao colocá-las primeiro no banco de dados.

A execução lenta do programa não é o único resultado negativo da ordem definida pelo usuário em programas Prolog. É muito fácil escrever declarações sob formas que provocam laços infinitos e, assim, a falha total do programa. Por exemplo, considere a seguinte forma de declaração recursiva:

```
f(X, Y) :- f(Z, Y), g(X, Z).
```

Devido à ordem de avaliação primeiramente pela profundidade, da esquerda para a direita do Prolog, independentemente do propósito da declaração, isso causará um laço infinito. Como um exemplo deste tipo de declaração, considere

```
antepassados(X, X).  
antepassados(X, Y) :- antepassados(Z, Y), pais(X, Z).
```

Ao tentar satisfazer a primeira submeta do lado direito da segunda proposição, o Prolog instancia *Z* para tornar *antepassados* verdadeiro. Depois, ele tentará satisfazer esta nova submeta, retornando imediatamente para a definição de *antepassados* e repetindo o mesmo processo, levando a uma recursão infinita.

Esse problema particular é idêntico ao que um analisador sintático recursivo descendente tem com a recursão à esquerda em uma regra gramatical, conforme discutimos no Capítulo 3. Para solucioná-lo, simplesmente inverte-se a ordem dos termos no lado direito da proposição acima. O problema com isso é que uma simples mudança de disposição de termos não deve ser crucial para a exatidão do programa. Afinal de contas, a ausência da necessidade do programador preocupar-se com a ordem de controle é supostamente uma das vantagens da programação lógica.

Além de permitir que o usuário controle a ordem do bancos de dados e de submetas, o Prolog, em outra concessão à eficiência, admite algum controle explícito do *backtracking*. Isso é feito com o operador *cut*, especificado por um ponto de exclamação (!). O operador *cut* é, de fato, uma meta, não um operador. Como uma meta, ele sempre obtém sucesso imediatamente, mas não pode ser novamente satisfeita pelo *backtracking*. Assim, um efeito colateral do *cut* é que as submetas à sua esquerda em uma meta composta também não podem ser novamente satisfeitas pelo *backtracking*. Por exemplo, na meta

a, b, !, c, d.

Se tanto a como b obtiverem sucesso mas c falhar, a meta inteira falhará. Ela seria usada se fosse conhecido que, quando c falha, é uma perda de tempo satisfazer novamente b ou a.

O propósito do *cut*, então, é permitir que o usuário crie programas mais eficientes dizendo ao sistema quando ele não deve satisfazer novamente as submetas que se presume poderiam não resultar em uma prova completa.

Como um exemplo do uso do operador *cut*, considere as regras de membro da Seção 16.6.7, as quais serão repetidas abaixo:

```
membro(Elemento, [Elemento | _]).  
membro(Elemento, [_ | Lista]) :- membro(Elemento, Lista).
```

Se o argumento de lista de *membro* representar um conjunto, então ele poderá ser satisfeito somente uma vez (os conjuntos não contêm nenhum elemento duplicado). Portanto, se *membro* for usado como uma submeta em uma declaração de meta múltipla, poderá haver um problema. O problema é que se *membro* tiver sucesso mas a submeta seguinte falhar, o backtracking tentará satisfazer *membro* novamente prosseguindo uma comparação anterior. Entretanto, uma vez que o argumento de *lista* de *membro* tem somente uma cópia do elemento para iniciar, ele possivelmente não poderá ser bem-sucedido novamente, o que fará, por fim, que a meta falhe, apesar de quaisquer tentativas adicionais de satisfazer o mesmo novamente.

A solução para essa ineficiência é adicionar um lado direito à primeira declaração da definição de *membro*, com o operador *cut* como o único elemento, como em

```
membro(Elemento, [Elemento | _]) :- !.
```

O backtracking não tentará satisfazer *membro* novamente, mas, em vez disso, fará com que a submeta inteira falhe.

*Cut* é especialmente útil em uma estratégia de programação em Prolog chamada **gerar e testar**. Nesses programas, a meta consiste em submetas que geram soluções potenciais, então checadas por submetas posteriores de "teste". As soluções rejeitadas exigem backtracking para submetas "geradoras", que geram novas soluções potenciais. Como um exemplo de "gerar e testar", considere o seguinte, que aparece em Clocksin e Mellish (1984):

```
dividir(N1, N2, Resultado) :- é_inteiro(Resultado),  
                           Produto1 is Resultado * N2,  
                           Produto2 is (Resultado + 1) * N2,  
                           Produto1 <= N1, Produto2 > N1, !.
```

Esse programa realiza divisão de números inteiros, usando adição e multiplicação. Uma vez que a maioria dos sistemas Prolog oferece a divisão como um operador, ele não é de fato útil, a não ser para ilustrar um programa de gerar e testar simples.

O predicado *é\_inteiro* é bem-sucedido, contanto que seu parâmetro possa ser instanciado para algum número inteiro não-negativo. Se seu argumento não for instanciado, *é\_inteiro* o instanciará para o valor 0. Se ocorrer o contrário para um número inteiro, *é\_inteiro* instanciará para o maior valor inteiro seguinte.

Assim, em *dividir*, *é\_inteiro* é a submeta geradora. Ela gera elementos da sequência 0, 1, 2, ..., um a cada vez em que é satisfeita. Todas as outras são as submetas de teste — elas fazem a verificação para determinar se o valor produzido pelo *é\_inteiro* é, de fato, o quociente dos dois primeiros parâmetros, *N1* e *N2*. O propósito do *cut* como última submeta é simples: ele impede que *dividir* tente encontrar uma solução alternativa assim que tiver encontrado a solução. Não obstante *é\_inteiro* possa gerar um número

Hidden page

Desse modo, o Prolog “pensa” que `jake` é um irmão de si mesmo. Isso acontece porque o sistema instancia primeiro `M` com `bill` e `X` com `jake` para tornar a primeira submeta, `pais(M, Y)`, verdadeira. Depois, ele parte do início do banco de dados novamente para comparar a segunda submeta, `pais(M, Y)` e chega nas instâncias de `M` com `bill` e `Y` com `jake`. Uma vez que as duas submetas são satisfeitas independentemente, com ambas as comparações iniciando-se no banco de dados, surge a resposta acima. Para evitar isso, `X` deve ser especificado para ser um irmão de `Y` somente se eles tiverem os mesmos pais e não forem iguais. Infelizmente, declarar que eles não são iguais não é direto no Prolog, conforme discutiremos. O método mais exato exigiria adicionar um fato para cada par de átomos, declarando que eles não eram os mesmos. Isso pode, é claro, fazer com que o banco de dados fique muito grande, porque, muitas vezes, há bem mais informações negativas do que informações positivas. Por exemplo, a maioria das pessoas tem 364 maiores datas de nascimento do que de aniversário.

Uma solução alternativa simples é declarar na meta que `X` não deve ser o mesmo que `Y`, como em

```
irmãos(X, Y) :- pais(M, X), pais(M, Y), not(X = Y)
```

Em outras situações, a solução não é tão simples.

O operador Prolog `not` é satisfeito neste caso se a resolução não puder satisfazer a submeta `X = Y`. Portanto, se o `not` obtiver sucesso, não significará necessariamente que `X` é o mesmo que `Y`. Assim, o operador Prolog `not` não é equivalente ao operador lógico NOT, no qual este significa que seu operando provavelmente é verdadeiro. Esta não equivalência pode levar a um problema caso tenhamos uma meta da forma

```
not(not(alguma_meta)).
```

a qual deveria ser equivalente a

```
alguma_meta.
```

se o operador Prolog `not` fosse um operador lógico NOT verdadeiro. Em alguns casos, entretanto, eles não são a mesma coisa. Por exemplo, considere novamente as regras `membro`:

```
membro(Elemento, [Elemento | _]) :- !.  
membro(Elemento, [_ | Lista]) :- membro(Elemento, Lista).
```

Para descobrir um dos elementos de uma lista dada, poderíamos usar a meta

```
membro(X, [mary, fred, barb]).
```

a qual faria com que `X` fosse instanciado com `mary`, que seria, então, impressa. Entretanto, se usássemos

```
not(not(membro(X, [mary, fred, barb]))).
```

a seguinte seqüência de eventos iria desenvolver-se: primeiro, a meta interna seria bem-sucedida, instanciando `X` para `mary`. Depois, o Prolog simplesmente tentaria satisfazer a meta seguinte:

```
not(membro(X, [mary, fred, barb])).
```

Isso falharia, porque `membro` obteve sucesso. Quando essa meta falhasse, `X` seria “desinstaciado”, porque o Prolog sempre “desinstancia” todas as variáveis em todas as metas que falham. Em seguida, o Prolog tentaria satisfazer a meta `not` externa, a qual seria bem-sucedida, porque seu argumento falhou. Finalmente, o resultado, `X`, seria impresso. Mas `X`

não estaria atualmente instanciado, de modo que o sistema indicaria isso. Geralmente, variáveis desinstaciadas também são impressas na forma de uma cadeia de dígitos precedidos por um sublinhado. Assim, o fato de o `not` do Prolog não ser equivalente a um NOT lógico pode ser, no mínimo, enganoso.

A razão fundamental pela qual o NOT lógico não pode ser uma parte integrante do Prolog é a forma da cláusula de Horn:

$$A \leftarrow B_1 \cap B_2 \cap \dots \cap B_n$$

Se todas as proposições  $B_i$  forem verdadeiras, pode-se concluir que  $A$  também é. No entanto, independentemente da veracidade ou da falsidade de qualquer um ou de todos os  $B_i$ s, não se pode concluir que  $A$  é falso. A partir da lógica positiva, pode-se concluir somente ela mesma. Dessa forma, o uso da forma da cláusula de Horn evita quaisquer conclusões negativas.

#### 16.7.4 Limitações Intrínsecas

Uma meta fundamental da programação lógica, conforme afirmamos na Seção 16.4, é oferecer programação não-baseada em procedimentos, ou seja, um sistema pelo qual os programadores especificam o que se espera que um programa faça, mas não precisam detalhar como isso será realizado. O exemplo lá apresentado para classificação é reescrito aqui:

```
ordenada(lista_velha, lista_nova) :- permuta(lista_velha, lista_nova), ordenada(lista_nova).
ordenada(lista) :- forall(j, 1 <= j < n, lista(j) <= lista(j + 1))
```

Isso pode ser facilmente escrito em Prolog. Por exemplo, a submeta ordenada pode ser expressa como

```
ordenada([]).
ordenada([x]).
ordenada([x, y | lista]) :- x <= y, ordenada([y | lista]).
```

O problema com o processo de classificação acima é que ele não faz nenhuma idéia de como classificar, a não ser simplesmente enumerar todas as permutações da lista dada até que venha a criar uma que corresponde à classificada — um processo muito lento, de fato.

Até agora, ninguém descobriu um processo pelo qual a descrição de uma lista classificada pode ser transformada em algum algoritmo eficiente de classificação. A resolução é capaz de muitas coisas interessantes, mas certamente não esta. Portanto, um programa Prolog que classifica uma lista deve especificar os detalhes de como isso pode ser feito, como acontece em uma linguagem imperativa ou funcional.

Todos esses problemas significam que a programação lógica deve ser abandonada? Não, em absoluto, não! Ela é capaz de lidar com muitas aplicações úteis. Além disso, baseia-se em um conceito intrigante, e, portanto, é interessante em si mesma e por si mesma. Por fim, há a possibilidade de que alguma nova técnica de inferência seja desenvolvida para um sistema de linguagem de programação lógica que exija somente o quê, não o como, em sua especificação de programas.

Hidden page

Um dos problemas fundamentais para o projetista de um sistema especialista é lidar com as inconsistências e com a falta de completude inevitáveis do banco de dados. A programação lógica parece adequar-se bem a esses problemas. Por exemplo, as regras de inferência padrão podem ajudar a lidar com o problema da falta de completude.

O Prolog pode e tem sido usado para construir sistemas especialistas. Ele preenche as necessidades básicas destes, usando a resolução como a base para processamento de consultas, usando a capacidade de adicionar fatos e regras para proporcionar a capacidade de aprendizagem além de sua facilidade de rastreamento (*trace*) para informar ao usuário sobre o “raciocínio” que há por trás de determinado resultado. Falta ao Prolog a capacidade automática do sistema consultar o usuário para obter informações adicionais quando é necessário.

Um dos usos mais conhecidos da programação lógica em sistemas especialistas é o sistema de construção destes, conhecido como APES, descrito por Sergot (1983) e Hammond (1983). O sistema APES inclui uma facilidade muito flexível para coletar informações do usuário durante a construção de um sistema especialista. Ele também inclui um segundo interpretador para produzir explicações para suas respostas às consultas.

O APES tem sido usado de maneira bem-sucedida para produzir diversos sistemas especialistas, inclusive um para as normas de um programa de benefícios sociais do governo e um para a *British Nationality Act*, a fonte definitiva das leis de cidadania britânica.

### **16.8.3 Processamento de Linguagem Natural**

Certos tipos de processamento de linguagem natural podem ser feitos com programação lógica. Em especial, as suas interfaces para sistemas de computador, como bancos de dados inteligentes e outros sistemas baseados no conhecimento, podem ser convenientemente feitos com a programação lógica. Para descrever a sintaxe de linguagem, formas de programação lógica têm sido consideradas equivalentes às gramáticas livres de contexto. Os procedimentos de prova em sistema de programação lógica têm sido considerados equivalentes a certas estratégias de análise sintática. De fato, a resolução de encadeamento retrógrado pode ser usada diretamente para analisar sentenças cujas estruturas são descritas por gramáticas livres de contexto. Também foi descoberto que alguns tipos de semântica de linguagens naturais podem tornar-se claros ao se modelar as linguagens com programação lógica. Em especial, a pesquisa em redes semânticas baseadas na lógica mostrou que conjuntos de sentenças em linguagens naturais podem ser expressos na forma clausal (Deliyanni e Kowalski, 1979). Kowalski (1979) também discute as redes semânticas baseadas na lógica.

### **16.8.4 Educação**

Na área da educação, há várias experiências em ensinar crianças de até sete anos de idade a usar a linguagem de programação lógica micro-Prolog (Ennals, 1980). Os pesquisadores reivindicam um grande número de vantagens no ensino do Prolog a jovens. Primeiro, é possível fazer a introdução à informática usando tal abordagem. Isso tem o efeito colateral de ensinar lógica, o que pode resultar em pensamento e expressão mais claros. Pode ajudar os estudantes a aprender uma variedade de matérias, como, por exemplo, resolver equações matemáticas, lidar com a gramática de linguagens naturais e entender as regras e a ordem do mundo físico.

As experiências no ensino da programação lógica para crianças muito pequenas produziram o interessante resultado de que é mais fácil ensiná-la a um iniciante do que a um programador com uma quantidade significativa de experiência em uma linguagem imperativa.

## 16.9 Conclusões

Muitos acreditam que o Prolog é ainda, pelo menos neste ponto, uma grande experiência. Ele tem um grande número de proponentes, entretanto, como muitas outras linguagens tiveram. Alguns acreditam que ele pode ser pelo menos uma parte da solução para a crise de software, na qual as linguagens imperativas atualmente em uso não conseguem lidar com os problemas que precisam ser resolvidos por computadores (Cuadrado e Cuadrado, 1985).

Algumas das razões pelas quais os adeptos acreditam que o Prolog é melhor do que as linguagens imperativas são as seguintes, conforme foi declarado originalmente por Jacques Cohen (1985), um dos promotores do Prolog:

- Uma vez que o Prolog baseia-se na lógica, seus programas têm mais probabilidade de serem organizados e escritos mais logicamente, o que levaria a menos erros e a menos manutenção.
- O processamento Prolog é naturalmente paralelo, tornando os seus interpretadores especialmente capazes de tirar proveito de máquinas com múltiplos processadores.
- Devido à concisão dos programas Prolog, o tempo de desenvolvimento é diminuído, tornando-o uma boa ferramenta para prototipação.

Evidentemente, há pessoas que não concordam. Muitos cientistas da computação são céticos em relação à utilidade do Prolog fora de algumas pequenas áreas da inteligência artificial. Alguns acreditam que ele substituirá o LISP como a principal linguagem da inteligência artificial, ainda que isso certamente não esteja claro nesse momento. Warren et al. (1977) fez uma comparação entre as duas linguagens.

## RESUMO

A lógica simbólica constitui a base para a programação lógica e para as linguagens de programação lógica. A abordagem de programação lógica é usar, como banco de dados, uma coleção de fatos e de regras que declaram relações entre fatos, e usar um processo de inferência automática para verificar a validade de novas proposições, supondo que os fatos e as regras do banco de dados sejam verdadeiros. Essa abordagem é desenvolvida para demonstração automática de teoremas.

O Prolog é a linguagem de programação lógica mais popular. As origens da programação lógica situam-se no desenvolvimento que Robinson fez da regra de resolução para inferência lógica. O Prolog foi desenvolvido principalmente por Colmerauer e Roussel, em Marselha, com alguma ajuda de Kowalski, em Edimburgo.

Os programas lógicos não devem ser baseados em procedimentos, o que significa que as características da solução são especificadas, mas o processo completo de obtê-la não o é.

As declarações Prolog são fatos, regras ou metas. A maioria é composta de estruturas, de proposições atômicas e de operadores lógicos, embora expressões aritméticas também sejam permitidas.

A resolução é a atividade principal de um interpretador Prolog. Este processo, que usa o backtracking extensivamente, envolve principalmente casamento de padrões entre proposições. Quando variáveis estão envolvidas, elas podem ser instanciadas para valores para proporcionar casamentos. Tal processo de instanciação é chamado unificação.

Há uma série de problemas com a situação atual da programação lógica. Por razões de eficiência, e até mesmo para evitar laços infinitos, às vezes os programadores precisam declarar informações de fluxo de controle em seus programas. Além disso, há os problemas de pressuposição de mundo fechado e de negação.

A programação lógica tem sido usada em um grande número de áreas diferentes, principalmente em sistemas de bancos de dados relacionais, em sistemas especialistas e no processamento de linguagem natural.

## NOTAS BIBLIOGRÁFICAS

A linguagem Prolog é descrita em diversos livros. A forma de Edimburgo da linguagem é abordada em Clocksin e Mellish (1997). A implementação para microcomputadores é descrita em Clark e McCabe (1984).

Hogger (1984) é um excelente livro sobre a área geral da programação lógica. Ele é a fonte do material da seção deste capítulo sobre aplicações de programação lógica.

## QUESTÕES DE REVISÃO

1. Quais são os três principais usos da lógica simbólica na lógica formal?
2. Quais são as duas partes de um termo composto?
3. Qual é a forma geral de uma proposição na forma clausal?
4. Dê uma descrição geral (não rigorosa) de resolução e unificação.
5. Quais são as formas das cláusulas de Horn?
6. Qual é o conceito básico da semântica declarativa?
7. Quais são as três formas de um termo Prolog?
8. Qual é a forma sintática e o uso das declarações relativas a fatos e a regras no Prolog?
9. Explique as duas abordagens para casar metas com fatos em um banco de dados.
10. Explique a diferença entre uma busca primeiramente pela profundidade e uma primeiramente pela largura quando se discute como metas múltiplas são satisfeitas.
11. Explique como o backtracking funciona no Prolog.
12. Explique o que há de errado com a instrução Prolog `K is K + 1`.
13. Quais são as duas maneiras pelas quais um programador Prolog pode controlar a ordem de casamento de padrões durante a resolução?
14. Explique a estratégia de programação gerar-e-testar do Prolog.
15. Explique a pressuposição de mundo fechado usada pelo Prolog. Por que ela é uma limitação?
16. Explique o problema da negação com o Prolog. Por que ele é uma limitação?
17. Explique a ligação entre demonstração automática de teoremas e o processo de inferência do Prolog.
18. Explique a diferença entre linguagens baseadas em procedimentos e não baseadas em procedimentos.
19. Explique porque os sistemas Prolog devem fazer backtracking.
20. Qual é a relação entre resolução e unificação no Prolog?

Hidden page

Hidden page

# Bibliografia

- AARM. (1995) *Annotated Ada Reference Manual*. International Standard, ISO/IEC 8652: 1995, Version 6.0. 21 de dezembro de 1994. Intermetrics, Cambridge, MA.
- ACM. (1979) "Part A: Preliminary Ada Reference Manual" e "Part B: Rationale for the Design of the Ada Programming Language." *SIGPLAN Notices*, v. 14, n° 6.
- ACM. (1993a) History of Programming Language Conference Proceedings. *ACM SIGPLAN Notices*, v. 28, n° 3, março.
- ACM. (1993b) "High Performance FORTRAN Language Specification Part 1." FORTRAN Forum, v. 12, n° 4.
- Aho, A. V., R. Sethi e J. D. Ullman. (1986) *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.
- Aho, A. V., B. W. Kernighan e P. J. Weinberger. (1988) *The AWK Programming Language*. Addison-Wesley, Reading, MA.
- Amblter, A. L., D. I. Good, J. C. Browne, W. F. Burger, R. M. Cohen, C. G. Hoch e R. E. Wells. (1977) "Gypsy: A Language for Specification and Implementation of Verifiable Programs." Proceedings of the ACM Conference on Language Design for Reliable Software. *ACM SIGPLAN Notices*, v. 12, n° 3, p. 1-10.
- Andrews, G. R. e F. B. Schneider. (1983) "Concepts and Notations for Concurrent Programming". *ACM Computing Surveys*, v. 15, n° 1, p. 3-43.
- ANSI. (1976) *American National Standard Programming Language PL/I*. ANSI X3.53-1976. American National Standards Institute, Nova York.
- ANSI. (1978a) *American National Standard Programming Language FORTRAN*. ANSI X3.9-1978. American National Standards Institute, Nova York.
- ANSI. (1978b) *American National Standard Programming Language Minimal BASIC*. ANSI X3.60-1978. American National Standards Institute, Nova York.
- ANSI. (1985) *American National Standard Programming Language COBOL*. ANSI X3.23-1985. American National Standards Institute, Nova York.
- ANSI. (1989) *American National Standard Programming Language C*. ANSI X3.159-1989. American National Standards Institute, Nova York.
- ANSI. (1992) *American National Standard Programming Language FORTRAN 90*. ANSI X3.198-1992. American National Standards Institute, Nova York.
- Arden, B. W., B. A. Galler e R. M. Graham. (1961) "MAD at Michigan." *Datamation*, v. 7, n° 12, p. 27-28.
- Backus, J. (1954) "The IBM 701 Speedcoding System". *J. ACM*, v. 1, p. 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, p. 125-132.
- Backus, J. (1978) "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs". *Commun. ACM*, v. 21, n° 8, p. 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. (1962) "Revised Report on the Algorithmic Language ALGOL 60". *Commun. ACM*, v. 6, n° 1, p. 1-17.
- Ben-Ari, M. (1982) *Principles of Concurrent Programming*. Prentice-Hall, Englewood Cliffs, NJ.
- Birtwistle, G.M., O. -J. Dahl, B. Myhrhaug e K. Nygaard. (1973) *Simula BEGIN*. Van Nostrand Reinhold, Nova York.
- 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*, v. 17, n° 6, p. 216-229.
- Bodwin, 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*, v. 17, n° 6, p. 216-229.
- Bohm, C. e G. Jacopini. (1966) "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules". *Commun. ACM*, v. 9, n° 5, p. 366-371.

- Bolsky, M. e D. Korn. (1995) *The New KornShell Command and Programming Language*. Prentice-Hall, Englewood Cliffs, NJ.
- Booch, G. (1987) *Software Engineering with Ada*, 2<sup>a</sup> ed., Benjamin/Cummings, Redwood City, CA.
- Bradley, J. C. (1989) *QuickBASIC and QBASIC Using Modular Structures*. W. C. Brown, Dubuque, IA.
- Brinch Hansen, P. (1973) *Operating System Principles*. Prentice-Hall, Englewood Cliffs, NJ.
- Brinch Hansen, P. (1975) "The Programming Language Concurrent-Pascal". *IEEE Transactions on Software Engineering*, v. 1, n° 2, p. 199-207.
- Brinch Hansen, P. (1977) *The Architecture of Concurrent Programs*. Prentice-Hall, Englewood Cliffs, NJ.
- Brinch Hansen, P. (1978) "Distributed Processes: a Concurrent Programming Concept". *Commun. ACM*, v. 21, n° 11, p. 934-941.
- Campione, M. K. Walrath and A. Humil (2001) *The Java Tutorial*, 3<sup>a</sup> ed., Addison-Wesley, Reading, MA.
- Cardelli, L., J. Donahue, L. Glassman, M. Jordan, B. Kalsow e G. Nelson. (1989) Modula-3 Report (revisado). Digital System Research Center, Palo Alto, CA.
- Chambers, C. e D. Ungar. (1991) "Making Pure Object-Oriented Languages Practical". *SIGPLAN Notices*, v. 26, n° 1, p. 1-15.
- Chomsky, N. (1956) "Three Models for the Description of Language". *IRE Transactions on Information Theory*, v. 2, n° 3, p. 113-124.
- Chomsky, N. (1959) "On Certain Formal Properties of Grammars". *Information and Control*, v. 2, n° 2, p. 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, NJ.
- Clarke, L. A., J. C. Wileden e A. L. Wolf. (1980) "Nesting in Ada Is for the Birds." *ACM SIGPLAN Notices*, v. 15, n° 11, p. 139-145.
- Cleaveland, J. C. (1986) *An Introduction to Data Types*. Addison-Wesley, Reading, MA.
- Cleaveland, J. C. e R. C. Uzgalis. (1976) *Grammars for Programming Languages: What Every Programmer Should Know About Grammar*. American Elsevier, Nova York.
- Clocksin, W. E. e C. S. Mellish. (1997) *Programming in Prolog*, 4<sup>a</sup> ed. Springer-Verlag, Nova York.
- Cohen, J. (1981) "Garbage Collection of Linked Data Structures". *ACM Computing Surveys*, v. 13, n° 3, p. 341-368.
- Cohen, J. (1985) "Describing Prolog by Its Implementation and Computation". *Commun. ACM*, v. 28, n° 12, p. 1311-1324.
- Conway, M. E. (1963). "Design of a Separable Transition-Diagram Compiler". *Commun. ACM*, v. 6, n° 7, p. 396-408.
- Conway, R. e R. Constable. (1976) "PL/CS — A Disciplined Subset of PL/I". Technical Report TR76/293. Department of Computer Science, Cornell University, Ithaca, NY.
- Cornell University. (1977) *PL/C User's Guide, Release 7.6*. Department of Computer Science, Cornell University, Ithaca, NY.
- Correa, N. (1992) "Empty Categories, Chain Binding, and Parsing", p. 83-121, *Principle-Based Parsing*. Eds. R. C. Berwick, S. P. Abney e C. Tenny, Kluwer Academic Publishers, Boston.
- Cuadrado, C. Y. e J. L. Cuadrado. (1985) "Prolog Goes to Work". *BYTE*, agosto de 1985, p. 151-158.
- Dahl, O. -J., E. W. Dijkstra e C. A. R. Hoare. (1972) *Structured Programming*. Academic Press, Nova York.
- Dahl, O. -J. e K. Nygaard. (1967) "SIMULA 67 Common Base Proposal". Norwegian Computing Center Document, Oslo.
- Deliyanni, A. e R. A. Kowalski. (1979) "Logic and Semantic Networks". *Commun. ACM*, v. 22, n° 3, p. 184-192.
- Departament of Defense. (1960) "COBOL, Initial Specifications for a Common Business Oriented Language".
- Departament of Defense. (1961) "COBOL — 1961, Revised Specifications for a Common Business Oriented Language".
- Departament of Defense. (1962) "COBOL — 1961 EXTENDED, Extended Specifications for a Common Business Oriented Language".
- Departament of Defense. (1975a) "Requirements for High Order Programming Languages, STRAWMAN". Julho.
- Departament of Defense. (1975b) "Requirements for High Order Programming Languages, WOODENMAN". Agosto.
- Departament of Defense. (1976) "Requirements for High Order Programming Languages, TINMAN". Junho.
- Departament of Defense. (1977) "Requirements for High Order Programming Languages, IRONMAN". Janeiro.
- Departament of Defense. (1978) "Requirements for High Order Programming Languages, STEELMAN". Junho.
- Departament of Defense. (1980a) "Requirements for High Order Programming Languages, STONEMAN". Fevereiro.
- Departament of Defense. (1980b) "Requirements for the Programming Environment for the Common High Order Language, STONEMAN".
- Departament of Defense. (1990) "Ada 9X Requirements". Office of the Under Secretary of Defense for Acquisition, Washington, DC.
- Deutsch, L. P. e D. G. Bobrow. (1976) "An Efficient Incremental Automatic Garbage Collector". *Commun. ACM*, v. 19, n° 3, p. 522-526.
- Dijkstra, E. W. (1968a) "Goto Statement Considered Harmful". *Commun. ACM*, v. 11, n° 3, p. 147-149.
- Dijkstra, E. W. (1968b) "Cooperating Sequential Processes". In *Programming Languages*, F. Genyus (ed.). Academic Press, Nova York, p. 43-112.

- Dijkstra, E. W. (1972) "The Humble Programmer". *Commun. ACM*, v. 15, nº 10, p. 859-866.
- Dijkstra, E. W. (1975). "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs". *Commun. ACM*, v. 18, nº 8, p. 453-457.
- Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ.
- Dijkstra, E. W. (1982). "Selected Writings on Computing: A personal perspective". Springer-Verlag, Nova York-Berlim.
- Dybvig, R. K. (1996) *The Scheme Programming Language*, 2<sup>a</sup> ed. Prentice-Hall PTR, Upper Saddle River, NJ.
- Ellis, M. A. e B. Stroustrup (1990) *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA.
- Ennals, J. R. (1980) "Logic as a Computer Language for Children". Logic Programming Research Reports. Theory of Computing Research Group, Department of Computing, Imperial College of Science and Technology, Londres.
- Farber, D. J., R. E. Griswold e F. P. Polansky. (1964) "SNOBOL, a String Manipulation Language". *J. ACM*, v. 11, nº 1, p. 21-30.
- Farrow, R. (1982) "LINGUIST 86: Yet Another Translator Writing System Based on Attribute Grammars". *ACM SIGPLAN Notices*, v. 17, nº 6, p. 160-171.
- Feuer, A. e N. Gehani. (1982) "A Comparison of the Programming Languages C and Pascal". *ACM Computing Surveys*, v. 14, nº 1, p. 73-92.
- Fischer, C. N., G. F. Johnson, J. Mauney, A. Pal e D. L. Stock. (1984) "The Poe Language-Based Editor Project." *ACM SIGPLAN Notices*, v. 19, nº 5, p. 21-29.
- Fischer, C. N. e R. J. LeBlanc. (1977) "UW-Pascal Reference Manual". Madison Academic Computing Center, Madison, WI.
- Fischer, C. N. e R. J. LeBlanc. (1980) "Implementation of Runtime Diagnostics in Pascal". *IEEE Transactions on Software Engineering*, SE-6, nº 4, p. 313-319.
- Fischer, C. N. e R. J. LeBlanc. (1991) *Crafting a Compiler*. Benjamin/Cummings, Menlo Park, CA.
- Flanagan, D. (1998) *JavaScript: The Definitive Guide*. O'Reilly Publ. Co. Sebastopol, CA.
- Floyd, R. W. (1967) "Assigning Meanings to Programs". *Proceedings Symposium Applied Mathematics, in Mathematical Aspects of Computer Science*, ed. J.T. Schwartz American Mathematical Society, Providence, RI.
- Frege, G. (1982) "Über Sinn und Bedeutung". *Zeitschrift für Philosophie und Philosophisches Kritik*, v. 100, p. 25-50.
- Friedl, J. E. F. (1997) *Mastering Regular Expressions*. O'Reilly Publ. Co., Sebastopol, CA, 368 p.
- Friedman, D. P. e D. S. Wise. (1979) "Reference Counting's Ability to Collect Cycles Is Not Insurmountable". *Information Processing Letters*, v. 8, nº 1, p. 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. Republicado (1982) pela North-Holland Publishing, Amsterdã.
- Gehani, N. (1983) *Ada: An Advanced Introduction*. Prentice-Hall, Englewood Cliffs, NJ.
- Ghezzi, C. e M. Jazayeri. (1987) *Programming Language Concepts*, 2<sup>a</sup> ed. Wiley, Nova York.
- Gilman, L. e A. J. Rose. (1976) *APL: An Interative Approach*, 2<sup>a</sup> ed. J. Wiley, Nova York.
- Goldberg, A. e D. Robson. (1983) *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA.
- Goldberg, A. e D. Robson. (1989) *Smalltalk-80: The Language*. Addison-Wesley, Reading, MA.
- Goodenough, J. B. (1975) "Exception Handling: Issues and Proposed Notation." *Commun. ACM*, v. 18, nº 12, p. 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.
- Gordon, M. (1979) *The Denotational Description of Programming Languages, An Introduction*. Springer-Verlag, Berlin-Nova York.
- Gosling, J., B. Joy e G. Steele. (1996) *The Java Language Specification*. Addison-Wesley, Reading, MA.
- Gries, D. (1981) *The Science of Programming*. Springer-Verlag, Nova York.
- Griswold, R. E. e M. T. Griswold. (1983) *The ICON Programming Language*. Prentice-Hall, Englewood Cliffs, NJ.
- Griswold, R. E., F. Poage e I. P. Polonsky. (1971) *The SNOBOL 4 Programming Language*, 2<sup>a</sup> ed. Prentice-Hall, Englewood Cliffs, NJ.
- Hammond, P. (1983) APES: A User Manual. Department of Computing Report 82/9. Imperial College of Science and Technology, Londres.
- Henderson, P. (1980) *Functional Programming: Application and Implementation*. Prentice-Hall, Englewood Cliffs, NJ.
- Hoare, C. A. R. (1969) "An Axiomatic Basis of Computer Programming". *Commun. ACM*, v. 12, nº 10, p. 576-580.
- Hoare, C. A. R. (1972) "Proof of Correctness of Data Representations". *Acta Informatica*, v. 1, p. 271-281.
- Hoare, C. A. R. (1973) "Hints on Programming Language Design". *Proceedings ACM SIGACT/SIGPLAN Conference on Principles of Programming Languages*. Publicado também como Technical Report STAN-CS-73-03, Stanford University Computer Science Department.
- Hoare, C. A. R. (1974) "Monitors: An Operating System Structuring Concept". *Commun. ACM*, v. 17, nº 10, p. 549-557.
- Hoare, C. A. R. (1978) "Communicating Sequential Processes". *Commun. ACM*, v. 21, nº 8, p. 666-677.

- Hoare, C. A. R. (1981) "The Emperor's Old Clothes". *Commun. ACM*, v. 24, nº 2, p. 75-83.
- Hoare, C. A. R. e N. Wirth. (1973) "An Axiomatic Definition of the Programming Language Pascal". *Acta Informatica*, v. 2, p. 335-355.
- Hogger, C. J. (1984) *Introduction to Logic Programming*. Academic Press, Londres.
- Holt, R. C., G. S. Graham, E. D. Lazowska e M. A. Scott. (1978) *Structured Concurrent Programming with Operating Systems Applications*. Addison-Wesley, Reading, MA.
- Horn, A. (1951) "On Sentences Which Are True of Direct Unions of Algebras". *J. Symbolic Logic*, v. 16, p. 14-21.
- Hudak, P. e J. Fasel. (1992) "A Gentle Introduction to Haskell". *ACM SIGPLAN Notices*, 27(5), maio de 1992, p. T1-T53.
- Huskey, H. K., R. Love e N. Wirth. (1963) "A Syntactic Description of BC NELIAC". *Commun. ACM*, v. 6, nº 7, p. 367-375.
- IBM. (1954) "Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN". IBM Corporation, Nova York.
- IBM. (1956) "Programmer's Reference Manual, The FORTRAN Automatic Coding System for the IBM 704 EDPM". IBM Corporation, Nova York.
- IBM. (1964) "The New Programming Language", IBM UK Laboratories.
- Ichbiah, J. D., J. C. Heillard, 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*, v. 14, nº 6, Part B.
- IEEE. (1985) "Binary Floating-Point Arithmetic." IEEE Standard 754, IEEE, Nova York.
- Ingerman, P. Z. (1967). "Panini-Backus Form Suggested". *Commun. ACM*, v. 10, nº 3, p. 137.
- Intermetrics. (1993) Programming Language Ada, Draft, Version 4.0. Cambridge, MA.
- ISO. (1982) *Specification for Programming Language Pascal*. ISO7185-1982. International Organization for Standardization, Genebra, Suíça.
- Iverson, K. E. (1962). *A Programming Language*. John Wiley, Nova York.
- Jensen, K. e N. Wirth. (1974) *Pascal Users Manual and Report*. Springer-Verlag, Berlim.
- Johnson, S. C. (1975) "Yacc — Yet Another Compiler Compiler". Computing Science Report 32, AT&T Bell Laboratories, Murray Hill, NJ.
- Jones, N. D. (ed.) (1980) *Semantic-Directed Compiler Generation*. Lecture Notes in Computer Science, v. 94. Springer-Verlag, Heidelberg, RFA.
- Kay, A. (1969) The Reactive Engine. Tese de Ph.D. Universidade de Utah, setembro.
- Kernighan, B. W. e R. Pike. (1984) *The UNIX Programming Environment*. Prentice-Hall, Englewood Cliffs, NJ.
- Kernighan, B. W. e D. M. Ritchie. (1978) *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ.
- Knuth, D. E. (1967) "The Remaining Trouble Spots in ALGOL 60". *Commun. ACM*, v. 10, nº 10, p. 611-618.
- Knuth, D. E. (1968a) "Semantics of Context-Free Languages". *Mathematical Systems Theory*, v. 2, nº 2, p. 127-146.
- Knuth, D. E. (1968b) *The Art of Computer Programming*, v. I, 2<sup>a</sup> ed. Addison-Wesley, Reading, MA.
- Knuth, D. E. (1974) "Structured Programming with GOTO Statements". *ACM Computing Surveys*, v. 6, nº 4, p. 261-301.
- Knuth, D. E. (1981) *The Art of Computer Programming*, v. II, 2<sup>a</sup> ed. Addison-Wesley, Reading, MA.
- Knuth, D. E. e Luis Trabb Pardo. (1977) "Early Development of Programming Languages" Em *Encyclopedia of Computer Science and Technology*, v. 7. Dekker, Nova York, p. 419-493.
- Kowalski, R. A. (1979) *Logic for Problem Solving*. Artificial Intelligence Series, v. 7. Elsevier-North Holland, Nova York.
- Lampson, B. W. (1983) "A Description of the Cedar Language". Tech. Report CSL-83-15. Xerox Palo Alto Research Center, dezembro.
- Lampson, B. W., J. J. Horning, R. L. London, J. G. Mitchell e G. J. Popek. (1977) "Report on the Programming Language Euclid." *ACM SIGPLAN Notices*, v. 12, nº 2 (Relatório Revisado, XEROX PARC Technical Report CSL78-2).
- Laming, J. H., Jr. e N. Zierler. (1954) "A Program for Translation of Mathematical Equations for Whirlwind I". Memorando de Engenharia E-364. Instrumentation Laboratory, Massachusetts Institute of Technology, Cambridge, MA.
- Ledgard, H. (1984) *The American Pascal Standard*. Springer-Verlag, Nova York.
- Ledgard, H. F. e M. Marcotty. (1975) "A Genealogy of Control Structures". *Commun. ACM*, v. 18, nº 11, p. 629-639.
- Liskov, B. e A. Snyder. (1979) "Exception Handling in CLU". *IEEE Transactions on Software Engineering*, v. SE-5, nº 6, p. 546-558.
- Lomet, D. (1975) "Scheme for Invalidating References to Freed Storage". *IBM J. of Research and Development*, v. 19, p. 26-35.
- MacLaren, M. D. (1977) "Exception Handling in PL/I". *ACM SIGPLAN Notices*, v. 12, nº 3, p. 101-104.
- Marcotty, M., H. F. Ledgard e G. V. Bochmann. (1976) "A Sampler of Formal Definitions". *ACM Computing Surveys*, v. 8, nº 2, p. 191-276.
- Mather, D. G. e S. V. Waite (eds.). (1971) *BASIC*, 6<sup>a</sup> ed. University Press of New England, Hanover, NH.
- McCarthy, J. (1960) "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". *Commun. ACM*, v. 3, nº 4, p. 184-195.
- McCarthy, J., P. W. Abrahams, D. J. Edwards, T. P. Hart e M. Levin. (1995) *LISP 1.5 Programmer's Manual*. 2<sup>a</sup> ed. MIT Press, Cambridge, MA.
- McCracken, D. (1970) "Whither API". *Datamation*, 15 de setembro, p. 53-57.

- Meyer, B. (1992) *Eiffel the Language*. Prentice-Hall, Englewood Cliffs, NJ.
- Microsoft. (1991) *Microsoft Visual Basic Language Reference*. Documento DB20664-0491, Redmond, WA.
- Milner, R., M. Tofte e R. Harper. (1990) *The Definition of Standard ML*. MIT Press, Cambridge, MA.
- Milos, D., U. Pleban e G. Loegel. (1984) "Direct Implementation of Compiler Specifications". *ACM Principles of Programming Languages* 1984, p. 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, CA.
- Mössenbock, H. (1993) *Object-Oriented Programming in Oberon-2*. Springer-Verlag, Nova York.
- 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. Reproduzido (1982) by North-Holland Publishing, Amsterdam.
- Naur, P. (ed.) (1960) "Report on the Algorithmic Language ALGOL 60". *Commun. ACM*, v. 3, nº 5, p. 299-314.
- Newell, A. e H. A. Simon. (1956) "The Logic Theory Machine — A Complex Information Processing System". *IRE Transactions on Information Theory*, v. IT-2, nº 3, p. 61-79.
- Newell, A. e F. M. Tonge. (1960) "An Introduction to Information Processing Language V". *Commun. ACM*, v. 3, nº 4, p. 205-211.
- Nilsson, N. J. (1971) *Problem Solving Methods in Artificial Intelligence*. McGraw-Hill, Nova York.
- Osterhout, J. K. (1994) *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA.
- Pagan, F. G. (1981) *Formal Specifications of Programming Languages*. Prentice-Hall, Englewood Cliffs, NJ.
- Papert, S. (1980). *Mindstorms: Children, Computers and Powerful Ideas*. Basic Books, Nova York.
- Perlis, A. e K. Samelson. (1958) "Preliminary Report — International Algebraic Language". *Commun. ACM*, v. 1, nº 12, p. 8-22.
- Peyton Jones, S. L. (1987) *The Implementation of Functional Programming Languages*. Prentice-Hall, Englewood Cliffs, NJ.
- Polivka, R. P. e S. Pakin. (1975) *APL: The Language and Its Usage*. Prentice-Hall, Englewood Cliffs, NJ.
- Pratt, T. W. (1984) *Programming Languages: Design and Implementation*. 2<sup>a</sup> ed. Prentice-Hall, Englewood Cliffs, NJ.
- Pratt, T. W. e M. V. Zelkowitz (2001). *Programming Languages: Design and Implementation*. 4<sup>a</sup> ed. Prentice-Hall, Englewood Cliffs, NJ.
- Rees, J. e W. Clinger. (1986) "Revised Report on the Algorithmic Language Scheme". *ACM SIGPLAN Notices*, v. 21, nº 12, p. 37-79.
- Remington-Rand. (1952) "UNIVAC Short Code." Coleção não-publicada de notas repetidas. Prefácio de A.B. Tonik, datado em 25 de outubro de 1955 (1 p.); Prefácio de J.R. Logan, não-datado, mas aparentemente de 1952 (1 p.); Exposição preliminar, 1952? (22 pp., em que as páginas 20-22 parecem ser uma substituição posterior); Informação complementar sobre o Short code, tópico um (7 páginas); Adendos # 1, 2, 3, 4 (9 páginas).
- Richards, M. (1969) "BCPL: A Tool for Compiler Writing and Systems Programming". *Proc. AFIPS SJCC*, v. 34, p. 557-566.
- Robinson, J. A. (1965) "A Machine-Oriented Logic Based on the Resolution Principle". *Journal of the ACM*, v. 12, p. 23-41.
- Roussel, P. (1975) "PROLOG: Manual de Reference et D'utilisation". Relatório de Pesquisa. Artificial Intelligence Group, Univ. of Aix-Marseille, Luminy, França.
- Rovner, P. (1986) "Extending Modula-2 to Build Large, Integrated Systems". *IEEE Software*, v. 3, nº 6, Novembro.
- Rubin, F. (1987) "GOTO Statement Considered Harmful/considered harmful" (carta ao editor). *Commun. ACM*, v. 30, nº 3, p. 195-196.
- Rutishauser, H. (1967) *Description of ALGOL 60*. Springer-Verlag, Nova York.
- Sammet, J. E. (1969) *Programming Languages: History and Fundamentals*. Prentice-Hall, Englewood Cliffs, NJ.
- Sammet, J. E. (1976) "Roster of Programming Languages for 1974-75". *Commun. ACM*, v. 19, nº 12, p. 655-669.
- Schorr, H. e W. Waite. (1967) "An Efficient Machine Independent Procedure for Garbage Collection in Various List Structures". *Commun. ACM*, v. 10, nº 8, p. 501-506.
- Scott, D. S. e C. Strachey. (1971) "Towards a Mathematical Semantics for Computer Language". *Proceedings, Symposium on Computers and Automation*, ed. J. Fox Polytechnic Institute of Brooklyn Press, Nova York, p. 19-46.
- Sebesta, R. W. (1991) *VAX Structured Assembly Language Programming*, 2<sup>a</sup> ed. Benjamin/Cummings Publ. Co., Redwood City, CA.
- Sergot, M. J. (1983) "A Query-the-User Facility for Logic Programming". In *Integrated Interactive Computer Systems*, eds. P. Degano e E. Sandewall. North-Holland Publishing, Amsterdam.
- Sewry, D. A. (1984b) "Modula-2 and the Monitor Concept". *ACM SIGPLAN Notices*, v. 19, nº 11, p. 33-41.
- Shaw, C. J. (1963) "A Specification of JOVIAE". *Commun. ACM*, v. 6, nº 12, p. 721-736.
- Sommerville, I. (1992) *Software Engineering*, 4<sup>a</sup> ed., Addison-Wesley, Reading, MA.
- Steele, G. L., Jr. (1984) *Common LISP*. Digital Press, Burlington, MA.
- Stoy, J. E. (1977) *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*. MIT Press, Cambridge, MA.
- Stroustrup, B. (1983) "Adding Classes to C: An Exercise in Language Evolution". *Software — Practice and Experience*, v. 13, p. 139-161.
- Stroustrup, B. (1984) "Data Abstraction in C". *AT & T Bell Laboratories Technical Journal*, v. 63, nº 8.

- Stroustrup, B. (1986) *The C++ Programming Language*. Addison-Wesley, Reading, MA.
- Stroustrup, B. (1988) "What is Object-Oriented Programming?". *IEEE Software*, maio de 1988, p. 10-20.
- Stroustrup, B. (1991) *The C++ Programming Language*, 2d ed. Addison-Wesley, Reading, MA.
- Stroustrup, B. (1997) *The C++ Programming Language*, 3d ed. Addison-Wesley, Reading, MA.
- Sussman, G. J. e G. L. Steele, Jr. (1975) "Scheme: An Interpreter for Extended Lambda Calculus". MIT AI Memo N° 349 (dezembro de 1975).
- Suzuki, N. (1982) "Analysis of Pointer 'Rotation'". *Commun. ACM*, v. 25, n° 5, p. 330-335.
- Tanenbaum, A. S. (1978) "A Comparison of Pascal and ALGOL 68". *Computer Journal*, v. 21, p. 316-323.
- Tanenbaum, A. S. (1990) *Structured Computer Organization*, 3<sup>a</sup> ed. Prentice-Hall, Englewood Cliffs, NJ.
- Tanenbaum, A. S., Y. Langsam e M. J. Augenstein (1990) *Data Structures Using C*. Prentice-Hall, Englewood Cliffs, NJ.
- Taylor, W., L. Turner e R. Waychoff (1961) "A Syntactic Chart of ALGOL 60". *Commun. ACM*, v. 4, p. 393.
- Teitelbaum, T e T. Reps. (1981) "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment". *Commun. ACM*, v. 24, n° 9, p. 563-573.
- Teitelman, W. (1975) *INTERLISP Reference Manual*. Xerox Palo Alto Research Center, Palo Alto, CA.
- Thompson, S. (1996) *Haskell: The Craft of Functional Programming*. Addison-Wesley, Reading, MA, 500 páginas.
- Turner, D. (1986) "An Overview of Miranda". *ACM SIGPLAN Notices*, v. 21, n° 12, p. 158-166.
- Turner, D. (1990) (ed.) *Research Topics in Functional Programming*. Addison-Wesley, Reading, MA.
- Ullman, J. D. (1994) *Elements of ML Programming*. Prentice-Hall, Englewood Cliffs, NJ.
- Van Emden, M. H. (1980) "McDermott on Prolog: A Rejoinder". *SIGART Newsletter*, n° 72, agosto, p. 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*, v. 14, n° 2, p. 79-218.
- Wall, L., T. Christiansen e R. L. Schwartz. (1996) *Programming Perl*, 2d ed. O'Reilly & Associates, Sebastopol, CA.
- Wall, L., T. Christiansen, and J. Orwart (2000) *Programming Perl*, 3d ed. O'Reilly & Associates, Sebastopol, CA.
- Warren, D. H. D., L. M. Pereira e F. C. N. Pereira. (1977) "Prolog: The Language and Its Implementation Compared to LISP". *ACM SIGPLAN Notices*, v. 12, n° 8 e *ACM SIGART Newsletter*, v. 6, n° 4.
- Warren, D. H. D., L. M. Pereira e F. C. N. Pereira. (1979) "User's Guide to DEC System-10 Prolog". Occasional Paper 15. Department of Artificial Intelligence, Univ. of Edinburgh, Escócia.
- Watt, D. A. (1979) "An Extended Attribute Grammar for Pascal". *ACM SIGPLAN Notices*, v. 14, n° 2, p. 60-74.
- Wegner, P. (1972) "The Vienna Definition Language". *ACM Computing Surveys*, v. 4, n° 1, p. 5-63.
- Weissman, C. (1967) *LISP 1.5 Primer*. Dickenson Press, Belmont, CA.
- Welsh, J., M. J. Sneeringer e C. A. R. Hoare. (1977) "Ambiguities and Insecurities in Pascal". *Software—Practice and Experience*, v. 7, n° 6, p. 685-696.
- Wexelblat, R. L. (ed.). (1981) *History of Programming Languages*. Academic Press, Nova York.
- Wheeler, D. J. (1950) "Programme Organization and Initial Orders for the EDSAC". *Proc. R. Soc. London, Ser. A*, v. 202, p. 573-589.
- Wilkes, M. V. (1952) "Pure and Applied Programming". In *Proceedings of the ACM National Conference*, v. 2, Toronto, p. 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, MA.
- Wilkes, M. V., D. J. Wheeler e S. Gill. (1957) *The Preparation of Programs for an Electronic Digital Computer*, 2<sup>a</sup> ed. Addison-Wesley, Reading, MA.
- Wirth, N. (1971) "The Programming Language Pascal". *Acta Informatica*, v. 1, n° 1, p. 35-63.
- Wirth, N. (1973) *Systematic Programming: An Introduction*. Prentice-Hall, Englewood Cliffs, NJ.
- Wirth, N. (1975) "On the Design of Programming Languages". *Information Processing 74 (Proceedings of IFIP Congress 74)*, North Holland, Amsterdam, p. 386-393.
- Wirth, N. (1977) "Modula: A Language for Modular Multi-Programming". *Software — Practice and Experience*, v. 7, p. 3-35.
- Wirth, N. (1985) *Programming in Modula-2*, 3<sup>a</sup> ed. Springer-Verlag, Nova York.
- Wirth, N. (1988) "The Programming Language Oberon". *Software — Practice and Experience*, v. 18, n° 7, p. 671-690.
- Wirth, N. e C. A. R. Hoare. (1966) "A Contribution to the Development of ALGOL". *Commun. ACM*, v. 9, n° 6, p. 413-431.
- Wulf, W. A., D. B. Russell e A. N. Habermann. (1971) "BLISS: A Language for Systems Programming". *Commun. ACM*, v. 14, n° 12, p. 780-790.
- Zuse, K. (1972) "Der Plankalkül". Manuscrito preparado em 1945, publicado em *Berichte der Gesellschaft für Mathematik und Datenverarbeitung*, N° 63 (Bonn, 1972); Parte 3, 285 p. Tradução inglesa para todas as páginas, menos 176-196 no n° 106 (Bonn, 1976), p. 42-244.

# Índice

## A

Abordagem ansiosa à reclamação de armazenamento, 260  
Abordagem preguiçosa à reclamação de armazenamento, 260  
Abrangências de lista na Haskell, 578–579  
Abstração. Ver também Abstração de dados  
capacidade de escrita (*writability*) e, 29  
conceito de, 410–411  
de processo, 410–411  
para estruturas sintáticas, 113–114  
Abstração de dados, 34–35. Capítulo 10. Ver também Tipo de dados abstratos  
Acesso  
a variáveis não-locais, 363–365  
funções para matriz, 235–237  
profundo, 398–402  
raso, 401–403  
ACM, 63, 64  
*Ad hoc*, polimorfismo, 357–359  
*Ad hoc*, vinculação, 356  
Ada 95, 95–96  
avaliação do suporte para OOP, 471–472  
características gerais da, 469–470  
classes na, 469–470  
comparação com o C++, 471–472  
comunicação assíncrona na, 510–502  
concorrência na, 508–512  
embasamento histórico da, 94–96  
herança na, 469–470  
objetos protegidos na, 509–511  
programação orientada a objeto em, 468–470  
tipos-marcados (*tagged*) em, 469–470  
vinculação dinâmica na, 470–471  
Ada  
acessando ambientes de referenciamento não-locais na, 364–365  
associativa, 237–239  
associatividade de operadores na, 272  
avaliação, 93–95  
avaliação curto-circuito na, 283–284  
avaliação da manipulação de exceções na, 536–537  
case na, 305

comandos protegidos, 320–322  
constant na, 204–205  
construção de seletor, 304–306  
continuação na, 533–535  
embasamento histórico da, 91–92  
faixa discreta na, 311  
fatias, 231–232  
finalização de tarefas da, 507–508  
formas de rótulo em, 319–320  
implementação de parâmetros em, 347–348  
inicialização de matrizes em, 29  
inicialização de variáveis em, 204–205  
instrução for na, 311  
instrução seletora em, 299–300  
laços infinitos, 315–317  
manipulação de exceções na, 30, 532–537  
mecanismo de controle de laços localizado pelo usuário na, 315–317  
mensagens assíncronas, 510–511  
métodos de passagem de parâmetros da, 343–344  
operações com registros na, 240–242  
operadores booleanos na, 281–282  
operadores sobrecarregados na, 367  
pacotes em, 417  
pacotes genéricos em, 425–426  
parâmetros de palavra-chave, 333  
parâmetros padrão na, 333–334  
parâmetros que são matrizes multidimensionais na, 349  
passagem de mensagens em, 501–505  
ponteiros em, 254  
pragma na, 508  
precedência de operadores na, 270–271  
prioridades de tarefas na, 507–508  
processo de projeto, 92–93  
programas errôneos em, 348  
seletores múltiplos em, 304–306  
semáforos binários na, 507–509  
subprogramas genéricos em, 358–362  
subprogramas sobrecarregados na, 357–358  
tarefas, 501–504  
tipificação forte da, 189–191  
tipos de dados abstratos na, 417–420  
tipo acesso, 250, 254

- tipos de dados abstratos parametrizados em, 425–427  
 tipos derivados na, 192  
 tipos enumeração na, 223  
 tipos privados limitados em, 418  
 tipos privados na, 418  
 tipos união na, 246–247  
 unidades de biblioteca em, 362  
 use, 420  
 variável variant restrita na, 245–246  
 vinculação de exceções a manipuladores em, 532–534  
 visão geral da, 93–94  
 with, 411
- Adição unária, 269–270
- Agregados de dados constantes, 230
- Aho, Al, 20–21
- AIMACO, 68–69
- ALGOL 58, 64–66  
 receção do, 65–66  
 visão geral do, 64–65
- ALGOL 60  
 avaliação do, 66–68  
 embasamento histórico do, 63–64  
 estrutura de bloco em, 66  
 instrução for em, 308–311  
 instruções compostas em, 296–298  
 laços controlados por contador, 308–311  
 processo de projeto, 63  
 processo de projeto inicial do, 64  
 visão geral do, 66–67
- ALGOL 68  
 avaliação do, 82–84  
 matrizes dinâmicas no, 82–83  
 processo de projeto do, 81–82  
 tipos de dados definidos pelo usuário, 82–83  
 tipos união em, 129–131  
 visão geral do, 82–83
- ALGOL Bulletin, 66
- ALGOL W, 84, 213  
 métodos de passagem de parâmetros do, 343–344
- Algoritmos  
 de análise, 178–180  
 de marcação, 261–262  
 desloca-reduz, 186–188
- Algoritmos de coleta de lixo, 260–263
- ALIGN em HPF, 517–518
- Alocação, 185–186  
 células de tamanho fixo, 259–262  
 células de tamanho variável, 262–263
- Ambiente de referenciamento, 200–203  
 confusão ao implementar nomes de subprogramas, 403–405  
 local, 336–337  
 de subprogramas que são passados como parâmetros, 337–357, 403–404, 404–405
- Ambientes não-locais, acessando de, 363–366
- Ambigüidade gramatical, 117–118
- Amigos, 461–462
- Analisadores ED, 170–174
- Análise léxica, 38–39
- Análise sintática, 176–177  
 algoritmos, 178–180  
 analisadores baixo-cima, 160, 166–179  
 analisadores cima-baixo, 177–179, 184–193  
 classe de gramática EE, 182–185  
 descendente recursiva, 179–185
- Ancestrais estáticos, 194–195
- Antecedente, 588–589
- Apelidos, 180  
 com parâmetros de passagem por referência, 341–342
- API, 36, 79–80  
 associatividade de operadores em, 272  
 expressões, 272  
 operações com matrizes em, 230–231  
 origens e características da, 79–80  
 precedência de operadores na, 270–271  
 vinculação dinâmica de tipos na, 183–184
- Aplicações  
 científicas, 17–20  
 comerciais, 20  
 de linguagens funcionais, 579–581  
 de linguagens lógicas, 611–613
- append,  
 inteligência artificial, 20–21  
 no Prolog, 603  
 na Scheme, 569–570
- Apply-to-all, forma funcional  
 na matemática, 554–555  
 na Scheme, 571–572
- APT, linguagem, 21–22
- Armazenamento dinâmico, alocação e desalocação de, 185–186
- Arquitetura, 485–486
- Arquitetura de computador, 33–35
- Arquitetura de multiprocessador, 485–486
- Arquitetura de von Neumann, 34–35, 41
- Árvores de análise, 116–117  
 amplamente atribuídos, 126–128  
 ilustrando a associatividade da adição, 121  
 ilustrando a precedência de operadores, 119–120
- ASCII, 215–216
- Asserções, 133–135
- Association for Computing Machinery (ACM), 63, 64
- Associatividade, 270–274
- Ativações de subprogramas, 379–380
- Atribuição de modo misto, 288–289
- Atribuição simples, 116–117, 285
- Atribuições do operador unário, 286–287
- Atributos  
 herdados, 126–128  
 intrínsecos, 126–128  
 na Eiffel, 471–473  
 sintetizados, 126–128  
 vinculação a variáveis de, 181–185
- Autômato finito, 150–151
- Avaliação preguiçosa na Haskell, 576–578, 578–581
- awk, linguagem, 20–21
- Axioma, 134–136

**B**

Babbage, Charles, 92  
 Backtracking, técnica de, no Prolog, 598  
 Backus, John, 47, [52-53](#), 113-114  
 Backus-Naur, forma  
     ambigüidade na, [117-118](#)  
     derivações na, [114-115](#)  
     estendida, [123-125](#)  
     fundamentos da, 113-117  
     origens da, 113-114  
     regras recursivas na, [114-115](#)  
 Bancos de dados Prolog, ordem de, 606  
 BASIC  
     avaliação do, 74-76  
     processo de projeto do, [73-74](#)  
 BASIC-PLUS, 74  
 Bauer, Fritz, [267](#)  
 BCD, 215-216  
 BCPL, [86](#)  
*Bell Laboratories*, [86](#), 99  
 BLISS, [20-21](#)  
 Blocos, [195-196](#), 295-296  
     COMMON, 365-366  
     implementação de 397-399  
     na Smalltalk, 446-448  
 BNE Ver Backus-Naur, forma  
 Borland C++, [44-45](#)  
**break**, 303-305  
 Brinch Hansen, Per, 494-496  
 Busca primeiramente pela largura (*breadth-first*), 598  
 Busca primeiramente pela profundidade (*depth-first*), 598  
 Buscar-executar, ciclo, [41](#)  
 Byron, Augusta Ada, 92

**C**

Cabeçalho de subprograma, 331  
 Cadeia de chamadas, 383  
 Cadeias  
     implementação de, [220-221](#)  
     opções de tamanho de, [219](#)  
 Cadeias de caracteres, [204-239](#)  
     avaliação de, [219](#)  
     comparação de padrões com, 217-218  
     concatenação de, 216-218  
     implementação de, [220-221](#)  
     opções de tamanho de strings, [219](#)  
     operações para, 215-219  
     questões de projeto de, 215-217  
     referências de substrings a, 215-217  
 Cadeias de tamanho dinâmico, [219](#)  
 Cadeias de tamanho dinâmico limitado, [219](#)  
 Cadeias de tamanho estático, [219](#)  
 Cálculo de predicados, 586-590  
     demonstrando teoremas em, 589-591  
     forma clausal no, 586-590  
     proposições no, 587-589

Cambridge Polish [notação], 558  
 Campos de registros, referências a, [239-242](#)  
 Capacidade de escrita (*writability*)  
     abstração e, [29](#)  
     expressividade e, [29-30](#)  
     simplicidade e ortogonalidade e, 28-29  
     tipos string e, [219](#)  
 CAR, função Scheme, 561  
 Carregador, [41](#)  
**case**, 301-304  
 Cast, [279-280](#)  
 CBL, [69-70](#)  
 CDR  
     alocação e desalocação de, [259-260](#)  
     função Scheme, 561  
     tamanho fixo, [259-262](#)  
 Células de tamanho único, [259-262](#)  
 Células de tamanho variável, [262-264](#)  
     alocação e desalocação de, [262-263](#)  
 Cfront, 100  
 Chain-offset, 388  
 Chamadas, semântica das, [376](#)  
 Chomsky, Noam, 112-113  
 Church, Alonzo, 553-554  
 Cii Honeywell/Bull, projeto da linguagem, 92  
 Classe  
     métodos, 434  
     objetos, 434  
     variáveis, 435  
 Classe derivada, 434  
 Classe virtual, 435-436  
 Classe-pai, 434  
 Classes  
     modeladas, 425-427  
     na Ada 95, 469-470  
     no C++, [420-424](#)  
     no Java, 422-417  
     no SIMULA 74, 416-417  
     na Smalltalk, 449-450  
 Classes abstratas no C++, 464-465  
 Cláusula **accept** aberta, 505  
 Cláusulas **accept** estendidas, 505  
 Cláusulas **accept** fechadas, 505-506  
 Cláusulas **accept** na Ada, 502  
 Cláusulas **accept** protegidas, 505-506  
 Cláusulas de conformidade, [243-244](#)  
 Cláusulas de Horn com e sem cabeça, 591  
 CLOS, 35-36  
 COBOL, [20](#), 68-69  
     avaliação do, [70-71](#)  
     embasamento histórico do, 68-70  
     estruturas de registro no, [239-240](#)  
     operações de registro no, 240-242  
     processo de projeto do, [69-70](#)  
 Código de máquina  
     execução de, [41](#)  
     programação em, [50-51](#)  
 Coerção, [189](#)  
 Cohen, Jacques, 612-613

Coincidência de padrões, 217-219  
 Colmerauer, Alain, 90, 593  
 Comandos protegidos, 320-324  
 COMMON, blocos, 365-366  
 Common Gateway Interface, 21-22  
 COMMON LISP, 62, 316-318, 573-577  
*Communications of ACM*, 66  
 Compatibilidade de tipos  
     por estrutura, 191-194  
     por nome, 191-194  
 Compilação  
     independente, 363-364  
     processo de, 38-42  
     separada, 362  
     unitária, 411  
 Compiladores  
     análise léxica dos, 176-177  
     análise sintática dos, 176-178  
     otimizando, 38-39  
     parte de análise sintática dos, 38-39  
     projeto de, 38-42  
 Computadores virtuais, 38-39  
 Concatenação, 216-218  
 Concorrência  
     categorias de, 486-487  
     conceitos fundamentais da, 487-491  
     física, 486  
     lógica, 486  
     níveis de, 484-485  
     nível de instrução, 516-517  
     nível de subprograma, 486-491  
     questões de projeto de, 490-492  
     razões para estudar, 486-491  
 Concurrent Pascal, 496-497  
 COND, função Scheme, 566-568  
 Condições de erro em tempo de execução, 522  
 Conectores lógicos, 587-588  
 Confiabilidade, 30-31  
     apelidos e, 31  
     legibilidade e capacidade de escrita, 31, 128  
     manipulação de exceções e, 30  
     verificação de tipos e, 30  
 Conjunção, 594-595  
 Conjunto domínio de uma função, 553  
 Conjunto imagem de uma função, 553  
 Conjuntos  
     avaliação de, 248-250  
     implementação de, 249-250  
     no Pascal e no Modula-2, 247-249  
     tipo-básico, 246-249  
 CONS, função Scheme, 561-562, 564  
 Consequente, 588-589  
**constant**, 204-205  
 Constantes  
     manifestas, 204-205  
     nomeadas, 202-205  
 Construções de seleção múltiplas, 299-300  
     modernas, 301-307  
     primeiras, 300-302  
     questões de projeto de, 299-301  
     tridimensionais, 300-301

Contadores de referência, 260  
 Continuação, 525-526  
     na manipulação de exceções da Ada, 533-535  
     na manipulação de exceções do C++, 538-539  
     na manipulação de exceções do Java, 541-544  
     na manipulação de exceções da PL/I, 563-567  
**continue**, 315-318  
 Controle da ordem de resolução no Prolog, 606-608  
 Controle simétrico. Ver Co-rotinas.  
 Conversão de tipos, 278-280  
     de alargamento, 278  
     de estreitamento, 278  
     explícita, 278-280  
     implícita, 189, 278-280  
     Ver também Coersão  
 Cooper, Jack, 92  
 Co-rotinas, 367-370  
 Corpo, pacotes, 417  
 Correção parcial, 140-142  
 Correção total, 140-142  
 Correspondência biunívoca, 553  
 CPU, 41  
 CSP, linguagem, 503  
 Currie, Malcolm, 91-92  
 Curto-circuito, avaliação, 283-284  
 Custo/benefício de projeto, 36-37  
 Custo da escrita de programas de, 30-32  
 Custo de uma linguagem, 30-33  
 Custos de treinamento, 30-31  
 Cut, operador, 606-607

**D**

Dahl, Ole-Johan, 80-409  
 Deadlock. Ver Enlace mortal  
 Declaração  
     equivalência, 192-193  
     explícita, 182-183  
     implícita, 182-183  
 Declarações de variáveis, 182-184  
 Declarações externas, 365-366  
**declare**, cláusulas, na Ada, 196  
 DEFINE, função Scheme, 563-567  
 DEFUN, função LISP, 575-577  
**delay**, 510-511  
**delete**, operador, 187-188  
 Delphi, 89-90  
 Derivação à extrema esquerda, 114-115  
 Derivações, públicas e privadas no C++, 460-462  
 Desalocação, 186  
 Descritores, 213  
 Deslocamento local, 383  
 Desvio incondicional, 317-416  
 Dijkstra, Edsger, 78, 320, 521  
 Discriminante, 243-244  
**display**, função Scheme, 563, 565-566  
 Displays, 392-398  
     ao implementar nomes de subprogramas, 403-404  
     ao implementar referências não-locais, 392-393  
     deslocamento (offset), 392-393

manutenção de, 392-398

**DISPOSE**, 253-254

Distinção entre maiúsculas e minúsculas, 176-178

**DISTRIBUTE** em HPF, 517-518

**DO**, instrução, 307-309

Domínios de programação, 43-45

*Dynabook*, 96

**E**

**Edwards**, Daniel J., 558

**EE**, classes de gramática, 182-185

Efeitos colaterais em expressões, 288

Efeitos colaterais funcionais, 274-276, 363-365

desaprovação de, 275

**Eiffel**

atributos na, 471-473

avaliação do suporte para OOP na, 472-475

características gerais da, 471-473

herança na, 471-475

origens da, 101-102

recursos na, 471-473

rotinas na, 471-473

vinculação dinâmica na, 472-475

**else**, cláusula, 297-299

**Else-if**, 304-307

Encadeamento dinâmico, 383

Encadeamento progressivo, 597

Encadeamento retrógrado, 597

Encadeamentos estáticos, 387

manutenção de, 390-393

Encapsulamento, 411-412

em classes SIMULA 74, 416-417

em tipos de dados abstratos Ada, 417

em tipos de dados abstratos C++, 420-422

em tipos de dados abstratos Modula-18, 420-421

Enlace mortal, 490-491

Entrada/saída, dependente da implementação, 68

**entry**, cláusulas, na Ada, 502

Enumeração definida pelo usuário, 221-222

Enumeração, literais de, 221

**EQ?**, função Scheme, 565

**equal**, função Scheme, 568-570

**EQUIVALENCE**, função Scheme, 563, 565-566

**EQV?**, instrução, 190

Erros de tipo, 189

Escalador, 489-490

Escopo, 193-194

avaliação do, 196-200

implementação do, 385-403

operador, 195-196

tempo de vida e, 200-201

Escopo de pacote, 423-424

Escopo dinâmico, 199

avaliação do, 199-200

implementação do, 398-403

no COMMON LIST, 573-574

Escopo estático, 193-200

avaliação do, 196-199

implementação do, 385-398

Estruturas de controle, 295

Estruturas de coleta de instruções, 295

Estruturas de dados, legibilidade, 26

**EVAL**, função Scheme, 375

**EVEN?**, função Scheme, 563, 565-566

Exceção desativada, 526-527

Exceção gerada, 523

Exceções em Java, 544-545

**exception**, cláusula, na Ada, 532

Expressão. Ver também tipos específicos

avaliação de, 269-276

coerção na, 279-280

condicional, 273-274

efeitos colaterais em, 274-276

erros em, 280-281

modo misto, 279

na Smalltalk, 441-444

relacionais, 280-282

Expressão matemática condicional, 563-567

Expressão S, 558

Expressividade, capacidade de escrita (*writability*) e, 29-30

Expressões aritméticas, 269

ordem de avaliação de operadores em, 269-271

ordem de avaliação de operandos em, 270-274

Expressões booleanas, 281-282

avaliação curto-círculo de, 283-284

avaliação de, 281-282

em Smalltalk, 447-449

Expressões de mensagem unárias, 442-443

Expressões lambda, 553-554

**extern**, 365-366

**F**

Falha de meta no Prolog, 598

Farber, D.J., 79-80

Fatias, 230-233

Fatoração à esquerda, 183-184

Fechaduras e chaves, 259-260

Fichas, 111

*Fifth Generation Computing Systems* (FGCS), 593

Fila pronta de tarefas, 489-490

Fim de arquivo, detecção de, 522

Finalização de tarefas na Ada, 507-508

**finally**, cláusula, no Java, 545-548

Fisher, David, 91-92

**flex**, matriz, no ALGOL 68, 82-83

Flex, linguagem, 97

FLOW-MATIC, 69

FLPL, 59

Fluxo de controle em matemática, 563-567

**for**, instrução

na Ada, 310-312

no C, C++ e Java, 308-314

no Pascal, 310-311

no ALGOL 68, 308-311

**FORALL** no HPF, 517-519

**foreach**, instrução, 316-319

- Forma clausal, 588–590  
 Forma funcional de composição em matemática, 553–555  
     na Scheme, 570–572  
 Formas de rótulo, 318–320  
 Formas funcionais, 553–555  
     apply-all, 554–555  
     composição, 553–555  
     construção, 554–555  
     na Scheme, 570–572  
 Formas identificadores, 176–178  
 Formas sentenciais, 115  
**FORTRAN** 75, 56–57  
     avaliação curto-círcuito no, 283–284  
     efeitos colaterais funcionais no, 275–276  
     implementação de subprogramas no, 376–379  
     SAVE no, 337  
     vinculação de subprogramas no, 376–377  
**FORTRAN** 88, 56–57  
     fatias de arrays na, 232  
     operações com matrizes no, 230  
     ponteiros no, 256–257  
**FORTRAN** I, 54–55  
**FORTRAN** II, 55–56  
**FORTRAN** IV, 56–57  
     instrução seletora no, 295–297  
**FORTRAN List Processing Language (FLPL)**, 59  
**FORTRAN**. Ver também *versões FORTRAN específicas, inclusive o HPF*  
     acessando ambientes não-locais no, 365–366  
     associatividade de operadores do, 272  
     atribuição de modo misto no, 381–382  
     avaliação do, 57  
     Blocos COMMON no, 365–366  
     compilação independente do, 363–364  
     declarações de tipo implícitas no, 183–184  
     desenvolvimento do, 53–55  
     GOTO atribuída no, 301–302  
     GOTO computada no, 301–302  
     IF aritmético no, 295–297, 300–302  
     instrução DO no, 307–309  
     métodos de passagem de parâmetros do, 342–343  
     parâmetros que são matrizes multidimensionais, 349–350  
     precedência de operadores do, 270–271  
     primeiras versões do, 53–56  
     registros de ativação para, 377–378  
     seletores múltiplos no, 300–302  
     tipos união no, 243–244  
 Função factorial, 384–385  
 Funções, 334, 363–365  
     amigas, 461–462  
     questões de projeto para, 363–364  
     Scheme primitivas, 559–563, 565–566  
     tipos de retorno para, 363–365  
 Funções construtoras, 420–422  
 Funções destrutoras, 420–422  
 Funções matemáticas, 553–555  
     fluxo de controle em, 563–567  
     formas funcionais, 553–555  
     simples, 553–554  
 Funções membro no C++, 420–421  
 Funções de predicado na Scheme, 565  
 Funções virtuais puras no C++, 464–465  
 Functor, 587, 593–594
- G**
- GAMM, 63  
 Gerenciamento do heap, 259–263  
 Glennie, Alick E., 53  
 Goldberg, Adele, 431  
 Gosling, James, 103, 211  
 GOTO atribuída, 301–302  
 GOTO composta, 301–302  
 Goto, instruções, 317–320  
     eliminação de, 318–319  
     ponteiros e, 257–259  
     problemas com, 317–319  
 GPSS, linguagem, 21–22  
 Gráficos ao estilo LOGO na Smalltalk, 453–456  
 Gráficos de fluxo, 320–321  
 Grafo dirigido, 124–125  
 Grafos de sintaxe, 124–125  
 Gramática de atributos, 126–131  
     avaliação de, 129–131  
     computação de valores na, 129–131  
     definição de, 126–128  
     exemplos de, 127–131  
     funções de computação de atributos da, 126–127  
     funções predicadas de, 126–127  
 Gramáticas  
     ambiguas, 117–118  
     analisadores cima-baixo, 184–193  
     classes de, EE, 182–185  
     definição de, 114–115  
     derivações e, 115  
     forma sentencial de, 115  
     independentes do contexto, 112–113  
     não-ambiguas, para expressões, 119–120  
     origens de, 112–113  
     para instruções de atribuição simples, 116–117  
     produções, 113–114  
     reconhecedores e, 125  
     recursão à esquerda, 122  
     regras, 113–114  
 Griswold, R.E., 80–81  
 GUIDE, 76
- H**
- Hápides, 259–260  
 Hash, 218–239  
 Haskell, 63, 576–581  
     abrangências de lista na, 576–579  
     avaliação preguiçosa na, 576–578, 578–581  
 Herança, 433–435  
     diamante, 439

- múltipla, 435, [430–440](#)  
 na Ada 100, 469–470  
 no C++, 460–470  
 na Eiffel, 471–475  
 na Smalltalk, [458](#)  
 simples, 435, [439](#)
- Herança de interface, 437
- High-Order Language Working Group (HOLWG)*, [91](#)
- High-Performance FORTRAN (HPF), 516–519
- Hoare, C.A.R., 301–302
- Hopper, Grace, [69–70](#), [109](#)
- HTML, [36](#)
- I**
- IAL, [64](#)
- IBM, computadores  
 IBM 701, [52–53](#)  
 IBM 704, [53](#), [54](#), 557–558  
 IBM 1401, 76  
 IBM 1620, 76  
 IBM 7080, 76  
 IBM 7090, 76  
 IBM System/360, 76  
 Ichbiah, Jean, 92
- IBM, linguagem assembly, [24–25](#)
- IF, aritmético, [269](#)
- IF, função Scheme, 563–568
- IF, instrução aninhamento de, 296–299
- Implementação  
 de blocos, 397–399  
 de construções orientadas a objeto, 474–479  
 de escopo dinâmico, 398–403  
 de escopo estático, 385–387  
 de matrizes, [232–237](#)  
 de métodos de passagem de parâmetros, 344–348  
 de parâmetros que são nomes de subprogramas, 402–405  
 de referências não-locais, 385–403  
 de registros, [242–243](#)  
 de subprogramas recursivos, 384–385  
 de tipos string de caracteres, [220–221](#)  
 de uniões, [246–249](#)  
 entendimento da, [17–18](#)  
 métodos de, [37–45](#)
- Implementação híbrida, [43–45](#)
- in out, modo, 343–344
- In, modo, 337–338
- in, modo, 343–344
- índices e matrizes, [225–227](#)
- Inferência de tipo, [184–185](#)
- Inferência lógica, 589–590
- Influências no projeto da linguagem, [33–36](#)  
 arquitetura de computador e, [33–35](#)  
 metodologia de programação e, [34–36](#)
- Inicialização de matrizes, [229–230](#)
- Inicialização de variáveis, [204–205](#)
- init, 496–497
- Inout, modo, 337–338
- Instanciação na comparação de proposições, 590
- Instrução de atribuição, [134–136](#), [283–289](#)  
 alvos condicionais, [285](#)  
 alvos múltiplos, [285](#)  
 como um operador, [287–288](#)  
 expressões condicionais, [273–274](#)  
 modo misto, [288–289](#)  
 na Smalltalk, 445–446  
 operador unário, [286–287](#)  
 operadores de atribuição compostos, [286](#)  
 simples, [285](#)
- Instrução de desvio incondicional, [317–320](#)  
 problemas com, [317–319](#)  
 restrições ao, [319–320](#)
- Instruções compostas, 295, 296–298
- Instruções de controle, [294](#)  
 desvio incondicional, [317–320](#)  
 iterativas, [305–307](#)  
 iteração baseada em estruturas de dados, [316–319](#)  
 legibilidade e, [25–26](#)  
 laços controlados por contador, [306–314](#)  
 mecanismo de controle de laço localizado pelo usuário, [314–318](#)  
 nas primeiras versões do FORTRAN, [294](#)  
 seleção múltipla, [299–307](#)
- Instruções relativas à meta no Prolog, 595–597
- Instruções relativas à regra no Prolog, 595–596
- Instruções de seleção, 295–296  
 aninhamento, 296–300  
 bidirecionais, 295–298  
 fechamento, [298–300](#)  
 múltiplas, 299–307  
 palavras especiais e, [298–300](#)  
 questões de projeto para, 295–297  
 Smalltalk, 449–450  
 unidirecionais, 295–296
- Instruções exit, [314–317](#)
- Instruções relativas a fatos no Prolog, 593–595
- Instruções iterativas, [305–316](#)  
 controladas logicamente, [312–316](#)  
 controladas por contador, [306–314](#)  
 corpo das, [306–308](#)  
 questões de projeto para, [305–307](#)
- Inteligência artificial, [20](#), [58–59](#), 580–581  
 primórdios da, [58–59](#)
- Interface de mensagem, 434
- International Algorithmic Language (IAL). Ver ALGOL 68.
- Interpretação  
 pura, [42](#)  
 velocidade de, [42](#)
- Interpretadores LISP, 557–558
- Invariante de laço, [138–141](#)
- IPL, linguagens, [59](#)
- Ironman Ada, documento de requisitos, [91–92](#)
- is, operador, no Prolog, 599
- Is-a, relação, 437
- Iteração na Smalltalk, [447–449](#)
- Iteradores, [317–319](#)
- Iverson, Ken, [79–80](#)

**J**

## Java

avaliação da manipulação de exceções no, 546–548  
 avaliação do, 104–105, 468–469  
 características gerais do, 466–467  
 continuação no, 541–544  
 encapsulamento no, 467–469  
 escopo de pacote no, 423–424, 467–468, 468–469  
 exceções não-verificadas no, 544–545  
 exceções verificadas no, 544–545  
**finally**, cláusula no, 545–548  
**for**, instrução no, 311–314  
 herança no, 466–468  
 laços lógicos no, 313–315  
 manipulação de exceções no, 541–548  
 métodos de passagem de parâmetros no, 343–344  
 parâmetros que são matrizes multidimensionais no, 349–350  
 processo de projeto do, 101–104  
 referências no, 256–258  
 retentor compartilhado acessado concorrentemente no, 514–515  
 sincronização de competição no, 512–514  
 sincronização de cooperação no, 513–514  
 threads no, 511–517  
 tipos de dados abstratos no, 422–426  
 vinculação dinâmica no, 467–468  
 vinculando exceções a manipuladores no, 541–543  
 visão geral do, 103–104

Johanniac, computador, 59  
 JOVIAL, 65

**K**

Kay, Alan, 96  
 Kemeny, John, 73, 375  
 Kernighan, Brian, 20–21  
 Knuth, Donald, 318–319  
 Korn, David, 20–21  
 Kowalski, Robert, 90, 585, 593  
 ksh, linguagem, 20–21  
 Kurts, Thomas, 73

**L**

Laços controlados logicamente, 313–315  
 questões de projeto para, 314  
 Laços controlados por contador, 306–314  
 instrução **DO** do FORTRAN 77 e 90, 308–309  
 instrução **for** Ada, 311–312  
 instrução **for** do ALGOL 60, 309–311  
 instrução **for** do C, C++ e Java, 312–313  
 instrução **for** Pascal, 311  
 questões de projeto de, 306–308  
 Laços, corpo de, 306–308  
 Laços lógicos pós-teste, 313–315  
 Laços lógicos pré-teste, 313–315  
 Lado direito, 108

LAMBDA, função Scheme, 563, 565–564  
 Laning e Zierler, sistema algébrico, 53  
 Legibilidade, 22–28  
 considerações sobre sintaxe na, 27–28  
 formas identificadoras e, 27  
 instruções de controle e, 26  
 ortogonalidade e, 24–26  
 palavras especiais e, 27  
 simplicidade e, 23–24  
 tipos de dados e estruturas, 26–27

LET, função Scheme, 570–571

Lexemas, 111

Ligação de subprograma, 376

Limitações intrínsecas do Prolog, 610

Linguagem B, 86

Linguagem C

acessando ambientes de referenciamento em, 365–366  
 associatividade de operadores da, 272  
 avaliação da, 87  
 coerção (cast) na, 279–280  
 declarações externas na, 365–366  
 embasamento histórico, 86–87  
 expressões condicionais na, 273–274  
 laço controlado por contador na, 311–314, 316–318  
 laços controlados logicamente na, 313–315  
 mecanismo de controle de laços localizado pelo usuário na, 315–318  
 métodos de passagem de parâmetros da, 343–344  
 operadores de atribuição compostos na, 286–287  
 operadores de decremento na, 286–287  
 operadores de incremento na, 286–287  
 parâmetros que são matrizes multidimensionais, 348–349  
 ponteiros na, 254–257  
 precedência de operadores da, 270–271  
 seletores múltiplos na, 303–306  
 tipos numéricos na, 213–214  
**typedef**, 202  
 uniões na, 243–244  
 verificação de tipos de parâmetro na, 343–345  
**void \***, ponteiros na, 256–257

Linguagem C++

associatividade de operadores, 272  
 avaliação da, 101  
 avaliação da manipulação de exceções na, 540–541  
 avaliação do suporte para programação orientada a objeto, 465–466  
 classes abstratas na, 464–465  
 comparada com a Smalltalk, 464–465  
 continuação na, 538–539  
**friend**, 461–462  
 função construtora na, 420–422  
 função destrutora na, 420–422  
 funções genéricas na, 359–361  
 funções modelo na, 359–361  
 funções virtuais puras na, 464–465  
 herança na, 460–464  
 manipulação de exceções, 537–541  
 membros de dados na, 420–421  
 métodos de passagem de parâmetro na, 343–344  
 operador de resolução de escopo na, 461

- parâmetros padrão na, 333–334  
 ponteiros na, [254–257](#)  
 processo de projeto da, 99–100  
 subprogramas sobrecarregados na, [357–358](#)  
 tipo de dados abstratos na, [420–424](#)  
 tipos de dados abstratos parametrizados, 425–427  
 tipos referência na, [256–258](#)  
 uniões na, [243–244](#)  
 verificação de tipo de parâmetro, 343–345  
 vinculação dinâmica na, 463–465  
 vinculando exceções a manipuladores na, 538–539  
 visão geral da, 100–101  
**void \***, ponteiros na, [256–257](#)  
**while** na, [312–314](#)
- Linguagem estruturada em blocos, [196](#)  
 Linguagem intermediária, [38–39](#)  
 Linguagem. Ver também Linguagens de programação;  
     linguagens específicas  
     critérios de avaliação, [21–33](#)  
     custo/benefício de projeto, [36–37](#)  
     geradores de, 111–113  
     reconhecedores, 111–112  
 Linguagem-fonte, [38–39](#)  
 Linguagens, gramáticas e expressões regulares, [175](#)  
 Linguagens de programação de sistemas, [20–21](#)  
     aplicações das, 579–581  
     comparação com linguagens imperativas, 580–582  
     fundamentos das, [554–556](#)  
 Linguagens de programação orientadas a objeto  
     exclusividade de objetos nas, 436–437  
     herança de implementação nas, 437  
     herança de interface nas, 437  
     herança múltipla nas, [439–440](#)  
     herança simples nas, [439](#)  
     polimorfismo nas, 438–439  
     questões de projeto para, 436–440  
     subtipos nas, 437  
     verificação de tipos nas, 438–439  
 Linguagens de programação. Ver também Linguagens  
     imperativas; Linguagens de programação  
     orientadas a objeto; linguagens específicas  
     baseadas em objetos, [433](#)  
     baseadas em procedimentos e não baseadas em  
         procedimentos, 592  
     benefícios do estudo das, [16–19](#)  
     critérios de avaliação das, [21–23](#)  
     funcionais, Capítulo 14  
     influências sobre o projeto de, [33–36](#)  
     lógicas, Capítulo 15  
     métodos de implementação de, [37–45](#)  
     não-imperativas, 592  
     orientadas a objeto, Capítulo 15  
 Linguagens de scripting, [21–22](#)  
 Linguagens imperativas, [34](#)  
 Linguagens para fins especiais, [21–22](#)  
*Linkeditor*, [41](#), 378  
 LISP, [20–21](#), [556](#)  
     avaliação do, [61–62](#)  
     estruturas no Prolog, 602–606  
     notação, [59–60](#)  
     primeiro desenvolvimento do, [59–60](#)
- primeiro interpretador do, 557–560  
 processamento do, primórdios do, [58–59](#)  
 processamento na Scheme, 561–562, 564  
 representação de lista interna para, [60–61](#)  
 sintaxe do, [60–62](#)  
 tipos e estruturas de dados, [59–60](#), [556–558](#)
- LIST**, função Scheme, 562, 564  
**LIST?**, função Scheme, 565  
 Literais  
     na Smalltalk, [441](#)  
     sobrerecarregadas, [222](#)  
 Lixo, [253–254](#)  
     coleta de, [260](#)  
 Lógica simbólica, 587  
**LOPHOLE**, no [Modula-13](#), 190  
 Lovelace, Augusta Ada, 92
- ## IV
- Macroinstruções, [37](#)  
**MAD**, linguagem, [65](#)  
 Manipulação de exceções, Capítulo 13  
     conceitos básicos da, [523–524](#)  
     em Java, [63–548](#)  
     em PL/I, 527–532  
     história da, 527–528  
     na Ada, 532–537  
     no C++, 537–541  
     questões de projeto da, [524–528](#)  
 Manipuladores (*handlers*) de exceção, [523](#)  
     continuação de, 525–526  
     definidos pelo usuário, [524–525](#)  
     geração de, [523](#)  
     vinculando exceções a, [524–526](#)  
**MAPCAR**, função Scheme, 571–572  
 Markup, linguagens, [36](#)  
 Matrizes, [225](#)  
     avaliação de, [232–233](#)  
     conformantes, [228](#)  
     dinâmicas na pilha, [226–228](#)  
     estáticas, [227](#)  
     fatias de, [230–233](#)  
     fixa-dinâmicas na pilha, [226–227](#)  
     fixa-dinâmicas no monte, [226–227](#)  
     funções de acesso para, [235–237](#)  
     implementação de, [232–237](#)  
     índices e, [225–226](#)  
     initialização de, [229–230](#)  
     na Plankalkül, [49–50](#)  
     número de subscritos em, [229](#)  
     operações de, [230–231](#)  
     questões de projeto de, [225](#)  
     sintaxe de referências, [226–227](#)  
     vinculações de subscritos e, [226–228](#)
- Matriz associativas  
     estrutura e operações de, [237–239](#)  
     implementação de, [238](#)
- Matriz multidimensionais, [234–237](#)  
     como parâmetros, 348–350
- Mauchly, John, [51–52](#)

McCarthy, John, 59-60, 551, 557-558  
 McCracken, Daniel, 37  
 Mecanismos de controle de laços definidos pelo usuário, 314-318  
 questões de projeto, 314-316  
**member**, função Scheme, 567-569  
 Membros de dados, 420-421  
 Memória  
 células, 180-182  
 conexão com o processador, 41  
 endereço, 178-180  
**Menos unário**, 269-270  
 Mensagens em linguagens orientadas a objeto, 434  
**Metalinguagem**, 113-114  
 Metodologias de programação, 34-36  
 Método sobreposto, 434  
 Método virtual, 435-436  
 Métodos, 434  
 na Smalltalk, 444-445, 449-451  
 Métodos de instância e variáveis, 435  
 Meyer, Bertrand, 101-102  
**Microinstruções**, 37  
**Micro-Prolog**, 593  
 Milner, Robin, 62  
 Minsky, Marvin, 59-60  
 Miranda, 63, 576-578  
 ML, 62, 575-578  
 inferência de tipo na, 184-185, 576-577  
 Modelo de rastreamento para o Prolog, 600  
 Modula, 88  
 Modula-2  
 avaliação curto-círculo no, 283-284  
 conjuntos no, 247-249  
 embasamento histórico do, 88-89  
 instrução de seleção no, 299-300  
 tipos de dados abstratos no, 420-421  
 Modula-3, 89, 190  
 Modularização, 411  
 Módulos, 411  
 no Modula-2, 420-421  
 Monitores  
 avaliação de, 499-500  
 e retentor compartilhado acessado concorrentemente, 498-499  
 no Concurrent Pascal, 494-496  
 simulação de, na Ada, 508-509  
 sincronização de competição com, 496-497  
 sincronização de cooperação com, 496-498  
 Mouse, dispositivos de apontamento, 459  
 Multiple-Instruction Multiple-Data (MIMD), 486

**N**

Naur, Peter, 66, 113-114, 293  
**NELIAC**, 65  
 Nesting\_depth, 388  
*New Programming Language*. Ver NPL, Linguagem.  
**new**, operador, 187-188  
 Newell, Allen, 59  
**NEWLINE**, função Scheme, 563, 565-566

Nomes  
 forma de, 176-178  
 palavras especiais, 177-179  
 predefinidos, 177-178  
 questões de projeto de, 176-177  
 Nomes de subprograma que são parâmetros, 355-357  
 implementação de, 402-405  
**Norwegian Computing Center**, 80-81  
**NPL**, linguagem, 22  
**NULL?**, função Scheme, 565  
 Números de nível, 238-240  
 Números reais, 213-214  
 Nygaard, Kristen, 80-81

**O**

**Oberon**, 89  
**Objeto**, 434  
 Objetos protegidos na Ada 95, 509-511  
 Ocultação de informação  
 em tipos de dados abstratos Ada, 417-418  
 em tipos de dados abstratos C++, 420-422  
 em tipos de dados abstratos SIMULA 67, 416-417  
**ON**, na PL/I, 527-528  
 Operações com matrizes elementares, 230  
 Operador de identidade, 269-270  
 Operadores  
 associatividade de, 120-122  
 atribuição, 285  
 booleanos, 281-282  
 precedência de, 118-120, 269-271  
 relacionais, 280-282  
 sobrecarregados, 275-278  
 sobrecarregados definidos pelo usuário, 367  
 unários, 269-270, 271-272  
 Operadores binários, 269  
 Operadores ternários, 269  
 Operandos, ordem de avaliação, 274-276  
 Ordem de coluna maior, 234  
 Ordem de linha maior, 234  
 Ortonalidade  
 capacidade de escrita (writability) e, 28-29  
 legibilidade e, 24-26  
 no ALGOL 68, 25-26  
**others**, cláusula, 304-306  
 Ottimização, 38-39  
 Oustershout, John, 20-21  
 Out, modo, 337-338  
 out, modo, 343-344

**P**

P, operador, 491-492  
**packed**, 215-217  
 Pacotes genéricos na Ada, 424-427  
 Pacotes  
 corpo, 417  
 especificação, 417  
 Pai-estático, 194-195

- Palavra-chave, [177-178](#)  
 expressões na Smalltalk, [443](#)  
 parâmetros, [333](#)
- Palavras especiais  
 fechamento de seleção e, [298-300](#)
- Palavras reservadas, [177-178](#)
- Panini, [113-114](#)
- Papert, Seymour, [96](#)
- Parâmetros  
 correspondência de, [333](#)  
 formais, [332-333](#)  
 matrizes multidimensionais, [348-350](#)  
 nome de subprograma, [355-357](#)  
 palavra-chave, [332](#)  
 posicionais, [333](#)  
 reais, [333](#)  
 verificação de tipo, [343-346](#)
- Parâmetros, métodos de passagem, [337-354](#)  
 considerações sobre o projeto de, [351](#)  
 das principais linguagens, [343-344](#)  
 exemplos, [351-355](#)  
 implementação de, [345-348](#)  
 modelos de implementação de, [338-343](#)  
 modelos semânticos, [337-338](#)  
 por nome, [341-343](#)  
 por resultado, [339](#)  
 por referência, [340-341](#)  
 por valor, [338-339](#)  
 por valor-resultado, [340](#)
- Parênteses em expressões, [273-274](#)
- Pascal. Ver também Concurrent Pascal  
 associatividade de operadores do, [272](#)  
 atribuição de modo misto no, [288-289](#)  
 avaliação curto-circuito no, [283](#)  
 avaliação do, [84-86](#)  
 embasamento histórico do, [83-84](#)  
**case**, instrução, no, [302-303](#)  
 conjuntos no, [247-249](#)  
 e tipificação forte, [190](#)  
 instrução de seleção no, [298-299](#)  
 laço controlado por contador no, [310-311](#)  
 laços controlados logicamente no, [314-316](#)  
 matrizes conformantes no, [228-229](#)  
 ponteiros no, [253-254](#)  
 precedência de operadores do, [270-271](#)  
 registros variantes no, [243-246](#)  
 restrições a gotos no, [319-320](#)  
 seletores múltiplos no, [302-303](#)  
 simplicidade de projeto do, [83-84](#)  
 tipos enumeração no, [222](#)  
 tipos união no, [243-246](#)
- Passagem de mensagens  
 avaliação da, [508-509](#)  
 conceito de, [500-501](#)  
 conceitos de projeto de, [501](#)  
 na Ada, [501-505](#)  
 retentor compartilhado acessado concorrentemente com, [507](#)  
 sincronização de competição com, [505-506](#)  
 sincronização de cooperação com, [505-506](#)
- PDP-11, computador, [287](#)
- Perfil de parâmetro, [332](#)
- Perl, linguagem  
 coincidência de padrões na, [217-219](#)  
 iterações baseadas em estruturas de dados da, [316-319](#)  
 matrizes associativas na, [237-239](#)  
 operações com string de caracteres na, [217-218](#)
- Perlis, Alan, [63](#)
- Pesquisa de diferenciação simbólica, [59](#)
- Pilha em tempo de execução, [381-382](#)
- Pilha oculta, [402-403](#)
- PL/C, [78](#)
- PL/CS, [78](#)
- PL/I  
 avaliação da manipulação de exceções na, [531-532](#)  
 continuação na, [528](#)  
 descrição da semântica da, [133-134](#)  
 embasamento histórico da, [76-77](#)  
 manipuladores de exceção na, [527-528](#)  
 problemas com ponteiros na, [252](#)  
 processo de projeto da, [78](#)  
 rótulos na, [319-320](#)  
 vinculando exceções a manipuladores na, [528](#)  
 visão geral da, [77-78](#)
- PL/S, [20-21](#)
- Plankalkül, [49-51, 65](#)
- Polensky, EP, [80-81](#)
- Polimorfismo, [435-436](#)
- Polimorfismo paramétrico, [357-359](#)
- Ponteiros  
 avaliação dos, [257-259](#)  
 implementação de, [257-260](#)  
 na Ada, [254](#)  
 no C e no C++, [254-257](#)  
 no Pascal, [253-254](#)  
 no FORTRAN [90, 256-257](#)  
 operações de, [250-252](#)  
 problemas de, [252-254](#)  
 questões de projeto dos, [250-251](#)  
 representações de, [257-259](#)  
 valoração, [251-252](#)
- Ponteiros pendurados, [252](#)  
 solução para, [257-260](#)
- Pós-condicionais, [133-135](#)
- Pós-condições, [133-135](#)
- pragma**, [535-536](#)
- Precedência, [269-271](#)
- Pré-condição, mais fraca, [134-135](#)
- Pressuposição de mundo fechado no Prolog, [608](#)
- Primitiva, função Scheme, [560-567](#)
- Prioridades de tarefas, [507-508](#)
- Problema de navegação no Prolog, [608-610](#)
- Problema de produtor-consumidor, [487-489](#)
- Procedimentos, [333-334](#)
- Processamento de linguagem natural, [580-581, 612](#)
- Produção, [113-114](#)
- Profundidade\_estrática, [388](#)
- PROG, função LISP, [574-575](#)
- Programação lógica  
 aplicações de, [611-613](#)  
 deficiências da, [606-610](#)  
 linguagens, [35-36, Capítulo 15](#)

- visão geral da, [591-593](#)  
 Programação orientada a dados, [34-35](#)  
 Programação orientada a objeto, [34-36](#), 435-436  
     herança em, [433-435](#)  
     polimorfismo e vinculação dinâmica na, 435-436  
 Programação orientada para o processo, [34-35](#)  
 Programas Ada errôneos, 347  
 Projeto ortogonal, 81-84  
**Prolog**  
     aplicações do, 611-613  
     aritmética no, 599  
     avaliação do, [91](#)  
     deficiências do, [606-610](#)  
     elementos básicos do, 593-606  
     estruturas de lista no, 602-606  
     instanciação de variáveis no, 593-594  
     instrução relativas a meta do, 595-597  
     instruções relativas a regra no, 594-596  
     instruções do, 593-597  
     instruções fact no, 593-595  
     limitações intrínsecas do, [610](#)  
     método gerar-e-testar no, [607](#)  
     modelo de rastreamento (*trace*) para o, 600  
     operador cut no, [606-607](#)  
     origens do, [90](#), 593  
     processo de inferência do, 596-598  
     processo de projeto do, [90](#)  
     termos no, 593-594  
     vantagens do, [612-613](#)  
     visão geral do, [90-91](#)  
**Prolog++**, [35-36](#)  
**Proposição de hipótese**, [591](#)  
**Proposição de meta**, [591](#), 596-597  
**Proposições**, 587  
     forma clausal, 588-590  
**Proposições atômicas**, 587  
**Proposições compostas**, 587-588  
**Proposições lógicas**, 587-590  
**Proteção**, 505  
**Protegidas**, 434  
**Protocolo**, 332  
**Protocolo de mensagem**, 434  
**Protótipo**, 332, 343-345  
**Prova de exatidão**, [140-143](#)  
**Pseudocódigos**, [50-53](#)  
**public**, 418
- Q**
- queue**, tipo, 497-498  
**QuickBASIC**, 75-76  
**QUOTE**, função Scheme, 560-561
- R**
- raise**, na Ada, 534-535  
**range**, [223](#)  
**Reconhecedores**, 111-112  
**Recursos desaprovados do FORTRAN**, [52](#)
- Recursos imperativos da Scheme, 572-574  
 Recursos na Eiffel, 471-473  
 Recursos obsoletos do FORTRAN, [52](#)  
 Referência a substring, [215-217](#)  
 Referência amplamente qualificada, [239-242](#)  
 Referência pendente. Ver Ponteiros pendurados  
 Referências  
     a campos de registro, [239-242](#)  
     oscilantes, [252](#)  
 Referências elípticas, [240-242](#)  
 Referências não-locais, 363-365  
     mechanismo para implementação de, 385-398  
**Registro de ativação**, [376-378](#)  
     em linguagens assemelhadas ao ALGOL, 380-382  
     instância, 377-378  
     no FORTRAN [77](#), 377-378  
**Registro variante**, [243-246](#)  
     não-restrito, [245-246](#)  
 Registros de instância de classe, 474-478  
**Regra da consequência**, [134-136](#)  
**Regras de inferência**, [134-136](#)  
**Regras gramaticais recursivas**, [122](#)  
**Regras na BNF**, [113-114](#)  
**release**, 491-493  
**Rendezvous**, representação gráfica de, 504  
**repeat-until**, [314-316](#)  
**Resolução**, 589-590  
 Retentores, acessado concorrentemente, compartilhado  
     com monitores, 498-499  
     com semáforos, 493-495  
     usando passagem de mensagens, 507  
**Retomada**, 368  
**Retornos, semântica de**, [376](#)  
**Richard, Martin**, [86](#)  
**Ritchie, Dennis**, [86](#), 329  
**Robinson, Alan**, 589-590  
**Rotinas na Eiffel**, 471-473  
**Roussel, Phillippe**, [90](#), 593  
**RPG, linguagem**, [21-22](#)  
**Russell, Stephen B.**, 558  
**Rutishauser, H.**, 68
- S**
- Sammet, Jean**, 106  
**Satisfazendo metas no Prolog**, 596-597  
**SAVE**, no FORTRAN, 337  
**Schai, A.**, [148-149](#)  
**Scheme**, [61-62](#), 559-574  
     fluxo de controle na, 563-568  
     formas funcionais na, 570-572  
     funções para definir funções na, 563, 565-567  
     funções primitivas da, 559-563, 565-566  
     funções que constroem programas Scheme em, 571-573  
     recursos imperativos da, 572-574  
**Schwartz, Jules I.**, [65](#)  
**select**, assíncrono na Ada 95, 510-511  
     cláusulas, na Ada, 504-505  
**Seletores bidirecionais**,  
     questões de projeto, 295-297

- Seletores de aninhamento, 296–299  
 Seletores tridirecionais, 300–301  
 Seletores unidirecionais, 295–296  
**self**, 450–451  
 Semáforo binário, 493–494, 507–509  
 Semáforos, 491–492  
     avaliação da, 493–496  
     retentor compartilhado acessado concorrentemente com, 493–495  
     sincronização de competição com, 493–495  
     sincronização de cooperação com, 491–494  
 Semântica declarativa, 591  
     dinâmica, 131–132  
     estática, 126–127  
 Semântica axiomática, 133–134  
     as pré-condições mais fracas na, 133–136  
     asseções na, 133–135  
     avaliação da, 142–144  
     axiomas na, 134–136  
     instruções de atribuição na, 134–137  
     laços de pré-teste lógico na, 137–142  
     laços invariantes para, 138–141  
     pós-condições na, 133–135  
     pré-condições na, 133–135  
     prova de exatidão usando, 140–143  
     regras de inferência na, 134–136  
     seleções na, 137–138  
     sequências na, 136–138  
 Semântica denotacional, 142–144  
     avaliação da, 146–148  
     estado do programa em, 144–145  
     expressões em, 145–146  
     instruções de atribuição na, 146–147  
     laços de pré-teste lógico em, 144–145  
 Semântica operacional, 131–132  
     avaliação da, 133–134  
     processo básico da, 131–134  
 Sentença, 111  
 SET!, função Scheme, 572–574  
 SETQ, função COMMON LISP, 574–575  
 SGBDR. Ver Sistemas de gerenciamento de banco de dados relacionais  
 SHARE, 63, 76, 22  
 Shaw, J.C., 59  
 Shell, 20–21  
 Shell, comandos, no UNIX, 28  
 Short Code, 51–52  
 SIGNAL, na PL/I, 529  
 Símbolo de início de uma gramática, 114–115  
 Símbolos (*tokens*), 105, 150–151  
 Símbolos não-terminais, 114–115  
 Símbolos terminais, 113–114  
 Simon, Herbert, 59  
 Simplicidade  
     capacidade de escrita (*writability*) e, 28–29  
     legibilidade e, 22–24  
 SIMULA 65  
     ocultação de informação no, 416–417  
     processo de projeto do, 80–81  
     tipo de dados abstratos no, 416–417  
     visão geral do, 80–82  
 Sincronização competitiva, 487–488  
     com monitores, 496–497  
     com passagem de mensagens, 505–506  
     com semáforos, 493–495  
 Sincronização de cooperação, 487–488  
     com monitores, 496–498  
     com passagem de mensagens, 505–506  
     com semáforos, 491–494  
     falhas, 487–488  
 Single-Instruction Multiple-Data (SIMMD), 485–486  
 Sintaxe descrição de, 111–112  
     métodos formais de descrição de, 112–125  
 Sistema de compilação, UNIVAC, 52–53  
 Sistema de gerenciamento de arquivos, 37  
 Sistema operacional, 37  
 Sistemas de gerenciamento de bancos de dados relacionais, 611  
 Sistemas especialistas, 611–612  
 Smalltalk ambiente, 440–441  
     avaliação da, 98, 458–459  
     blocos na, 446–448  
     características gerais da, 440–441  
     classes na, 449–450  
     comparada com o C++, 464–465  
     escolha de método na, 444–445, 449–451  
     expressões de mensagens na, 442–444  
     expressões em, 441–444  
     gráficos ao estilo LOGO na, 453–456  
     herança na, 458  
     impacto da, 98  
     instruções de atribuição na, 445–446  
     instruções de controle na, 447–450  
     interface com o usuário da, 440–441  
     iteração na, 447–449  
     literais na, 441  
     processo de projeto da, 96–97  
     seleção na, 449–450  
     variáveis na, 442  
     verificação de tipos e polimorfismo na, 456–458  
     vinculação dinâmica na, 456–458  
     visão geral da, 97–98, 440–441  
 SNOBOL4, 218  
     origens e características do, 79–81  
 Sobrecarga de operadores, 275–276  
 Software básico, 20–21  
 Speedcoding, 52–53  
 SQRT, função Scheme, 560  
 Strawman, documento, 91–92  
 Stroustrup, Bjarne, 99, 175  
 Subclasse, 434  
 Submeta, 596–597  
 Subprograma sensível à história, 185–186  
 Subprogramas  
     acessando ambientes não-locais em, 363–365  
     ambientes de referenciamento locais em, 336–337  
     ativos, 202–203, 331, 381–382  
     características dos, 330–331  
     chamada, 331

Hidden page

# CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO

5<sup>a</sup> Edição

Robert W. Sebesta

Robert Sebesta oferece nesta obra as ferramentas necessárias para uma avaliação crítica das linguagens de programação existentes e futuras. Conceitos fundamentais, itens de projetos de várias construções e exame de escolhas em algumas das linguagens mais usadas, comparando-as com as possíveis alternativas, fazem parte do texto.

Este é um livro ideal para programadores, pois mostra como escolher a linguagem adequada para determinadas tarefas, aumenta a habilidade de aprender novas linguagens e de entender o significado de implementação.

Destaques desta edição:

- Mostra várias construções de linguagem e alternativas de projeto com Java, JavaScript, C++, C, Perl, Ada e Fortran.
- Intercala discussões sobre programação orientada a objeto com material de linguagens imperativas não-orientadas a objeto.
- Oferece o contexto histórico das necessidades específicas que determinaram as escolhas de projeto em linguagens existentes.

ISBN 85-363-0171-6



9 788536 301716

**artmed®**  
EDITORAS  
RESPEITO PELO CONHECIMENTO



www.bookman.com.br

Copyrighted material