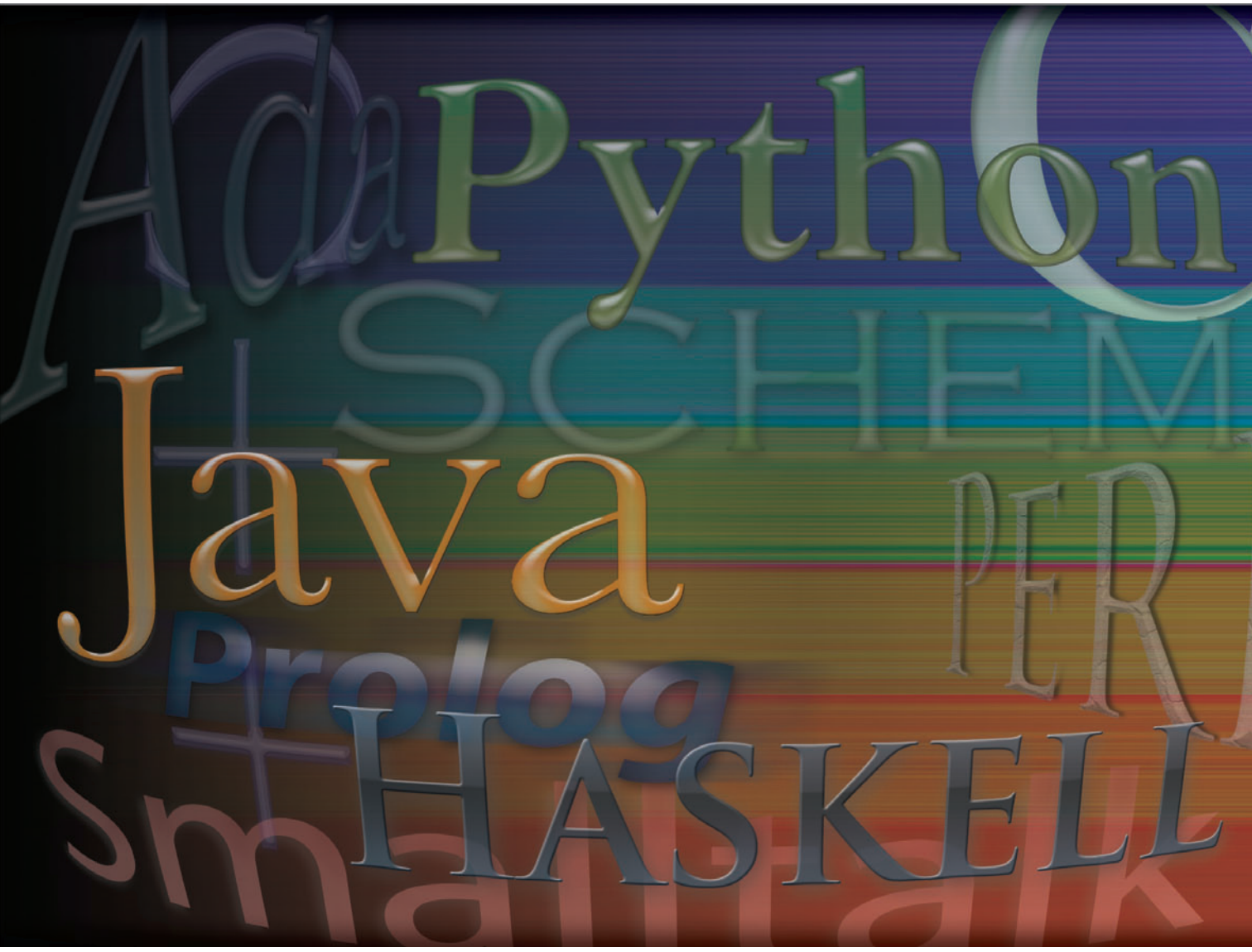
Linguagens de Programação

Segunda Edição

Princípios e Paradigmas



Allen B. Tucker

Robert E. Noonan

Mc Graw Hill

Linguagens de Programação

Princípios e Paradigmas

Allen B. Tucker
Bowdoin College

Robert E. Noonan
College of William and Mary

Tradução
Mario Moro Fecchio
Acauan Fernandes

Revisão Técnica
Eduardo Marques
Doutor em Engenharia de Sistemas Digitais pela USP
Mestre em Ciência da Computação pela USP
Docente do Instituto de Ciências Matemáticas e de Computação – ICMC – USP

Márcio Merino Fernandes
PhD em Ciência da Computação pela University Of Edinburgh – Escócia
Professor Adjunto da Universidade Federal de São Carlos – UFSCAR

Versão impressa
desta obra: 2008

AMGH Editora Ltda.
2010

The McGraw-Hill Companies

Linguagens de programação – princípios e paradigmas
Segunda edição
ISBN 978-85-7726-044-7

Tradução da segunda edição em inglês de Programming languages – principles and paradigms
© 2007 by The McGraw-Hill Companies, Inc.
ISBN da obra original: 978-0-07-286609-4

Coordenadora editorial: Guacira Simonelli
Editora de desenvolvimento: Alessandra Borges
Produção editorial: Nilceia Esposito ERJ Composição Editorial
Supervisora de pré-impressão: Natália Toshiyuki
Preparação de texto: Marta Almeida de Sá
Diagramação: ERJ Composição Editorial
Design de capa: Rokusek Design
Imagem de capa (USE): Rokusek Design

T89l Tucker, Allen B.
Linguagens de programação [recurso eletrônico] : princípios e paradigmas /a Allen B. Tucker, Robert E. Noonan ; tradução: Mario Moro Fecchio ; revisão técnica: Eduardo Marques, Márcio Merino Fernandes. – Dados eletrônicos. – Porto Alegre : AMGH, 2010.

Editado também como livro impresso em 2008.
ISBN 978-85-63308-56-6

1. Ciência da computação. 2. Linguagem de programação. I. Noonan, Robert E. II. Título.

CDU 004.43

Catalogação na publicação: Ana Paula M. Magnus – CRB-10/Prov-009/10

Para Maida.

Allen B. Tucker

Para Debbie e Paul.

Robert E. Noonan

Prefácio

O estudo de linguagens de programação evoluiu rapidamente desde que começamos o desenvolvimento da primeira edição* deste livro em 1999. Por exemplo, Java se tornou a linguagem dominante no currículo de ciência da computação, começando com CS1. A programação ágil emergiu como uma abordagem coerente para o projeto de software e suas preferências de linguagens são diferentes daquelas da programação tradicional. O uso de métodos formais em projeto de software começou a ser de uso comum e sua importância agora é significativa.

Em resposta a estes e a outros acontecimentos, esta obra espera capturar o interesse e os novos desafios que acompanham o projeto de linguagens de programação atuais e do futuro. Por exemplo, esta edição possui uma cobertura maior e mais profunda de todos os quatro paradigmas de programação e as linguagens que os acompanham.

	Cobertura da Linguagem	
Paradigma	**Primeira Edição**	**Esta Edição**
Imperativo (Capítulo 12)		C Ada Perl
Orientado a Objetos (Capítulo 13)	Java	Java Smalltalk Python
Funcional (Capítulo 14)	Scheme Haskell	Sheme Haskell
Lógico (Capítulo 15)	Prolog	Prolog

A segunda maior mudança nesta edição é que a discussão de princípios de projeto de linguagens nos primeiros capítulos (2-11) foi bastante ampliada. Acrescentamos novos exemplos de linguagens atuais (como Python e Perl), usando um estilo informal de apresentação. Além disso, eliminamos a maior parte da cobertura de linguagens antigas que não são mais tão usadas (como Pascal e Modula).

Os princípios básicos das linguagens de programação – sintaxe, nomes, tipos, semântica e funções – são os assuntos dos Capítulos 2, 4, 5, 7 e 9, respectivamente. Esses capítulos fornecem um estudo prático desses princípios usando uma seleção aprofundada de linguagens e exemplos.

Os leitores que preferirem um tratamento de sintaxe, sistemas de tipo, semântica, funções e gerenciamento de memória baseados em implementações encontrarão este material nos Capítulos complementares 3, 6, 8, 10 e 11. Esses capítulos podem ser usados seletivamente para enriquecer os princípios básicos com os quais eles estão relacionados.

* NE: edição em inglês.

Por exemplo, o estudo de sintaxe no Capítulo 2 pode ser enriquecido pelo estudo das fases léxica e sintática de um compilador no Capítulo 3. *Enfatizamos que qualquer um destes capítulos complementares pode ser preterido em relação aos demais, especialmente no início de um curso sobre linguagens de programação.*

π Três capítulos complementares incluem seções opcionais com tratamento matemático mais formal. Essas seções estão marcadas com o mesmo símbolo à margem como o deste parágrafo para indicar que são opcionais. O Apêndice B fornece uma revisão de tópicos de matemática discreta e notações correspondentes a estas seções para alunos que precisarem de uma revisão rápida.

Finalmente, os capítulos de tópicos especiais (16, 17 e 18) fornecem introduções detalhadas ao estudo de manipulação de eventos, concorrência e corretude de programas.

De modo geral, esta edição contém uma cobertura ampla e profunda dos princípios, paradigmas e tópicos especiais de linguagens de programação. O livro possui 18 capítulos. Já que cada professor possui diferentes visões sobre o que deve ser enfatizado em um curso de linguagem de programação, este livro fornece uma diversidade de opções.

ÊNFASE

Este texto enfatiza um tratamento prático e completo das questões-chave do projeto de linguagem de programação. Ele fornece a professores e alunos uma mistura de experiências baseadas em explicações e em implementações. As experiências baseadas em implementações incluem experimentações práticas com o projeto e implementações de um sub-conjunto modesto de C, chamado *Clite*, que está integralmente definido no Apêndice A para facilitar a referência.

Conforme citado anteriormente, esta edição traz um tratamento abrangente dos principais paradigmas de programação. Acreditamos que, para dominar um paradigma, os alunos devem usá-lo ativamente para resolver um problema de programação. Se, por exemplo, seus alunos não tiverem experiência com programação funcional, recomendamos que eles aprendam Scheme ou Heskell de forma que consigam completar um projeto de programação razoável. Parafraseando o comentário de um revisor:

> Para se entender um paradigma você deve se tornar esse paradigma.

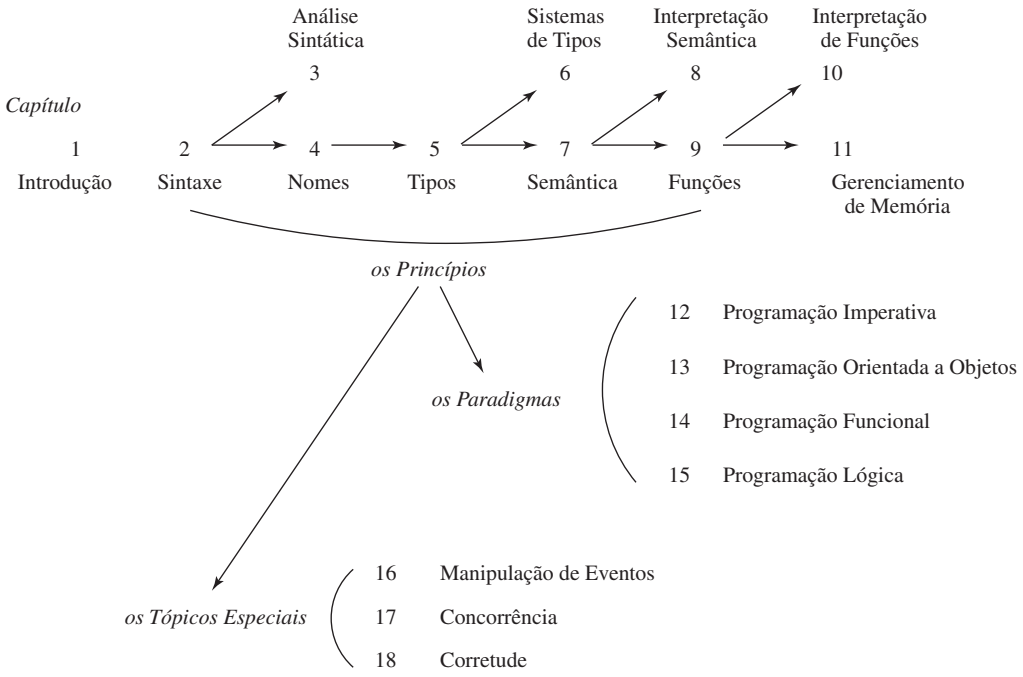
Se, por outro lado, o curso introdutório de matemática discreta ou IA já incluir a programação funcional, é possível pular este capítulo e enfatizar outro paradigma ou tópico especial.

ORGANIZAÇÃO DO CURSO

A Figura 1 mostra como o texto se divide em três seções principais:

- Princípios
- Paradigmas
- Tópicos especiais

Na primeira seção, os Capítulos 2, 4, 5, 7 e 9 cobrem cinco *princípios básicos* – sintaxe, nomes, tipos, semântica e funções. Partes dos capítulos restantes (3, 6, 8, 10 e 11) desta seção podem ser utilizados para abordar o tema de forma mais completa. Por outro



| Figura 1 Dependências do conteúdo entre os capítulos.

lado, cursos avançados ou de pós-graduação podem incluir diversos destes tópicos, em vez de enfatizar os capítulos posteriores.

A segunda seção do texto cobre os quatro principais paradigmas de programação:

- Programação imperativa
- Programação orientada a objetos
- Programação funcional
- Programação lógica

Estes capítulos são relativamente independentes um do outro e podem ser estudados em qualquer ordem. A escolha de paradigmas, é claro, pode variar de acordo com as preferências do professor e o currículo do curso.

O Capítulo 12 demonstra as características-chave da programação imperativa entre três diferentes linguagens: C, Ada e Perl. C foi escolhido porque ilustra os problemas causados por tipificação fraca. Em contraste, Ada é fortemente tipada, fornecendo assim

uma comparação interessante com C. Finalmente, Perl foi escolhida porque ilustra uma linguagem de scripting tipada dinamicamente. Para professores que incluírem o Capítulo 12, recomendamos que pelo menos uma linguagem não familiar aos alunos seja abordada em profundidade suficiente para se atribuir um bom projeto de programação.

No Capítulo 13 as características de uma linguagem orientada a objetos são exploradas usando Java, Smalltalk e Python. Smalltalk foi incluída por causa da sua simplicidade, Java por causa da sua dominância e Python por causa da sua agilidade em aplicações de scripting. O paradigma orientado a objetos continuará a ser importante no futuro.

Os paradigmas de programação funcional e lógico (Capítulos 14 e 15) continuam a ser distintos nas suas aplicações e respectivos estilos de programação. Nosso tratamento de programação funcional inclui discussões paralelas de Scheme e Haskell, de modo que os alunos possam dominar as idéias de programação funcional estudando uma das linguagens e omitindo a outra.

A terceira seção do texto cobre três *tópicos especiais*:

- Manipulação de eventos
- Concorrência
- Corretude

Os Capítulos 16 e 17, *Programação Orientada a Eventos* e *Programação Concorrente*, representam questões de controle incomuns que podem ser bem tratadas em um estudo de linguagens de programação. Ambos têm tido cada vez mais visibilidade em aplicações de programação, especialmente em computação científica e em sistemas embutidos. Tópicos-chave que os alunos precisam cobrir nestas áreas incluem comunicação, deadlocks, passagem de mensagens, não-determinismo, manipulação de eventos, comunicação interprocessos e uma amostra de muitas e variadas aplicações (sistemas operacionais, interações GUI e sistemas de alarmes de casas) nas quais estes tópicos aparecem com regularidade.

Finalmente, o Capítulo 18 sobre *Corretude de Programa* discute recentes desenvolvimentos no suporte de linguagens de programação a métodos formais. Por exemplo, os desenvolvedores em Spark Ada relatam 100 vezes menos erros com taxas de produtividade 3 a 5 vezes mais altas usando uma técnica chamada *corretude por construção* (Barnes, 2003). Por acreditarmos que o suporte a abordagens mais formais ao projeto de software será cada vez mais importante para as linguagens de programação (assim como a programação orientada a objetos é importante hoje), esperamos que os professores considerem a inclusão deste capítulo nos seus cursos de linguagens de programação.

O Capítulo 18 começa com uma revisão de semânticas axiomáticas (Hoare, 1969) e o seu uso na verificação de programas imperativos. Ele expande esta teoria cobrindo o conceito de *projeto por contrato* (Meyer, 1988) em linguagens orientadas a objetos e sua aplicação usando Java Modeling Language (JML). Este capítulo

também cobre o conceito de indução estrutural e seu uso para provar a correção de programas funcionais.

Alguns tópicos dos últimos capítulos nem sempre fazem parte dos cursos que tratam do estudo de paradigmas e linguagens computacionais. Por exemplo, o estudo sobre programação concorrente, orientada a eventos e corretude de programas podem, cada um, ser visto separadamente.

Além disso, boa parte do material dos capítulos 3, 6, 8, 10 e 11 pode servir de base no estudo sobre compiladores.

PRÉ-REQUISITOS E OUTRAS ORIENTAÇÕES

Os alunos que iniciam esse curso devem ter conhecimento de estruturas de dados. Sendo que, devem ter se familiarizado com listas encadeadas, pilhas, matrizes flexíveis e tabelas hash.

Além disso, conhecimento de Java é um pré-requisito se no curso forem ministrados alguns dos seguintes capítulos sobre implementações: 3, 6, 8, 16 e 17. Sem conhecimento de Java, os alunos devem ter experiência com C, C++ ou C#.

Recomendamos que os alunos também tenham acesso a um bom tutorial de Java, manual de referência e ambiente de programação. Tais tutoriais e referências estão disponíveis na Internet (veja, por exemplo, http://java.sun.com/docs/books/tutorial). Algumas das nossas discussões dependem estritamente de características que foram introduzidas em Java 1.4 e 1.5. Essas características são claramente observadas no texto.

Também esperamos que os alunos tenham algum conhecimento matemático. Embora este não seja um pré-requisito rígido, é essencial para os alunos que queiram se envolver nas seções orientadas a matemática em alguns dos capítulos (3, 6, 8, 10 e 18). Uma revisão das idéias básicas de funções, conjuntos, lógica e prova é fornecida no Apêndice B.

Este texto é consistente com as recomendações do *Computing Curricula* 2001 (CC2001, 2001). Ele também cobre todos os tópicos do curso de Linguagens de Programação descrito no *Liberal Arts Model Curriculum* (Walker e Schneider, 1996) e no esboço da sua versão mais nova de 2005 (www.lacs.edu).

Quanto ao *Computing Curricula* 2001, o material neste texto cobre todos os tópicos (PL1 até PL11) da seção de Linguagens de Programação do corpo de conhecimento básico. Também cobre outros tópicos nesse corpo de conhecimento básico, como programação orientada a eventos e concorrente (PF6), gerenciamento de memória (OS5), programação lógica e funcional (IS) e engenharia de software (SE). Este texto trata de cada tópico em maior profundidade do que a sugerida pelo *Computing Curricula* 2001.

FONTES DE LINGUAGEM E WEBSITE

Para este texto pode ser usado software com qualquer implementação de Java 1.5 ou superior. Implementamos o software Java para este livro usando o Java 1.5 da Sun. A seguir uma lista de sites que recomendamos para obter tutorial e outras informações sobre as principais linguagens cobertas neste livro.

Linguagem	Fonte Web*
Ada	gnu.org
C, C++	gnu.org
Haskell	haskell.org
Java 1.5	java.sun.com
Perl	perl.com
Prolog	swi-prolog.org
Python	python.org
Scheme	drscheme.org
Smalltalk	squeak.org

SUPLEMENTOS

Online Learning Center (Centro de aprendizagem on-line)

O Centro de aprendizagem on-line no endereço www.mhhe.com/tucker oferece recursos para o estudante e para o professor para ajudar a desenvolver os conceitos descritos no livro. Esses materiais estão disponíveis em inglês.

Recursos para o Professor

Para o professor, o Online Learning Center, em www.mhhe.com/tucker traz: apresentações em power point, manual do professor, entre outros. Tudo disponível em inglês.

Para terem acesso aos recursos on-line, os professores brasileiros precisam obter uma senha com a McGraw-Hill Interamericana do Brasil. A senha deve ser solicitada por e-mail: divulgacao_brasil@mcgraw-hill.com. Na Europa, a senha deve ser obtida com a McGraw-Hill de Portugal: servico_clientes@mcgraw-hill.com.

Recursos para o Estudante

O Online Learning Center, em www.mhhe.com/tucker disponibiliza para o estudante:

Um conjunto de software e suporte para acompanhar este texto com os seguintes materiais:

Uma implementação Java completa da sintaxe, sistema de tipos e semântica de Clite.

Agradecimentos

Muitas pessoas nos ajudaram no desenvolvimento deste livro. James Lu foi um colaborador-chave no início da conceitualização da primeira edição. Bill Bynum do College of William and Mary e Laurie King do College of the Holy Cross contribuíram com os Capítulos 4 e 8, respectivamente. David Coppit do William and Mary nos introduziram no uso de árvores de prova, que aparecem no Capítulo 18. Alunos da Bowdoin e William and Mary contribuíram com a primeira edição e em versões iniciais desta edição. De forma especial, Doug Vail desenvolveu soluções para alguns dos problemas mais desafiadores. Wyatt Dumas ajudou a reescrever o software para a segunda edição e deu contribuições significativas para o conteúdo de dois capítulos. Agradecemos a todos os nossos revisores:

Phil Ventura	*University of Buffalo, SUNY*
Aaron Keen	*California Polytechnic State University, San Luis Obispo*
John Donald	*San Diego State University*
Tia Watts	*Sonoma State University*
Ron Olsson	*University of California, Davis*
Thomas D. Rethard	*University of Texas, Arlington*

*NE: Os sites indicados neste livro poderão sofrer alterações ao longo do tempo em razão da natureza dinâmica da Internet.

Alex Thornton	*University of California, Irvine*
Gerald Baumgartner	*Ohio State University*
Ken Slonneger	*University of Iowa*
David R. Falconer	*California State University, Fullerton*
Tae W. Ryu	*California State University, Fullerton*
Qi Cheng	*University of Oklahoma*
Rainey Little	*Mississippi State University*
Jay-Evan J. Tevis	*Auburn University*
John Hannan	*Pennsylvania State University*
Neelam Soundarajan	*Ohio State University*
Robert van Engelen	*Florida State University*
Shannon Tauro	*University of California, Irvine*
Gloria Melara	*California State University, Northridge*
Amer Diwan	*University of Colorado, Boulder*
Susan Gauch	*University of Kansas*
Henri Casanova	*University of California, San Diego*
Cristina V. Lopes	*University of California, Irvine*
Salih Yurttas	*Texas A&M University*
Roman W. Swiniarksi	*San Diego State University*
Amar Raheja	*California State Polytechnic Universty, Pomona*
Franck Xia	*University of Missouri, Rolla*
Rajendra K. Raj	*Rochester Institute of Technology*
Randall D. Beer	*Case Western Reserve University*
Robert M. Cubert	*University of Florida*
Liang Cheng	*Lehigh University*
David Hemmendinger	*Union College*

pela sua leitura cuidadosa e comentários construtivos por todo o desenvolvimento da primeira e segunda edições deste texto, que foram aperfeiçoados pela sua compreensão coletiva. Os autores oferecem um agradecimento especial a David Hemmendinger da Union College pela sua cuidadosa edição e extensivas sugestões, a maioria das quais foi incorporada a esta edição.

Finalmente, gostaríamos de agradecer nossos editores, Rebecca Olsen e Alan Apt, pela sua visão, direção e apoio. Eles orientaram o desenvolvimento da segunda edição com extraordinária habilidade.

Allen B. Tucker
Bowdoin College

Robert E. Noonan
College of William and Mary

Resumo do conteúdo

Sumário

Sobre os Principais Autores

Allen B. Tucker é Professor Pesquisador de Ciências Naturais da Fundação Anne T. and Robert M. Bass no Departamento de Ciências da Computação no Bowdoin College. Graduou-se (Bachelor of Arts) em matemática na Wesleyan University e MS e PhD em ciência da computação na Northwestern University.

O Professor Tucker tem trabalhos publicados nas áreas de linguagens de programação, projeto de software, processamento de linguagem natural e curriculum design. Atua como Fulbright Lecturer na Ternopil Academy of National Economy na Ucrânia, como Erskine Lecturer visitante na University of Canterbury na Nova Zelândia, e Esigelec Lecturer visitante na França. Ele é associado à ACM (Association of Computing Machinery).

Robert E. Noonan é Professor de Ciência da Computação no College of William and Mary, onde lecionou por 30 anos. Possui o grau A.B. (*Artium Baccalaureatus*) em matemática no Providence College e graus MS e PhD em ciência da computação na Purdue University.

Tem publicações nas áreas de linguagens de programação, criação de compiladores e engenharia de software. É membro da ACM (Association of Computing Machinery), SIGPLAN (Special Interest Group of ACM), SIGCSE (Special Interest Group on Computer Science Education), e do Liberal Arts Computer Science (LACS) Consortium.

1 Visão Geral

"Uma boa linguagem de programação é um universo conceitual para pensar em programação."

Alan Perlis

VISÃO GERAL DO CAPÍTULO

Da mesma forma que as nossas linguagens naturais, as linguagens de programação facilitam a expressão e a comunicação de idéias entre pessoas. Entretanto, linguagens de programação diferem das linguagens naturais de duas maneiras importantes. Em primeiro lugar, linguagens de programação também permitem a comunicação de idéias entre pessoas e computadores. Em segundo lugar, as linguagens de programação possuem um domínio de expressão mais reduzido do que o das linguagens naturais. Isso quer dizer que elas facilitam apenas a comunicação de idéias *computacionais*. Assim, uma linguagem de

programação deve satisfazer requisitos diferentes daqueles de uma linguagem natural. Este texto explora esses requisitos e as alternativas de projeto de linguagens que eles evocam.

Neste estudo, identificamos as muitas semelhanças entre linguagens de programação e linguagens naturais. Também examinamos as diferenças fundamentais que são impostas pela configuração computacional na qual um programa deve funcionar. Examinamos as características das linguagens de programação tanto abstratamente quanto ativamente. Isso significa que combinamos um tratamento conceitualmente rico de projeto de linguagens de programação com um estudo prático em laboratório sobre como esses conceitos afetam projetistas de linguagens e programadores em uma ampla faixa de domínios de aplicações.

Este estudo é importante porque os alunos de ciência da computação de hoje serão os projetistas e usuários das linguagens de programação de amanhã. Para se tornar um projetista e usuário de linguagens bem informado você precisará compreender as linguagens de forma ampla – suas características, seus pontos fortes e fracos em uma ampla gama de estilos de programação e suas aplicações. Conhecer uma linguagem e o domínio de sua aplicação não fornece tal amplitude de conhecimento. Este livro lhe ajudará a obter tal amplitude.

1.1 PRINCÍPIOS

Projetistas de linguagens possuem um vocabulário básico sobre a estrutura, o significado e as preocupações pragmáticas dessas ferramentas que os auxiliam a entender como as linguagens funcionam. Esse vocabulário se divide em três categorias principais que chamamos de *princípios* de projeto de linguagens:

- Sintaxe
- Nomes e tipos
- Semântica

Muitos dos conceitos dessas categorias são emprestados da lingüística e da matemática, como veremos a seguir. Juntas, essas categorias fornecem um foco organizacional para os Capítulos básicos 2, 4, 5, 7 e 9, respectivamente. Mais profundidade de estudo em cada categoria é apresentada nos capítulos associados (3, 6, 8, 10 e 11), conforme explicado a seguir.

Sintaxe A *sintaxe* de uma linguagem descreve o que constitui um programa estruturalmente correto. Ela responde a muitas questões. Qual é a gramática para se escrever programas na linguagem? Qual é o conjunto básico de palavras e símbolos que os programadores usam para escrever programas estruturalmente corretos?

Veremos que a maioria da estrutura sintática de linguagens modernas de programação é definida com o uso de um formalismo lingüístico denominado *gramática livre de contexto*. Outros elementos de sintaxe estão fora do domínio de gramáticas livres de contexto e são definidos por outros meios. Um tratamento cuidadoso da sintaxe das linguagens de programação aparece no Capítulo 2.

Um estudo de sintaxe de linguagens traz muitas questões. Como um compilador analisa a sintaxe de um programa? Como os erros de sintaxe são detectados? Como uma gramática livre de contexto facilita o desenvolvimento de um analisador sintático? Essas questões mais profundas sobre sintaxe são abordadas no Capítulo 3.

Nomes e Tipos O vocabulário de uma linguagem de programação inclui um conjunto de regras cuidadosamente projetado para nomear entidades – variáveis, funções, classes, parâmetros e assim por diante. Nomes de entidades também têm outras propriedades durante a vida de um programa, como seu escopo, visibilidade e ligação. O estudo de nomes em linguagens de programação e seu impacto sobre a sintaxe e a semântica de um programa é o assunto do Capítulo 4.

Os *tipos* de uma linguagem denotam os tipos de valores que os programas podem manipular: tipos simples, tipos estruturados e tipos mais complexos. Entre os tipos simples estão valores inteiros, números decimais, caracteres e boleanos. Tipos estruturados incluem strings de caracteres, listas, árvores e tabelas hash. Tipos mais complexos incluem funções e classes. Os tipos são discutidos em maior extensão no Capítulo 5.

Um sistema de tipos permite ao programador entender e implementar apropriadamente operações sobre valores de diversos tipos. Um sistema de tipos especificado cuidadosamente permite ao compilador executar uma rigorosa verificação de tipos em um programa antes da sua execução, evitando assim erros em tempo de execução que podem ocorrer devido a operandos de tipos inapropriados. A especificação completa e a implementação de um sistema de tipos é o foco de um estudo mais aprofundado no Capítulo 6.

Semântica O significado de um programa é definido pela sua *semântica*, ou seja, quando um programa é executado, o efeito de cada comando sobre os valores das variáveis no programa é dado pela semântica da linguagem. Assim, quando escrevemos um programa, devemos entender idéias básicas como o efeito exato que uma atribuição possui sobre as variáveis do programa. Se tivermos um modelo semântico que seja independente de alguma plataforma específica, podemos aplicá-lo a uma diversidade de máquinas nas quais essa linguagem possa ser implementada. Estudamos semântica no Capítulo 7.

A implementação de semântica em tempo de execução também é interessante em um estudo mais profundo de semântica. Como um interpretador funciona e qual a conexão entre ele e a especificação da semântica de uma linguagem? Essas questões mais profundas são estudadas no Capítulo 8.

As funções representam o elemento chave da abstração procedural em qualquer linguagem. Uma compreensão da semântica da definição e chamada de funções é fundamental em qualquer estudo de linguagens de programação. A implementação de funções também requer uma compreensão dos elementos estáticos e dinâmicos de memória, incluindo a *pilha de tempo de execução*. A pilha também nos auxilia a entender outras idéias como o escopo de um nome e o tempo de vida de um objeto. Esses tópicos são tratados no Capítulo 9.

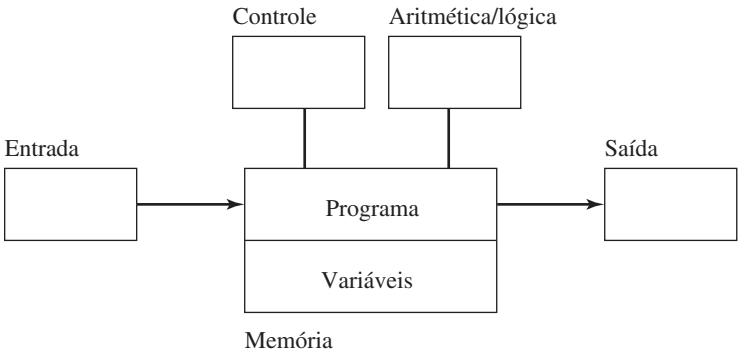
A implementação da pilha de chamadas e retornos de funções é um tópico central que merece estudo mais aprofundado. Além disso, estratégias para o gerenciamento de outra área de memória chamada de *heap* são importantes para a compreensão de objetos dinâmicos como matrizes. Técnicas de gerenciamento de *heap,* chamadas "coleta de lixo", estão muito relacionadas à implementação destes objetos dinâmicos. A pilha e o *heap* são estudados em detalhes nos Capítulos 10 e 11, respectivamente.

1.2 PARADIGMAS

De modo geral, pensamos em um "paradigma" como um padrão de pensamento que guia um conjunto de atividades relacionadas. Um paradigma de programação é um padrão de resolução de problemas que se relaciona a um determinado gênero de programas e linguagens. Quatro paradigmas de programação distintos e fundamentais evoluíram nas últimas três décadas:

- Programação imperativa
- Programação orientada a objeto
- Programação funcional
- Programação lógica

Algumas linguagens de programação são intencionalmente projetadas para suportar mais de um paradigma. Por exemplo, C++ é uma linguagem imperativa e orientada



| Figura 1.1 O modelo computacional de von Neumann-Eckert

a objeto, enquanto a linguagem experimental Leda (Budd, 1995) é projetada para suportar os paradigmas de programação imperativa, orientada a objeto, funcional e lógica. Essas linguagens são reminiscências de trabalhos anteriores (notadamente PL/I, Algol 68 e Ada) para projetar uma única linguagem que fosse mais de propósito geral do que outras linguagens de programação do seu tempo. Com exceção de C++, esses esforços falharam ao tentar atrair interesse sustentado.

Programação Imperativa A *programação imperativa* é o paradigma mais antigo, já que está fundamentado no modelo computacional clássico de "von Neumann-Eckert" (veja a Figura 1.1). Nesse modelo, tanto o programa quanto as suas variáveis são armazenados juntos, e o programa contém uma série de comandos para executar cálculos, atribuir valores a variáveis, obter entradas, produzir saídas ou redirecionar o controle para outro ponto nessa série de comandos.

A abstração procedural é um componente para a programação imperativa assim como as atribuições, os laços, as seqüências, os comandos condicionais e a manipulação de exceções. As linguagens de programação imperativa predominantes incluem Cobol, Fortran, C, Ada e Perl. O paradigma de programação imperativa é o assunto do Capítulo 12.

Programação Orientada a Objeto A *programação orientada a objeto (POO)* fornece um modelo no qual um programa é uma coleção de objetos que interagem entre si, passando mensagens que transformam seu estado. Neste sentido, a passagem de mensagens permite que objetos de dados se tornem ativos em vez de passivos. Essa característica ajuda a distinguir melhor a programação OO da imperativa. A classificação de objetos, herança e a passagem de mensagens são componentes fundamentais da programação OO. Linguagens orientadas a objetos importantes são Smalltalk, C++, Java e C#. A programação OO será estudada no Capítulo 13.

Programação Funcional A *programação funcional* modela um problema computacional como uma coleção de funções matemáticas, cada uma com um espaço de entrada (domínio) e resultado (faixa). Isso separa a programação funcional das linguagens que possuem o comando de atribuição. Por exemplo, o comando de atribuição não

```
x = x + 1
```

faz sentido na programação funcional ou na matemática.

As funções interagem e combinam entre si usando composição funcional, condições e recursão. Importantes linguagens de programação funcional são Lisp, Scheme, Haskell e ML. A programação funcional é discutida e ilustrada no Capítulo 14.

Programação Lógica A *programação lógica* (declarativa) permite a um programa modelar um problema declarando qual resultado o programa deve obter, em vez de como ele deve ser obtido. Às vezes, essas linguagens são chamadas de *baseadas em regras*, já que as declarações do programa se parecem mais com um conjunto de regras ou restrições sobre o problema, em vez de uma seqüência de comandos a serem executados.

Interpretar as declarações de um programa lógico cria um conjunto de todas as soluções possíveis para o problema que ele especifica. A programação lógica também fornece um veículo natural para se expressar o não-determinismo, o que é apropriado para problemas cujas especificações sejam incompletas. A principal linguagem de programação lógica é Prolog, e o paradigma de programação lógica é descrito no Capítulo 15.

1.3 TÓPICOS ESPECIAIS

Além desses quatro paradigmas, diversos tópicos essenciais para o projeto de linguagem de programação merecem abordagem extensiva neste livro-texto. Esses tópicos tendem a ser universais, na medida em que aparecem em dois ou mais dos paradigmas anteriores, em vez de em apenas um. Cada um deles é brevemente introduzido agora:

- Manipulação de eventos
- Concorrência
- Correção

Manipulação de Eventos A *manipulação de eventos* ocorre com programas que respondem a eventos gerados em uma ordem imprevisível. De certa forma, um programa orientado a evento é apenas um programa cujo comportamento é inteiramente determinado por questões de manipulação de eventos. A manipulação de eventos está freqüentemente acoplada ao paradigma orientado a objeto (por exemplo, Java applets), embora ela ocorra dentro dos paradigmas imperativos também (por exemplo, Tcl/Tk). Os eventos se originam de ações dos usuários na tela (cliques de mouse ou pressionamentos de teclas, por exemplo) ou então de outras fontes (como leituras de sensores em um robô). Linguagens importantes que suportam a manipulação de eventos incluem Visual Basic, Java e Tcl/Tk. Esse tópico é apresentado no Capítulo 16.

Concorrência A *programação concorrente* pode ocorrer dentro do paradigma imperativo, orientado a objeto, funcional ou lógico. A concorrência ocorre quando o programa possui uma coleção de elementos assíncronos que podem compartilhar informações ou sincronizar-se entre si em intervalos de tempo. A concorrência também ocorre dentro de um processo individual como a execução paralela de diferentes iterações de um laço. Linguagens de programação concorrente incluem SR (Andrews e Olsson, 1993), Linda (Carriero e Gelenter, 1989) e High Performance Fortran (Adams e outros, 1997). A programação concorrente será estudada no Capítulo 17.

Correção A *correção de programa* é um assunto que, até recentemente, tem atraído interesse apenas acadêmico. Todavia, linguagens mais novas e recursos de linguagens estão evoluindo para suportar o projeto de programas demonstravelmente corretos em uma diversidade de domínios de aplicações. Um programa é *correto* se satisfaz sua especificação formal para todas as suas entradas possíveis. A prova de correção é um assunto

complexo, mas ferramentas de linguagem para tratamento formal da correção por programadores estão agora se tornando disponíveis. O sistema Spark/Ada (Barnes, 2003) e a Java Modeling Language (Leavens et al., 1998) fornecem bons exemplos. Introduzimos o tópico de correção de programas no Capítulo 18.

1.4 UMA BREVE HISTÓRIA

As primeiras linguagens de programação foram as linguagens de máquina e as linguagens *assembly* dos primeiros computadores, começando na década de 1940. Centenas de linguagens de programação e dialetos foram desenvolvidos desde então. A maioria teve tempo de vida e utilidade limitados, enquanto algumas tiveram amplo sucesso em um ou mais domínios de aplicações. Muitas desempenharam um papel importante na influência sobre o projeto de futuras linguagens.

Um resumo do desenvolvimento histórico de diversas linguagens de programação influentes aparece na Figura 1.2. Embora certamente não seja completa, a Figura 1.2 identifica alguns dos eventos e das tendências mais influentes. Cada seta na Figura 1.2 indica uma significativa influência do projeto de uma linguagem mais antiga para uma sucessora.

A década de 1950 marcou o início da era das "linguagens de ordem mais alta" (abreviadas em inglês como HOLs). Uma HOL se distingue de uma linguagem de máquina ou assembly porque seu estilo de programação é independente de alguma arquitetura de máquina específica. As primeiras linguagens de ordem mais alta foram Fortran, Cobol, Algol e Lisp. Fortran e Cobol sobreviveram e se desenvolveram bastante desde o seu surgimento no final da década de 1950. Essas linguagens tiveram muitos seguidores e possuem uma grande quantidade de código legado no qual os programadores de hoje realizam manutenção. Por outro lado, Lisp teve substancialmente seu uso diminuído e Algol desapareceu completamente.

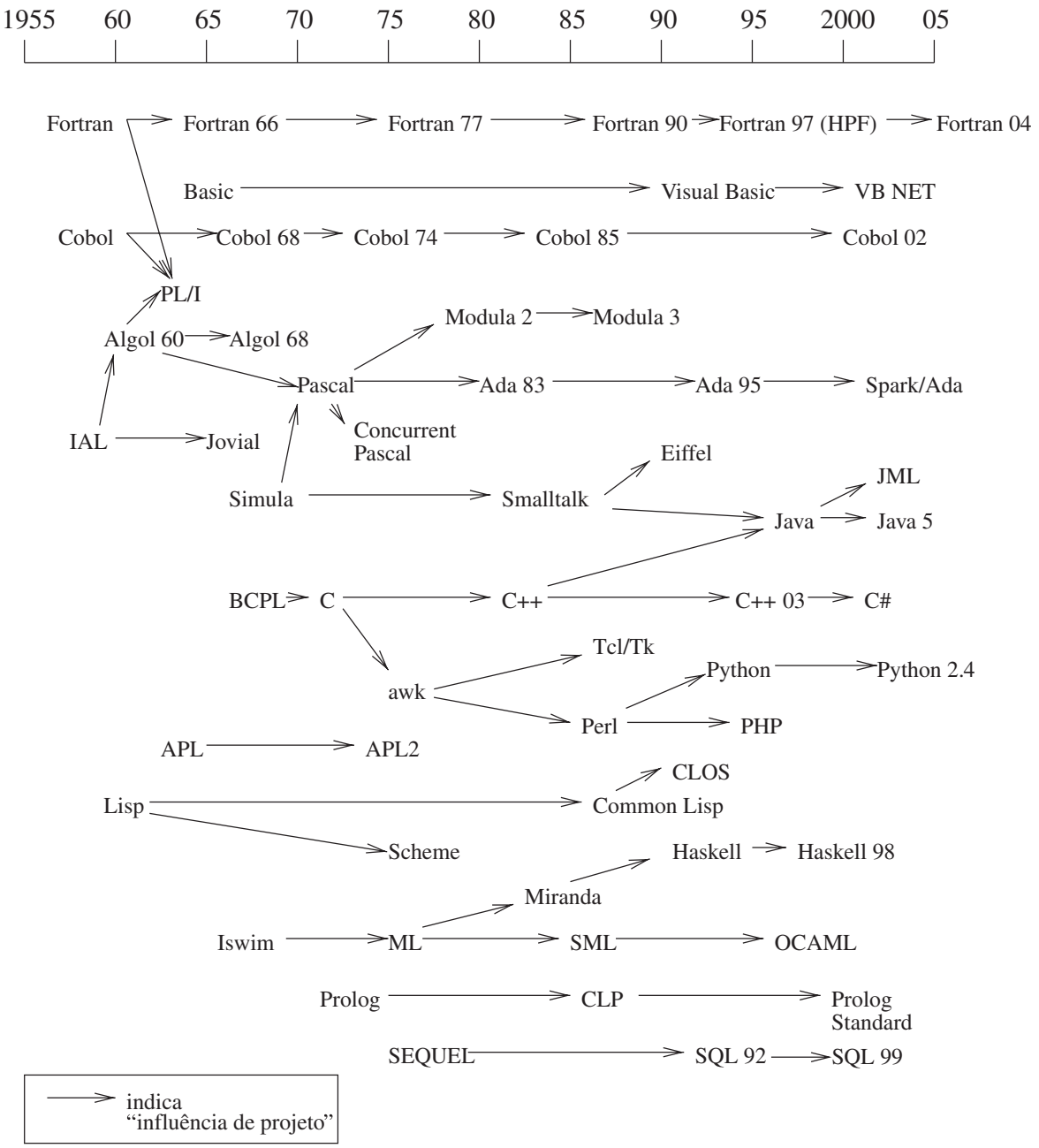
Contudo, os projetos inovadores dessas primeiras linguagens tiveram grande influência sobre suas sucessoras. Por exemplo, a demonstração de Fortran de que notação algébrica poderia ser traduzida em código eficiente agora é admitida como padrão, assim como a introdução da estrutura de registros em Cobol, o projeto de Pascal para a compilação em um passo e a demonstração de Algol de que uma gramática lingüística poderia definir formalmente sua sintaxe.

Talvez a maior motivação para o desenvolvimento de linguagens de programação nas últimas décadas tenha sido a rápida evolução da demanda de poder computacional e as novas aplicações por parte de uma grande e diversa comunidade de usuários. As seguintes comunidades de usuários podem reivindicar uma participação importante no cenário das linguagens de programação:

- Inteligência artificial
- Educação
- Ciência e engenharia
- Sistemas de informação
- Sistemas e redes
- World Wide Web

Os domínios dos problemas computacionais dessas comunidades são todos diferentes, assim como as principais linguagens de programação que se desenvolveram em torno delas. A seguir, esboçamos os principais objetivos computacionais e os projetos de linguagem que servem a cada uma dessas comunidades.

Inteligência Artificial A comunidade de programação de inteligência artificial está ativa desde o início da década de 1960. Esta comunidade se preocupa com o



| Figura 1.2 Um Resumo da História das Linguagens de Programação

desenvolvimento de programas que modelam o comportamento da inteligência humana, a dedução lógica e a cognição. A manipulação de símbolos, as expressões funcionais e o projeto de sistemas de prova lógica têm sido os objetivos centrais desse trabalho em andamento.

Os paradigmas da *programação funcional* e da *programação lógica* se desenvolveram bastante por intermédio dos esforços dos programadores de inteligência artificial.

Linguagens de programação funcional proeminentes nos últimos anos incluem Lisp, Scheme, ML e Haskell. As linguagens de programação lógica proeminentes incluem Prolog e CLP.

A primeira linguagem de IA, Lisp (um acrônimo de "*List Processor*"), foi projetada por John McCarthy em 1960. A Figura 1.2 sugere que Lisp foi dominante nos primeiros anos e tornou-se menos dominante nos anos mais recentes. Entretanto, as características básicas de Lisp motivaram o desenvolvimento de linguagens mais recentes como Scheme, ML e Haskell. A forte relação entre Lisp e *lambda calculus* (um formalismo para modelar a natureza de funções matemáticas) fornece uma base matemática sólida para a posterior evolução desses sucessores. O *lambda calculus* e seu relacionamento com linguagens funcionais são mais explicados no Capítulo 14.

Na área de programação lógica, apenas uma linguagem, a Prolog, tem sido o ator principal, e a Prolog tem tido pouca influência sobre o projeto de linguagens em outras áreas de aplicação.

Educação Nas décadas de 1960 e 1970, diversas linguagens importantes foram projetadas com o objetivo principal de ensinar programação aos alunos. Por exemplo, Basic foi projetada na década de 1960 por John Kemeny para facilitar o aprendizado de programação por meio de *compartilhamento de tempo*, uma arquitetura na qual um único computador é conectado diretamente a diversos terminais de uma só vez. Cada usuário de terminal compartilha tempo no computador recebendo regularmente uma pequena "fatia de tempo" de poder computacional. Basic teve grande popularidade no decorrer dos anos, especialmente como uma linguagem de ensino em escolas de segundo grau e em programas de ciência de nível superior.

A linguagem Pascal, uma derivativa de Algol, foi projetada na década de 1970 com o propósito de ensinar programação. Pascal serviu durante muitos anos como a principal linguagem de ensino no currículo de ciência da computação em nível superior.

Durante a última década, essas linguagens têm sido substituídas em programas educacionais por linguagens de "apelo comercial" como C, C++ e Java. Essa mudança tem benefícios e desvantagens. Por um lado, aprender uma linguagem de apelo comercial fornece aos graduados uma ferramenta de programação que eles podem usar imediatamente quando se iniciarem na profissão de computação. Por outro lado, tal linguagem é inerentemente mais complexa e menos prática de aprender como primeira linguagem em um curso de graduação.

O recente surgimento de Python pode fornecer um meio pelo qual cursos introdutórios de ciência da computação podem retornar à simplicidade e se concentrar novamente no ensino dos primeiros princípios. Por exemplo, Python possui uma sintaxe e uma semântica mais transparentes, o que a torna mais acessível de ser bem entendida por um novato do que qualquer uma das suas alternativas com apelo comercial. Além disso, cursos introdutórios que utilizam Python parecem inserir uma variedade mais rica de tópicos de ciência da computação do que cursos usando C, C++ ou Java.

Ciência e Engenharia A comunidade de programação científica e de engenharia desempenhou um papel importante no início da história da computação e continua a desempenhar um papel importante atualmente. Os primeiros programas foram escritos na década de 1940 para prever as trajetórias balísticas durante a Segunda Guerra Mundial, usando a bem conhecida fórmula física que caracteriza corpos em movimento. Esses programas foram escritos primeiro em linguagem de máquina e *assembly*, por matemáticos especialmente treinados.

Uma força motriz importante por trás de aplicações científicas e de engenharia por toda a história é a necessidade de se obter o maior poder de processamento possível. O poder de processamento dos supercomputadores de hoje é medido em *teraflops* (trilhões

de operações de ponto flutuante por segundo), e o líder atual trabalha a uma velocidade de 280 teraflops executando um teste de desempenho-padrão chamado LINPAK (veja mais informações em **www.top500.org**). Muitas das aplicações científicas e de engenharia de hoje são modelos de sistemas naturais complexos em áreas como a bioinformática e as ciências da terra e atmosféricas.

A primeira linguagem de programação científica, Fortran I, foi projetada por John Backus na IBM em 1954 (Backus et al., 1954). O acrônimo "Fortran" é uma abreviação de "*For*mula *Tran*slator" (Tradutor de Fórmulas). Fortran é provavelmente a linguagem de programação científica mais usada atualmente.

Versões iniciais de Fortran tiveram muitos problemas, todavia. O mais difícil era a questão da consistência – o mesmo programa em Fortran era executado de forma diferente em máquinas diferentes ou então nem era executado. Esses problemas geraram diversos esforços novos. Um destes produziu a linguagem Algol, abreviação de "*Algo*rithmic *L*anguage" (Linguagem Algorítmica), que foi projetada por um comitê internacional em 1959. O principal objetivo do projeto de Algol era fornecer uma linguagem mais bem definida que Fortran tanto para a computação quanto para a apresentação de algoritmos científicos e matemáticos.

Algol foi chamada originalmente de "*I*nternational *A*lgebraic *L*anguage" (IAL – Linguagem Algébrica Internacional). A linguagem Jovial foi projetada por Jules Schwartz na década de 1960 para refinar e aumentar os recursos de IAL. Esse acrônimo significa "*J*ules' *O*wn *V*ersion of the *I*nternational *A*lgebraic *L*anguage" (Versão Própria de Jules da Linguagem Algébrica Internacional). *Jovial* foi amplamente usada em aplicações do Departamento de Defesa dos Estados Unidos.

Outra linguagem interessante denominada APL (abreviação de "*A P*rogramming *L*anguage" – Uma Linguagem de Programação) (Iverson, 1962) foi projetada por Kenneth Iverson na década de 1960 para facilitar a programação rápida de álgebra de matrizes e outros cálculos matemáticos. APL tinha um conjunto de caracteres estendidos que incluía operadores de matrizes de símbolo único que poderiam substituir o tédio da escrita de laços for na maioria dos casos. A proliferação de tais símbolos especiais requereu o projeto de um teclado especializado para facilitar a digitação de programas em APL. Tais programas ficaram conhecidos pela sua brevidade; o cálculo de uma matriz que requeria um laço for explícito em uma linguagem convencional só precisava de um único símbolo em APL. A brevidade da APL também foi sua maldição aos olhos de muitas pessoas, ou seja, a maioria dos programas em APL era tão sucinta que desafiava a compreensão de todos que não fossem técnicos capacitados. O custo do suporte a um conjunto de caracteres especializado e um teclado estilizado também contribuiu para o fim da APL.

Até hoje, a computação científica permanece como uma atividade central na história da programação e das linguagens de programação. O domínio de seus problemas se refere principalmente à execução de cálculos complexos de forma muito rápida e precisa. Os cálculos são definidos por modelos matemáticos que representam fenômenos científicos. Eles são implementados principalmente por meio do uso do paradigma de *programação imperativa.* Linguagens modernas de programação que são amplamente usadas na área da programação científica incluem Fortran 90 (Chamberland, 1995), C (Kernighan e Ritchie, 1988), C++ (Stroustrup, 1997) e High Performance Fortran (Adams e outros, 1997).

Quanto mais complexo o fenômeno científico se torna, maior a necessidade de linguagens de programação e computadores altamente sofisticados e paralelos. Assim, a *programação concorrente* é fortemente motivada pelas necessidades de tais aplicações científicas como a modelagem de sistemas climáticos e de fluxo do oceano. Algumas linguagens, como High Performance Fortran, suportam programação concorrente por meio da inclusão de recursos a uma linguagem básica amplamente usada (Fortran). Outras, como SR e Occam, são projetadas especificamente para suportar programação concorrente. Linguagens de propósito geral, como Java, suportam concorrência como um dos seus muitos objetivos de projeto.

Sistemas de Informação Programas projetados para instituições gerenciarem seus sistemas de informação provavelmente são os mais prolíficos do mundo. As corporações perceberam, na década de 1950, que o uso de computadores poderia reduzir bastante o tédio da manutenção de registros e melhorar a precisão e a confiabilidade do que eles poderiam realizar. Os sistemas de informação encontrados em corporações incluem o sistema de folha de pagamento, o sistema de contabilidade, os sistemas de marketing e as vendas on-line, os sistemas de estoque e produção, e assim por diante. Tais sistemas se caracterizam pela necessidade de processar grandes quantidades de dados (muitas vezes organizados nos chamados bancos de dados), mas requerem transformações relativamente simples sobre os dados que estão sendo processados.

Tradicionalmente, os sistemas de informação têm sido desenvolvidos em linguagens de programação como Cobol e SQL. Cobol foi projetada primeiramente no final da década de 1950 por um grupo de representantes da indústria que queriam desenvolver uma linguagem que fosse portável em uma diversidade de diferentes arquiteturas de máquina. Cobol significa "*CO*mmon *B*usiness *O*riented *L*anguage" (Linguagem Comum Orientada aos Negócios), usa a língua inglesa como base da sua sintaxe e suporta um estilo de programação *imperativo*.

Programas em Cobol são construídos a partir de cláusulas, sentenças e parágrafos e, geralmente, tendem a ser mais prolixos que programas similares em outras linguagens. O objetivo era definir uma linguagem que fosse fácil para os programadores assimilarem. Se esse objetivo foi ou não atingido é algo que ainda está aberto a discussão. Apesar disso, Cobol se tornou rapidamente, e ainda continua sendo, a linguagem de programação mais amplamente usada para aplicações de sistemas de informação.

Em contraste, SQL (Pratt, 1990) emergiu na década de 1980 como uma ferramenta de programação *declarativa* para especificação de bancos de dados, geração de relatórios e recuperação de informações. SQL significa "*S*tructured *Q*uery *L*anguage" (Linguagem de Consulta Estruturada) e é a linguagem predominante usada para a especificação e a recuperação de informações de bancos de dados relacionais. O modelo relacional de bancos de dados é amplamente usado, em parte por causa do seu forte fundamento matemático em álgebra relacional.

Mais recentemente, os negócios desenvolveram ampla gama de aplicações de comércio eletrônico. Estas aplicações muitas vezes usam um modelo "cliente-servidor" de projeto de programa, que interage com os usuários em locais remotos e fornece acesso simultâneo a um banco de dados compartilhado. Um bom exemplo desse modelo é um sistema de encomenda de livros on-line, no qual o banco de dados reflete o estoque de livros da empresa e a interação auxilia o usuário na pesquisa pelo banco de dados, na seleção dos livros e no processo de encomenda. A *programação orientada a evento* é essencial nessas aplicações, e os programadores combinam linguagens como Java, Perl, Python e SQL para implementá-las.

Sistemas e Redes Programadores de sistemas projetam e realizam a manutenção do software básico que executa os sistemas – componentes do sistema operacional, software de rede, compiladores e depuradores de linguagens de programação, máquinas virtuais e interpretadores, sistemas embarcados e de tempo real (em telefones celulares, ATMs, aeronaves etc.). Esses tipos de software estão intimamente relacionados com as arquiteturas de máquinas específicas, como a Intel/AMD x86 e a Apple/Motorola/IBM PowerPC.

A maioria desses programas é escrita em C, que permite que os programadores cheguem a um nível muito próximo do nível de linguagem de máquina. A programação de sistemas é geralmente feita com o uso do paradigma de projeto *imperativo*. Todavia, os programadores de sistemas também devem lidar com a programação *concorrente* e *orientada a eventos,* além de ter uma preocupação especial com a *correção* do programa.

Assim, o exemplo primário de uma linguagem de programação de sistemas é C, projetada no início da década de 1970, em parte para suportar a codificação do sistema operacional Unix. De fato, cerca de 95% do código do sistema Unix é escrito em C. C++ foi projetado por Bjarne Stroustrup na década de 1980 como uma extensão de C para fornecer novos recursos que suportariam a programação orientada a objeto.

O nome da linguagem de programação Ada vem de Ada Lovelace, que, acredita-se, pode ter sido a primeira pessoa a programar computadores. No início da década de 1800, ela trabalhava com o inventor do computador Charles Babbage. O desenvolvimento da Ada foi financiado pelo Departamento de Defesa dos Estados Unidos, cuja idéia original era ter uma única linguagem que suportasse todas as aplicações desse departamento, especialmente aplicações de controle e sistemas embarcados. Embora Ada nunca tenha atingido esse objetivo específico, seu projeto tem alguns recursos notáveis. Hoje, Ada oferece uma base robusta sobre a qual o compilador Spark disponibiliza ferramentas de suporte à correção de programas.

Linguagens de *script* são largamente usadas atualmente para uma diversidade de tarefas de sistema. Por exemplo, um programa *awk* pode ser projetado rapidamente para verificar a consistência em um arquivo de senhas de uma máquina Unix. Algumas das principais linguagens de *script* são *awk* (Kernighan e Pike, 1984), Perl (Wall et al., 1996b), Tcl/Tk (Ousterhout, 1994) e Python (Lutz, 2001). Tratamos das linguagens de *script* no Capítulo 12, no qual a programação em Perl é explorada em algum detalhe.

World Wide Web A área mais dinâmica para novas aplicações de programação é a Internet, que é o veículo que permite o comércio eletrônico e uma ampla gama de aplicações acadêmicas, governamentais e industriais. A noção de computação centrada na Web e, conseqüentemente, de programação centrada na Web é motivada por um modelo interativo no qual um programa permanece ativo continuamente esperando pelo próximo evento, respondendo a este evento e retornando a seu estado continuamente ativo.

Linguagens de programação que suportam a computação centrada na Web usam a programação *orientada a evento*, que promove a interação sistema-usuário. A computação centrada na Web também usa o paradigma *orientado a objeto*, já que diversas entidades que aparecem na tela do usuário são modeladas de forma mais natural como objetos que enviam e recebem mensagens. As linguagens de programação que suportam computação centrada na Web incluem Perl, PHP (Hughes, 2001), Visual Basic, Java e Python.

1.5 SOBRE O PROJETO DE LINGUAGEM

O projeto de linguagem de programação é um desafio enorme. Os projetistas de linguagem são as pessoas que criam um meio de linguagem que permite aos programadores resolver problemas complexos. Para atingirem seu objetivo os projetistas devem trabalhar dentro de diversas restrições práticas e adotar objetivos específicos que se combinam para fornecer um foco a esse desafio. Esta seção dá uma visão geral desses objetivos e dessas restrições de projeto.

1.5.1 Restrições de Projeto

Os seguintes elementos de configurações computacionais impõem importantes restrições para os projetistas de linguagem:

- Arquitetura
- Configuração técnica
- Padrões
- Sistemas legados

Arquitetura Linguagens de programação são projetadas para computadores. Este fato é tanto uma bênção quanto uma maldição para os projetistas de linguagem. É uma bênção porque uma linguagem bem projetada e implementada pode melhorar muito a utilidade do computador em um domínio de aplicação. É uma maldição porque a maioria dos projetos de computadores nas últimas décadas foram limitados pelas idéias de arquitetura do modelo clássico de von Neumann-Eckert, discutido anteriormente. Muitas linguagens, como Fortran, Cobol e C, estão de acordo com essa arquitetura, enquanto outras, como Lisp, não estão.

Por alguns anos se tornou interessante considerar a idéia de arquitetura computacional como derivada do projeto de linguagem, em vez de como uma precursora. Na década de 1960, Burroughs projetou a B5500, que tinha uma arquitetura de pilha especialmente apropriada para executar programas em Algol. Outro esforço produziu o tipo de máquinas Lisp, que emergiram no início da década de 1980. Essas máquinas eram configuradas de modo que os programas em Lisp fossem executados eficientemente nelas e tiveram certo grau de sucesso durante alguns anos. Entretanto, as arquiteturas de máquina Lisp foram ofuscadas no final da década de 1980 pelo advento das arquiteturas RISC (*Reduced Instruction Set Computer* – Computadores com Conjunto Reduzido de Instruções), nas quais programas em Lisp podiam ser implementados eficientemente.

Assim, quando consideramos as virtudes de diversas opções de projeto de linguagens, sempre ficamos restritos pela necessidade de implementar a linguagem eficientemente e de forma eficaz dentro das restrições impostas pelas variações atuais do modelo clássico de von Neumann. A noção de que um bom projeto de linguagem pode levar a uma arquitetura computacional radicalmente nova e comercialmente viável possivelmente não é provável.

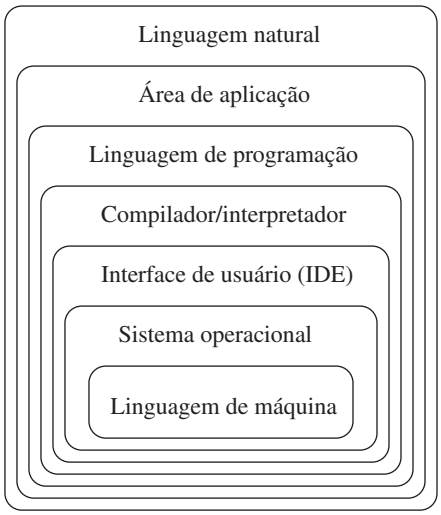
Configuração Técnica As linguagens são restritas não apenas pelos limites das arquiteturas de computadores, mas também devem satisfazer a outras restrições impostas pelas configurações técnicas nas quais elas são usadas: a área de aplicação, o sistema operacional, o IDE (*Integrated Development Environment* – Ambiente de Desenvolvimento Integrado), a rede e as outras preferências de uma determinada comunidade de programação. Por exemplo, Fortran é utilizado em determinadas plataformas por diferentes compiladores para se adaptar às necessidades de programadores científicos. Estes programadores trabalham em diversas profissões que usam seus estilos próprios de projeto de software, ferramentas e (acima de tudo) suas próprias linguagens naturais para comunicar-se entre eles. Essa visão do cenário complexo para o projeto de linguagens é resumida na Figura 1.3.

Algumas linguagens são intencionalmente mais de *uso geral* no seu projeto, tendo como objetivo ser úteis aos interesses de uma ampla gama de aplicações. Por exemplo, Ada (The Department of Defense, 1983) foi projetada para ser útil para todas as aplicações suportadas pelo Departamento de Defesa dos Estados Unidos, enquanto Cobol (Brown, 1977) foi projetada para suportar todas as aplicações orientadas a negócio. Embora Cobol tenha sido moderadamente bem-sucedida no seu objetivo, Ada o foi muito menos.

Outras linguagens são projetadas para serem de *propósito específico* na sua natureza. Por exemplo, Prolog (Clocksin e Mellish, 1997) foi projetada para servir aos interesses restritos do processamento de linguagem natural, às provas de teoremas e aos sistemas especializados. C foi projetada principalmente para suportar os interesses da programação de sistemas, embora tenha, desde então, sido adotada por uma faixa mais ampla de aplicações. Spark/Ada e JML foram projetadas, respectivamente, para suportar a prova formal de correção de programas em Ada e Java.

Padrões Quando uma linguagem de programação tem um uso suficientemente amplo entre programadores, geralmente se inicia um processo de *padronização*, ou seja, é feito um esforço para definir um padrão independente da máquina da linguagem à qual

Figura 1.3 Níveis de Abstração na Computação



todos os seus programadores devem aderir. A padronização da linguagem geralmente a estabiliza em diferentes plataformas e grupos de programação, viabilizando a portabilidade dos programas.

As duas maiores organizações que supervisionam e mantêm padrões de linguagens de programação são a American National Standards Institute (ANSI) e a International Standards Organization (ISO). Diversas linguagens foram padronizadas com o passar dos anos desde que o processo de padronização de linguagens começou. Algumas delas, com suas datas de padronização, são:

ANSI/ISO Cobol (2002)

ISO Fortran (2004)

ISO Haskell (1998)

ISO Prolog (2000)

ANSI/ISO C (1999)

ANSI/ISO C++ (2003)

ANSI/ISO Ada (2005)

ANSI Smalltalk (2002)

ISO Pascal (1990)

O processo de padronização de linguagens é complexo e lento, com muito tempo de envolvimento da comunidade e geralmente tem uma definição volumosa da linguagem-padrão como resultado.

A padronização de linguagens de programação é acompanhada da padronização de conjuntos de caracteres (por exemplo, os conjuntos ASCII e UNICODE) e bibliotecas (por exemplo, a C++ Standard Template Library) que suportam diretamente as atividades de programação.

O valor da padronização para a comunidade é que os projetistas de software e hardware participam do processo e se comprometem com suas implementações de compiladores e in-

terpretadores para que estejam em conformidade com o padrão. Tal conformidade é essencial na manutenção da portabilidade em diferentes plataformas de hardware e compiladores.

Alguns têm argumentado que a padronização de linguagens é uma influência negativa porque inibe inovações no projeto de linguagens, ou seja, versões-padrão das linguagens tendem a durar muito tempo, perpetuando, dessa forma, a vida de características pobres junto aos seus recursos mais valiosos. Entretanto, os padrões ISO e ANSI são revisados a cada cinco anos, o que fornece certa proteção contra a obsolescência prolongada.

Mais informações sobre padrões específicos de uma linguagem e o próprio processo de padronização podem ser encontradas em **www.ansi.org** e **www.iso.org**.

Sistemas Legados É bem sabido que a maior parte do tempo de um programador é gasto na manutenção de *sistemas legados*. Tais sistemas são aqueles artefatos de software que foram projetados e implementados pela equipe anterior de programação, mas que são mantidos e atualizados pela equipe atual. A maior parte de código em sistemas legados está provavelmente escrita em Cobol, a linguagem de programação predominante para sistemas de informação nas últimas quatro décadas.

Para suportar a manutenção de código legado, versões atualizadas e melhoradas de linguagens antigas devem ser *compatíveis para trás* com seus predecessores, ou seja, os programas antigos devem continuar a ser compilados e executados quando novos compiladores forem desenvolvidos para a versão atualizada. Assim, todos os recursos semânticos e sintáticos, mesmo os que forem menos desejáveis a partir do ponto de vista estético, não podem ser retirados sem prejudicar a integridade do código legado.

Por esse motivo, linguagens de programação mais antigas se tornaram sobrecarregadas com recursos quando novas versões surgiram; as linguagens raramente se tornam mais compactas enquanto se desenvolvem. Isso é particularmente verdadeiro para Fortran, Cobol e C++, que foi projetada como uma verdadeira extensão de C para manter compatibilidade para trás com código legado.

O projeto de Java – embora muitos dos seus recursos lembrem C++ –, afastou-se dessa tradição. Como tema central, os projetistas de Java queriam liberar sua nova linguagem de ter que suportar os recursos menos desejáveis de C++, de forma que eles simplesmente os cortaram. O resultado foi uma linguagem mais compacta, pelo menos temporariamente, já que versões recentes de Java adicionaram muitos recursos novos sem retirar um conjunto comparativamente grande de recursos obsoletos. Talvez seja inevitável que quando qualquer linguagem amadurece se torne naturalmente mais carregada de recursos para atender às crescentes demandas do domínio das aplicações.

1.5.2 Resultados e Objetivos

À luz desses requisitos, somos levados a fazer duas perguntas importantes:

1. Como uma linguagem de programação surge e se torna bem-sucedida?
2. Quais são as características-chave que tornam uma linguagem de programação ideal?

Examinando rapidamente o passado, primeiro observamos que algumas linguagens de programação bem-sucedidas foram projetadas por indivíduos, outras, por comitês de toda uma indústria, e outras ainda foram o produto de um forte apoio por parte dos seus desenvolvedores comerciais. Por exemplo, Lisp e C++ foram projetadas principalmente por indivíduos (John McCarthy e Bjarne Stroustrup, respectivamente), enquanto

Algol, Cobol e Ada foram projetadas por comitês[1]. PL/I, Java e C# são produtos dos seus desenvolvedores comerciais corporativos (IBM, Sun e Microsoft, respectivamente). Assim, não está claro se o processo de projeto – individual, comitê ou alavancagem corporativa – possui uma influência total sobre o sucesso de um projeto de linguagem.

Já que este estudo tem como objetivo preparar os leitores para avaliar e comparar linguagens de programação de modo geral, é importante termos um pequeno conjunto de características-chave por meio das quais você possa fazer isso. Nós as chamaremos de *objetivos de projeto*, já que têm servido como medidas eficazes de projetos bem-sucedidos de linguagens com o passar dos anos:

- Simplicidade e legibilidade
- Clareza nas ligações
- Confiabilidade
- Suporte
- Abstração
- Ortogonalidade
- Implementação eficiente

Simplicidade e Legibilidade Os programas devem ser fáceis de escrever. Também devem ser compreensíveis e fáceis de ler pelo programador mediano. Uma linguagem de programação ideal deve, então, suportar a escrita e a leitura. Além disso, deve ser fácil de aprender e ensinar.

Algumas linguagens, como Basic, Algol e Pascal, foram intencionalmente projetadas para facilitar a clareza de expressão. Basic, por exemplo, tinha um conjunto de instruções muito pequeno. Algol 60 tinha uma "linguagem de publicação" que fornecia um formato-padrão para a composição de programas que apareciam em artigos publicados. Pascal foi projetada explicitamente como uma linguagem para ensino, com recursos que facilitavam o uso de princípios da programação estruturada.

Outras linguagens foram projetadas para minimizar o número total de teclas pressionadas que era necessário para expressar um algoritmo ou a quantidade de armazenamento que o compilador requereria. Com certeza, os projetistas de APL e C valorizaram essas economias.

Clareza nas Ligações Um elemento de linguagem está *conectado* a uma propriedade quando essa propriedade é definida para ele. Uma boa linguagem deve ser muito clara sobre quando ocorre a ligação principal para cada elemento com sua propriedade. Aqui estão os principais tempos de ligação.

- Tempo de definição da linguagem: Quando a linguagem é definida, os tipos básicos de dados são ligados a rótulos especiais denominados *palavras reservadas*, que os representam. Por exemplo, números inteiros (*integers*) são ligados ao identificador `int`, e números reais (fracionários) são ligados a `float` na linguagem C.
- Tempo de implementação da linguagem: Quando o compilador ou interpretador da linguagem é escrito, os valores são ligados a representações de máquina. Por exemplo, o tamanho de um valor `int` em C é determinado no momento da implementação da linguagem.

1. No caso de Ada, o processo de projeto também teve um elemento de competição – diversos projetos competindo foram avaliados, e Ada surgiu como a linguagem mais propícia para satisfazer às necessidades do Departamento de Defesa dos Estados Unidos da América.

- Tempo de escrita do programa: Quando os programas são escritos em algumas linguagens, os nomes de variáveis são ligados a tipos, os quais permanecem ligados com esses nomes por toda a execução do programa. Por exemplo, uma variável pode ser ligada ao seu tipo quando ela é declarada, como na declaração

```
int x;
```

  que liga a variável `x` ao tipo `int`.
- Tempo de compilação: Quando os programas são compilados, os comandos e as expressões desses programas são ligados a seqüências de instruções em linguagem de máquina equivalente.
- Tempo de carga do programa: Quando o código de máquina é carregado, as variáveis estáticas são atribuídas a endereços fixos de memória, a pilha de tempo de execução é alocada a um bloco de memória, assim como o próprio código de máquina.
- Tempo de execução do programa: Quando os programas estão sendo executados, as variáveis são ligadas a valores, como na execução da atribuição `x = 3`.

Às vezes, um elemento pode ser ligado a uma propriedade em qualquer uma dentre diversas alternativas de tempos nessa série contínua. Por exemplo, a ligação de um valor com uma constante pode ser feita em tempo de compilação/carga de um programa ou no início da execução. Quando tais escolhas são possíveis, a noção de *ligação precoce* significa simplesmente que um elemento é ligado a uma propriedade o mais rápido possível (em vez de mais tarde) nessa série de tempo. A *ligação tardia* significa um atraso até a última oportunidade possível.

Como veremos no Capítulo 4, a ligação precoce leva a uma melhor detecção de erros e geralmente é menos custosa. Todavia, ligações tardias levam a uma maior flexibilidade de programação (conforme ilustrado no Capítulo 14). De modo geral, um projeto de linguagem deve considerar todas essas alternativas, e decisões sobre ligações são tomadas, em última instância, pelo projetista da linguagem.

Confiabilidade O programa se comporta da mesma forma toda vez que ele é executado com os mesmos dados de entrada? Ele se comporta da mesma forma quando é executado em diferentes plataformas? O seu comportamento pode ser especificado independentemente de modo a permitir a sua verificação formal (ou informal)?

Particularmente pertinente a essas questões é a necessidade de se projetar mecanismos apropriados de manipulação de exceções na linguagem. Além disso, linguagens que restrinjam o uso de aliases e vazamento de memória, que suportem tipagem forte, tenham sintaxe e semântica bem definidas e que suportem a verificação e validação de programas têm uma vantagem nessa categoria.

Suporte Uma boa linguagem de programação deve ser facilmente acessível por alguém que queira aprendê-la e instalá-la no seu próprio computador. De forma ideal, seus compiladores devem ser de domínio público, em vez de serem propriedade de uma corporação e caros para se obter. A linguagem deve ser implementada em múltiplas plataformas. Cursos, livros-texto, tutoriais e um grande número de pessoas familiarizadas com a linguagem são vantagens que ajudam a preservar e estender a vitalidade de uma linguagem.

Questões relacionadas a custo podem ser de maior preocupação para alunos e programadores individuais, em vez de para funcionários governamentais e corporativos, cujos custos de software são geralmente cobertos por suas empresas. A história das linguagens de programação tem apresentado sucesso em ambos os lados. Por exemplo, C, C++ e Java são linguagens não proprietárias, disponíveis no domínio público para uma ampla variedade de plataformas. Por outro lado, C# e Eiffel são

linguagens suportadas por empresas cujo uso é restringido pelo seu custo e as plataformas/IDEs nas quais elas são implementadas.

Abstração A abstração é um aspecto fundamental do processo de projeto de programas. Os programadores gastam muito tempo construindo abstrações, tanto de dados quanto procedurais, para explorar a reutilização de código e evitar reinventá-lo. Uma boa linguagem de programação suporta tão bem a abstração de dados e procedural que é a ferramenta de projeto preferida na maioria das aplicações.

Bibliotecas que acompanham linguagens modernas de programação comprovam a experiência acumulada de programadores na construção de abstrações. Por exemplo, as bibliotecas de classes Java contêm implementações de estruturas de dados básicos (como vetores e pilhas) que, em linguagens anteriores, tinham de ser projetadas explicitamente pelos próprios programadores. Com qual freqüência nós reinventamos um algoritmo de ordenação ou uma estrutura de dados de lista encadeada que provavelmente já tenha sido implementado milhares de vezes antes?

Ortogonalidade Uma linguagem é dita *ortogonal* se os seus comandos e recursos são construídos sobre um conjunto pequeno e mutuamente independente de operações primitivas. Quanto mais ortogonal uma linguagem, menos regras excepcionais são necessárias para escrever programas corretos. Assim, programas em uma linguagem ortogonal muitas vezes tendem a ser mais simples e mais claros do que aqueles em uma linguagem não-ortogonal.

Como um exemplo de ortogonalidade, considere a passagem de argumentos em uma chamada de função. Uma linguagem integralmente ortogonal permite que qualquer tipo de objeto, incluindo uma definição de função, seja passado como argumento. Veremos exemplos disso na nossa discussão sobre programação funcional no Capítulo 14.

Outras linguagens restringem os tipos de objetos que podem ser passados em uma chamada. Por exemplo, a maioria das linguagens imperativas não permite que definições de funções sejam passadas como argumentos e, portanto, não são ortogonais em relação a isso.

A ortogonalidade tende a se correlacionar com a simplicidade conceitual, já que o programador não precisa manter muitas regras excepcionais na sua cabeça. Alan Perlis coloca isso da seguinte maneira:

> É melhor fazer cem funções operarem em uma estrutura de dados do que dez funções em dez estruturas de dados.

Por outro lado, a não-ortogonalidade, muitas vezes está relacionada à eficiência porque suas regras excepcionais eliminam as opções de programação que consumiriam tempo ou espaço.

Implementação Eficiente Os recursos e as construções de uma linguagem devem permitir uma implementação prática e eficiente em plataformas contemporâneas.

Como um contra-exemplo, Algol 68 era um projeto de linguagem elegante, mas suas especificações eram tão complexas que era (quase) impossível implementá-las efetivamente. Versões iniciais de Ada foram criticadas pelas suas características de tempo de execução ineficientes, já que Ada foi projetada, em parte, para suportar programas que seriam executados em "tempo real". Programas embarcados em sistemas como aeronaves tinham de responder imediatamente a uma mudança brusca nos valores de entrada, como vento e velocidade. Programas em Ada ficavam na interface entre os sensores que forneciam as leituras e os mecanismos que deveriam responder a elas. Implementações iniciais de Ada ficavam longe desses ambiciosos objetivos de desempenho. Os críticos mais severos do desempenho de Ada ficaram conhecidos por dizer "Bom, existe 'tempo real' e 'tempo de Ada'"!

Implementações iniciais de Java receberam críticas por esse mesmo motivo, embora refinamentos recentes no sistema de compilação Java da Sun tenham melhorado seu desempenho em tempo de execução.

1.6 COMPILADORES E MÁQUINAS VIRTUAIS

Qualquer implementação de uma linguagem de programação requer que os programas nessa linguagem sejam analisados e então traduzidos em um formato que possa ser:

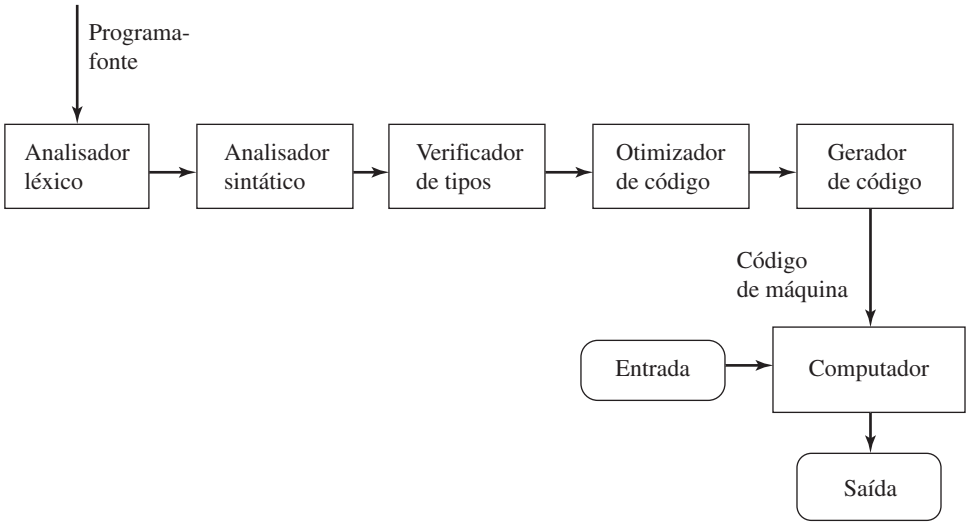
1. Executado por um computador (por exemplo, uma "máquina real") ou
2. Executado por um interpretador (por exemplo, um software que simule uma "máquina virtual" e seja executado em uma máquina real).

A tradução do primeiro tipo é, muitas vezes, chamada de *compilação*, enquanto a tradução do segundo tipo é chamada *interpretação*.

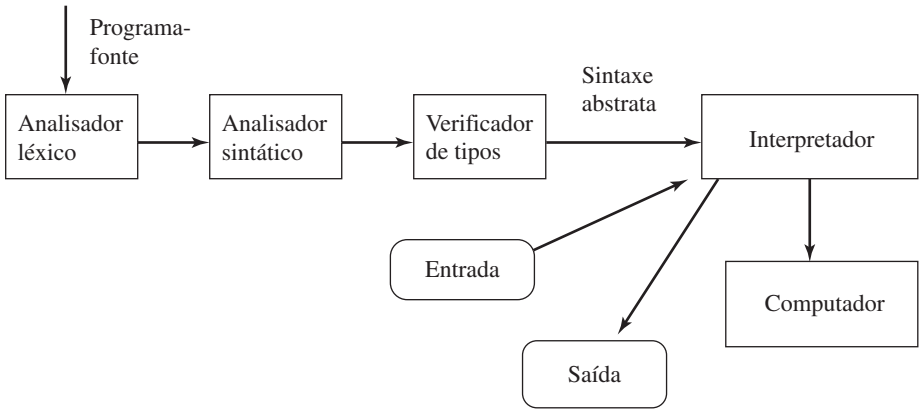
Compiladores O processo de *compilação* traduz um programa-fonte para a linguagem de um computador. Depois, o *código de máquina* resultante pode ser executado nesse computador. Esse processo é mostrado na Figura 1.4. Por exemplo, Fortran, Cobol, C e C++ são linguagens compiladas típicas.

Os cinco passos do processo de compilação são análise léxica, análise sintática, verificação de tipos, otimização de código e geração de código. Os três primeiros passos se relacionam a encontrar e relatar erros para o programador. Os dois últimos passos se relacionam à geração de código de máquina eficiente para ser executado no computador-alvo.

O código de máquina de um programa compilado se combina com sua entrada para ser executado em um passo separado que segue a compilação. Erros em tempo de execução geralmente são rastreáveis no programa-fonte por meio do uso de um depurador.



| Figura 1.4 O Processo de Compilação e Execução



| Figura 1.5 Máquinas Virtuais e Interpretadores

Teremos algumas oportunidades de explorar os três primeiros passos da compilação nos Capítulos 3 e 6. Entretanto, os assuntos de geração e otimização de código são normalmente estudados em um curso de compilador, e não serão abordados neste texto.

Máquinas Virtuais e Interpretadores Outras linguagens são implementadas por meio do uso de um processo *interpretativo*, conforme mostrado na Figura 1.5. Aqui, o programa fonte é traduzido para uma forma abstrata intermediária, que é então executada de modo interpretado. Lisp e Prolog, por exemplo, são, muitas vezes, implementadas usando interpretadores (embora compiladores para essas linguagens também existam).

Como a Figura 1.5 sugere, os primeiros três passos de um compilador também acontecem em um interpretador. Entretanto, a representação abstrata do programa que sai desses três passos se torna o objeto de execução de um interpretador. O próprio interpretador é um programa que executa os passos do programa abstrato enquanto é executado em uma máquina real. O interpretador geralmente é escrito em uma linguagem distinta da que está sendo interpretada.

Às vezes, uma linguagem é projetada de modo que o compilador seja escrito uma só vez, direcionando o código para uma *máquina virtual* abstrata, e então essa máquina virtual é implementada por um interpretador em cada uma das diferentes máquinas reais. Este é o caso de Java, cuja máquina abstrata foi chamada de Java Virtual Machine (JVM) (Lindholm e Yellin, 1997). Ao fazer essa escolha, os projetistas da linguagem Java desistiram um pouco da eficiência (já que código interpretado geralmente requer mais recursos do que código de máquina) em favor de flexibilidade e portabilidade, ou seja, qualquer mudança na especificação da linguagem Java pode ser implementada pela alteração de um único compilador, em vez de uma família de compiladores.

Uma vantagem importante da compilação sobre a interpretação é que o desempenho em tempo de execução de um programa compilador geralmente é mais eficiente (em tempo e espaço) do que seu desempenho interpretado. Por outro lado, a qualidade da interação com o sistema que o programador desfruta pode ser melhor com um interpretador do que com um compilador. Já que o desenvolvimento de programas é interativo, segmentos individuais de programas (funções) podem ser testados na medida em que são

projetados, em vez de esperar até que o programa inteiro no qual eles devem ser inseridos seja completado[2].

Versões posteriores de Java obtiveram de volta um pouco da eficiência perdida pela inserção de um compilador *just-in-time* (JIT) da JVM. Esse recurso permite que o *byte code* da JVM seja traduzido durante a execução em código nativo da máquina hospedeira antes que seja executado.

O conceito de máquina virtual é valioso por outros motivos que podem compensar sua inerente perda de eficiência. Por exemplo, é oportuno implementar uma linguagem e seu interpretador usando uma máquina virtual existente, visando ao projeto ou à experimentação com a própria linguagem. De forma semelhante, é útil estudar fundamentos e paradigmas de projeto de linguagens usando um *interpretador* disponível que facilite a experimentação e a avaliação de programas nessa linguagem.

O conceito de máquina virtual possui um valor prático imediato neste estudo, ou seja, você terá acesso a um interpretador para um pequeno subconjunto C denominado *Clite*. O uso do Clite facilita muito o nosso trabalho porque elimina detalhes específicos de máquina que muitas vezes podem esconder os princípios e as idéias que seriam ensinadas. Por exemplo, no estudo da sintaxe da linguagem, você pode exercitar o interpretador Clite para explorar a estrutura sintática de diferentes elementos da linguagem como expressões aritméticas e laços.

1.7 RESUMO

O estudo de linguagens de programação inclui princípios, paradigmas e tópicos especiais. Os princípios são estudados tanto conceitualmente quanto na prática, por intermédio do interpretador Clite.

O domínio de um ou mais novos paradigmas – programação imperativa, orientada a objeto, funcional ou lógica – também é importante para este estudo. Essa atividade nos ajuda a apreciar uma faixa mais ampla de aplicações computacionais e descobrir abordagens para a resolução de problemas com os quais ainda não estamos familiarizados.

A investigação de um ou mais tópicos especiais – manipulação de eventos, concorrência ou correção – nos permite examinar com cuidado três características específicas de projeto de linguagem e os desafios de programação que envolvem sua utilização efetiva.

No total, esperamos que este estudo ajude a ampliar sua visão e suas habilidades técnicas no vasto cenário das linguagens de programação. Em especial, você deve esperar obter:

- A apreciação do uso de ferramentas práticas para examinar os princípios de projeto de linguagem.
- A apreciação do valor de diferentes paradigmas de programação que sejam especialmente poderosos em domínios específicos de aplicações.
- Experiências em laboratório com novas linguagens e ferramentas de projeto, tanto para testar os princípios quanto para dominar novas técnicas de resolução de problemas.

Para apoiar este estudo você pode ocasionalmente visitar o site do livro, que traz também outros materiais de apoio pedagógico e de aprendizado.

2. Entretanto, o desenvolvimento de depuradores e IDEs modernos para linguagens compiladas tem neutralizado substancialmente essa vantagem nos últimos anos.

EXERCÍCIOS

1.1 Uma pesquisa on-line sobre "linguagens de programação" produzirá *links* para fontes de informação confiáveis de todas as linguagens de programação importantes do passado e do presente. Para cada uma das seguintes linguagens, use a Internet para aprender algo sobre ela. Escreva, com suas próprias palavras, um breve resumo (um parágrafo) das suas características diferenciais, assim como sua relação histórica com outras linguagens que a precederam ou seguiram.

(a) Eiffel
(b) Perl
(c) Python

1.2 Dê um exemplo de um comando em C, C++ ou Java, que seja especialmente ilegível. Reescreva esse comando em um estilo mais legível. Por exemplo, você já viu a expressão `A[i++]` em um programa C/C++?

1.3 Código ilegível não é algo exclusivo de C, C++ e Java. Considere as seguintes opiniões fortemente defendidas sobre as fraquezas de determinadas linguagens nas últimas quatro décadas ou mais:

> "É praticamente impossível ensinar boa programação a alunos que tiveram uma exposição anterior a BASIC; como programadores em potencial, eles ficam mentalmente mutilados além da esperança de regeneração." E. Dijkstra
>
> "O uso de COBOL enfraquece a mente; seu ensinamento, portanto, deve ser considerado como ofensa criminal." E. Dijkstra
>
> "APL é um equívoco levado à perfeição. É a linguagem do futuro para as técnicas de programação do passado: ela cria uma nova geração de programadores preguiçosos." E. Dijkstra
>
> "Não existe agora, e nem existirá, uma linguagem de programação na qual seja difícil escrever programas ruins." L. Flon

(a) Dijkstra parece não ter muita consideração com Basic, Cobol ou APL. Entretanto, ele *teve* muita consideração com Algol e suas sucessoras. Faça uma leitura eficaz na Internet para determinar quais as características gerais que Algol possuía que a tornava superior a linguagens como Basic, Cobol e APL.
(b) O que Flon quer dizer com essa última declaração? As linguagens de programação são inerentemente imperfeitas? Ou ele está sugerindo que os programadores são inerentemente ineptos? Ou existe uma interpretação intermediária a essas duas? Explique.

1.4 Dê uma característica de C, C++ ou Java que ilustre a ortogonalidade. Dê uma característica diferente da discutida no texto que ilustre a não-ortogonalidade.

1.5 Duas implementações diferentes de uma linguagem são *incompatíveis* se houver programas que sejam executados de forma diferente (dão resultados diferentes) sob uma implementação em relação à outra. Após ler em um site e em outras fontes sobre versões anteriores de Fortran, você pode determinar se ela tinha versões incompatíveis? De que forma específica (tipo de comando, tipo de dado etc.) esta incompatibilidade aparecia? O que pode ser feito para eliminar incompatibilidades entre duas implementações diferentes de uma linguagem?

1.6 O trabalho de padronização para a linguagem C começou em 1982 com um grupo de trabalho ANSI, e o primeiro padrão C foi concluído em 1989. Este trabalho foi posteriormente aceito como um padrão ISO em 1990 e continua sendo até hoje. Leia o suficiente na Internet para determinar quais alterações significativas foram feitas no padrão C desde 1990.

1.7 Encontre o padrão C++ na Internet. O que significa *não conformidade* quando o padrão discute uma característica da linguagem suportada por um determinado compilador? Para o compilador C++ no seu computador, há características que não possuem conformidade?

1.8 Após aprender o suficiente no site de Java (**java.sun.com**) e em outras fontes, o que você pode dizer sobre a situação do trabalho de padronização de Java pela ANSI e pela ISO neste momento?

1.9 Encontre a versão Python 2.4 na Internet. Quais são as características novas que essa versão adiciona a Python 2.3? Quais características antigas da Python são eliminadas pela versão mais nova, se houver alguma?

1.10 Compare duas linguagens que você conheça usando os objetivos do projeto de linguagem destacados na Seção 1.5.2. Para cada objetivo, determine qual das duas linguagens satisfaz melhor esse objetivo e justifique sua conclusão com um exemplo. Ou seja, na comparação de C e Java você poderia concluir que C possui implementação mais eficiente porque ele compila em código nativo em vez de em um interpretador.

2 Sintaxe

"... uma linguagem que é simples para ser analisada pelo compilador também é simples para ser analisada pelo programador humano."

Niklaus Wirth (1974)

VISÃO GERAL DO CAPÍTULO

Quando pensamos em um programa, distinguimos sua forma (como ele é escrito) do seu significado (o que acontece quando o executamos). De modo geral, a forma de um programa é a sua *sintaxe*, enquanto o seu significado é denominado *semântica*. Os compiladores detectam erros de sintaxe nos nossos programas, como erros de tipo e de

declaração, enquanto os erros em tempo de execução, como uma divisão por zero, são de natureza semântica. Podemos definir sintaxe mais precisamente da seguinte maneira:

> **Definição:** A *sintaxe* de uma linguagem de programação é uma descrição precisa de todos os seus programas gramaticalmente corretos.

A sintaxe pode ser descrita por um conjunto de regras, da mesma forma que uma linguagem natural. Por exemplo, a regra de que uma sentença deve ser finalizada com um terminador apropriado (um ponto final, um ponto de interrogação ou de exclamação) é parte da sintaxe da língua portuguesa. Para uma linguagem de programação, uma descrição clara e concisa da sua sintaxe é especialmente importante; sem tal especificação, escritores de compiladores e programadores não trabalhariam bem.

Métodos formais para definir a sintaxe têm sido usados desde o surgimento de Algol no início da década de 1960 [Naur (ed.), 1963] e têm sido usados para definir a sintaxe da maioria das linguagens desde então. Neste capítulo, exploramos o uso tanto de métodos formais quanto de informais para definir a sintaxe de uma linguagem de programação nos níveis léxico, concreto e de sintaxe abstrata.

> **Definição:** A *sintaxe léxica* de uma linguagem define as regras para os símbolos básicos, incluindo identificadores, literais (por exemplo, inteiros e pontos flutuantes), operadores e pontuação.
>
> **Definição:** A *sintaxe concreta* de uma linguagem se refere à representação real desses programas usando símbolos léxicos como seu alfabeto.
>
> **Definição:** A *sintaxe abstrata* considera apenas as informações essenciais dos programas, sem se preocupar com idiossincrasias sintáticas como pontuação ou parênteses.

A sintaxe abstrata é uma ferramenta muito útil para relacionar as expressões sintáticas de uma linguagem com a sua semântica, como veremos em capítulos posteriores.

Para auxiliar a ilustrar nosso estudo, introduzimos a minilinguagem imperativa simples Clite, que será uma ferramenta útil para examinarmos as propriedades sintáticas e semânticas de linguagens de programação. Clite possui as seguintes características:

- É um subconjunto de C/C++, de modo que os programas escritos em Clite serão compilados e executados com o uso de um compilador padrão C/C++, só precisando de alguns `#include` padrão.
- Diferentemente de C e C++, Clite possui uma especificação sintática concisa, que facilita a discussão de conceitos fundamentais de sintaxe.
- A simplicidade de Clite facilita a implementação de um interpretador para ela, fornecendo um suplemento prático para a aprendizagem.

2.1 GRAMÁTICAS

A sintaxe de uma linguagem pode ser amplamente especificada por meio de um formalismo denominado *gramática*. Uma gramática é escrita em uma linguagem de descrição de linguagens, ou *metalinguagem*, e seu propósito é definir todas as *strings* (cadeias de caracteres) permitidas que puderem formar um programa sintaticamente correto. Gramática é uma metalinguagem fundamentada em uma teoria formal desenvolvida pelo lingüista Noam Chomsky (1957), que definiu quatro níveis de gramática, conhecidos como *regular, livre de contexto, sensível ao contexto* e *irrestrita*. A hierarquia de Chomsky é apresentada com mais detalhes na Seção 3.1.

Definição: Uma *gramática livre de contexto* possui um conjunto de produções *P*, um conjunto de símbolos terminais *T* e um conjunto de símbolos não-terminais *N*, um dos quais, *S*, é distinguido como o *símbolo inicial*.

Definição: Uma *produção* gramatical possui a forma $A \rightarrow \omega$, em que *A* é um símbolo não-terminal e ω é uma *string* de símbolos não-terminais e terminais.

Uma forma de gramática livre de contexto, denominada Forma de Backus-Naur (abreviada como BNF, de Backus-Naur Form) tem sido amplamente usada para definir a sintaxe das linguagens de programação.

2.1.1 Gramáticas na Forma de Backus-Naur (BNF)

Em 1960, a BNF foi adaptada da teoria de Chomsky por John Backus e Peter Naur para expressar uma definição sintática formal para a linguagem de programação Algol [Naur (ed.), 1963]. Da mesma forma que em muitos textos, usamos o termo *gramática BNF* como sinônimo de gramática livre de contexto. Veja na Observação (pág. 26) uma discussão mais completa sobre as diferenças entre as duas.

Uma produção é uma regra para reescrita que pode ser aplicada a uma *string* de símbolos denominada *forma sentencial*. Uma produção é interpretada da seguinte maneira: o símbolo *A* é, muitas vezes, chamado de *lado esquerdo*, enquanto a *string* ω é chamada de *lado direito* da produção. Nas gramáticas BNF, os conjuntos de símbolos terminais e não-terminais são disjuntos.

Quando uma gramática BNF é usada para definir a sintaxe de uma linguagem de programação, os *não-terminais N* identificam as categorias gramaticais da linguagem como *Identificador, Inteiro, Expressão, Comando* e *Programa*. O símbolo inicial *S* identifica a principal categoria gramatical sendo definida pela gramática (normalmente *Programa*) e é geralmente definido na primeira produção. Os *símbolos terminais T* formam o alfabeto básico a partir do qual os programas são construídos.

Para ilustrar essas idéias, aqui está um par de produção que define a sintaxe da categoria gramatical *dígitoBinário*:

dígitoBinário $\rightarrow$ 0

dígitoBinário $\rightarrow$ 1

Este par define um *dígitoBinário* como 0 ou 1, mas nada mais além disso. Os símbolos não-terminais são todos os símbolos que aparecem no lado esquerdo de, pelo menos, uma produção. Para a gramática anterior, *dígitoBinário* é o único não-terminal. Os símbolos terminais são todos os outros símbolos que aparecem nas produções; para a gramática anterior, 0 e 1 são os símbolos terminais.

Quando em uma série de produções todas tiverem o mesmo símbolo não-terminal nos seus lados esquerdos, elas podem ser combinadas em uma única produção. Por exemplo, as duas produções anteriores podem ser abreviadas como:

dígitoBinário $\rightarrow$ 0 | 1

Neste caso, as alternativas são separadas por uma barra vertical (|), que significa literalmente "ou", de forma que a interpretação permanece a mesma que a do par de produções original. Neste exemplo, tanto a seta para a direita quanto a barra vertical são *metassímbolos*, que são símbolos que fazem parte da metalinguagem e não fazem parte da linguagem que está sendo definida.

Observação

Forma de Backus-Naur

A BNF começou como uma metalinguagem específica usada para definir a sintaxe da Algol 60. Os não-terminais consistiam de nomes que eram escritos entre sinais (< e >), que também apareciam no lado esquerdo de pelo menos uma produção. Todos os outros símbolos menos os metassímbolos :: = e | eram interpretados como símbolos terminais.

Assim, a gramática para um *Inteiro* apareceria na BNF como:

```
<inteiro> :: = <dígito> | <inteiro> | <dígito>
<dígito> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Mais tarde, um padrão ISO (ISO 14977 [1996]) foi desenvolvido para gramáticas BNF, mas tem sido amplamente ignorado.

A notação BNF original tentou incorporar as limitações do conjunto de caracteres dos mecanismos de entrada da época. Como a tecnologia dos computadores possibilitou o uso dos dispositivos-padrão de entrada para criar notações mais expressivas, as notações mais limitadas foram abandonadas. Vemos um exemplo disso neste texto, que contém símbolos matemáticos cuja digitação pelos autores seria inimaginável há 40 anos.

Em homenagem ao trabalho pioneiro de Backus e Naur no desenvolvimento da BNF e no seu uso para definir a sintaxe de Algol, usamos o termo *gramática BNF* como sinônimo do termo *gramática livre de contexto* usado por Chomsky.

O lado direito de uma produção pode ser qualquer seqüência de símbolos terminais e não-terminais, permitindo que uma variedade de construções interessantes sejam definidas concisamente. Considere a seguinte gramática BNF $G_{inteiro}$, que define a categoria gramatical *Inteiro* como uma seqüência de *Dígitos* decimais.

Inteiro → *Dígito* | *Inteiro Dígito*
Dígito → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Aqui, a segunda produção define os dígitos decimais habituais. A primeira produção permite que um *Inteiro* seja um *Dígito* apenas ou um *Inteiro* seguido por um *Dígito*. A segunda alternativa ilustra o uso de recursão para definir uma seqüência de dígitos arbitrariamente longa. Esta produção define assim um *Inteiro* como uma seqüência de um ou mais dígitos.

2.1.2 Derivações

Para determinar se uma determinada *string* de símbolos pertence a uma categoria gramatical, as regras de produção para essa categoria podem ser usadas para derivar a *string*. Por exemplo, suponha que queiramos determinar se 352 é um *Inteiro*. Para fazer isso, podemos desenvolver uma *derivação* para essa *string* usando as regras de produção da gramática.

1. Primeiro, escreva o símbolo inicial *Inteiro*.
2. Substitua o *Inteiro* pela *string Inteiro Dígito*, o que é permitido pela segunda alternativa no lado direito da primeira regra de produção.
3. Substitua *Inteiro Dígito* por *Inteiro* nesta *string*, usando novamente a mesma regra, produzindo a *string Inteiro Dígito Dígito*.

4 Substitua *Dígito* por *Inteiro* nessa *string*, desta vez usando a primeira alternativa da primeira regra de produção, obtendo *Dígito Dígito Dígito*.
5 Substitua 3 como um tipo específico de *Dígito* usando a segunda regra de produção, obtendo a *string* 3 *Dígito Dígito*.
6 Substitua 5 pelo próximo *Dígito* nessa *string*, obtendo a *string* 3 5 *Dígito*.
7 Finalmente, substitua 2 por *Dígito* nessa *string*, derivando a *string* 352.

Tecnicamente, uma *derivação* é uma seqüência de *strings* separadas pelo símbolo ⇒ no qual em cada etapa um não-terminal é substituído pelo lado direito de uma de suas produções. A primeira *string* da série é a categoria gramatical desejada e a última é a *string* a ser derivada. A seqüência anterior de passos é, portanto, apropriadamente escrita da seguinte maneira:

Inteiro ⇒ *Inteiro Dígito*
⇒ *Inteiro Dígito Dígito*
⇒ *Dígito Dígito Dígito*
⇒ *3 Dígito Dígito*
⇒ *35 Dígito*
⇒ *352*

Aqui, cada instância de ⇒ denota a aplicação de uma única regra de produção para transformar uma *string* um passo mais próximo da *string* a ser derivada. Cada *string* em tal derivação é chamada de *forma sentencial*, a qual pode conter tanto símbolos terminais quanto não-terminais. Por exemplo, a forma sentencial *3 Dígito Dígito* ocorre no quarto passo dessa derivação.

Assim, acabamos de derivar a *string* 352 como uma instância da categoria gramatical *Inteiro* de acordo com a gramática $G_{inteiro}$. Nossa derivação é uma prova por construção de que

Inteiro ⇒ * 352

Isso significa que a *string* 352 pode ser derivada da categoria gramatical *Inteiro* em zero ou mais passos. O símbolo ⇒ * é uma instância da notação de estrela de Kleene, na qual a estrela * é usada para denotar zero ou mais ocorrências do símbolo à sua esquerda, neste caso, o *símbolo de derivação* (⇒).

Agora podemos definir precisamente o significado do termo *linguagem* a partir de um ponto de vista puramente sintático:

> **Definição:** A *linguagem L* definida por uma gramática BNF *G* é o conjunto de todas as *strings* terminais que podem ser derivadas do símbolo inicial.

Por exemplo, a linguagem definida pela gramática $G_{inteiro}$ é o conjunto de todas as *strings* que são seqüências finitas de dígitos decimais, formando assim a categoria sintática *Inteiro*. A *string* 352 é um membro específico dessa linguagem, já que podemos derivá-la usando as produções de $G_{inteiro}$. A derivação dada anteriormente é chamada de *derivação mais à esquerda* porque, a cada etapa, substituímos o não-terminal mais à esquerda na forma seqüencial por uma de suas alternativas. Muitas outras seqüências de derivação são possíveis. Por exemplo, poderíamos substituir

o não-terminal mais à direita em cada passo, em cujo caso teríamos uma *derivação mais à direita*, ou seja:

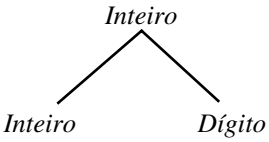
Inteiro ⇒ *Inteiro Dígito*
⇒ *Inteiro 2*
⇒ *Inteiro Dígito 2*
⇒ *Inteiro 52*
⇒ *Dígito 52*
⇒ *352*

2.1.3 Árvores de Análise

Outra forma de demonstrar que uma determinada *string* faz parte da linguagem definida por uma gramática BNF é descrever a derivação em uma forma gráfica. Esta forma é chamada de *árvore de análise*, na qual cada passo da derivação corresponde a uma nova subárvore. Por exemplo, o passo da derivação:

Inteiro ⇒ *Inteiro Dígito*

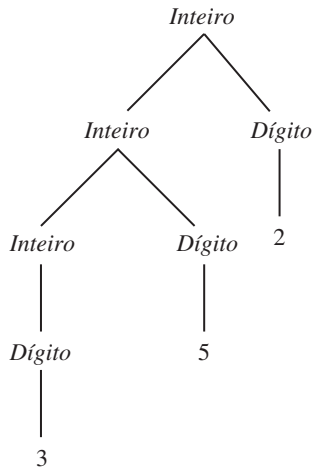
pode ser escrito como a subárvore:

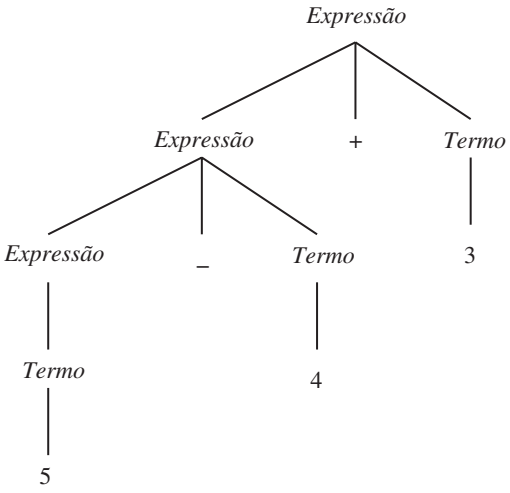


A raiz de cada subárvore em uma árvore de análise é o modo correspondente ao não-terminal para o qual a substituição está sendo feita em uma derivação, e os filhos imediatos da esquerda para a direita correspondem ao lado direito da produção usada nessa substituição.

Uma árvore de análise completa para a derivação da *string* 352 é mostrada na Figura 2.1. Observe que, uma vez que a árvore de análise é desenhada, a ordem dos passos da derivação é perdida; tanto as derivações mais à esquerda quanto as mais à direita resultam na mesma árvore.

Figura 2.1 Árvore de Análise para 352 como um *Inteiro*





| Figura 2.2 Análise da *String* 5 − 4 + 3

A análise da árvore possui diversas outras características notáveis. Primeiro, o nó da raiz da árvore de análise sempre contém o símbolo inicial da gramática (*Inteiro,* neste exemplo). Em segundo lugar, cada nó interno contém um não-terminal (neste caso, *Inteiro* ou *Dígito*). Na verdade, sempre há o mesmo número de nós internos na árvore de análise que são passos (⇒) na derivação (6, neste exemplo). Em terceiro lugar, cada nó interno tem como seus descendentes diretos os elementos que aparecem no lado direito de uma regra gramatical, na ordem da esquerda para a direita. Finalmente, as folhas da árvore de análise são sempre símbolos terminais da gramática. Ler os nós folha da esquerda para a direita reconstrói a *string* que for analisada (352, neste exemplo).

Resumindo, uma derivação é uma representação linear simples de uma árvore de análise, e muitas vezes é mais útil quando a *string* derivada não possui uma estrutura gramatical de interesse. Uma árvore de análise é preferida quando a estrutura gramatical é mais complexa, como veremos em diversos exemplos a seguir.

Considere a gramática G_0, que define a linguagem de expressões aritméticas tendo os operadores + e − e os operandos inteiros de um dígito.[1]

Expressão → *Expressão* + *Termo* | *Expressão* − *Termo* | *Termo*
Termo → 0 | ...| 9 | (*Expressão)*

Uma árvore de análise para a *string* 5 − 4 + 3 para a gramática G_0 é demonstrada na Figura 2.2.

A estrutura dessa árvore de análise é um pouco mais interessante do que para a gramática $G_{inteiro}$. Uma interpretação da esquerda para a direita das folhas dessa árvore sugere que a expressão é avaliada como se tivesse sido escrita (5 − 4) + 3. Esta interpretação produz um resultado numérico diferente do que se calcularmos 5 − (4 + 3). Isso reflete o fato de que o operando da esquerda do operador + é uma *Expressão* da qual

1. Nas gramáticas para uso humano, seqüências longas de alternativas são, muitas vezes, encurtadas. Assim, 0 | ... | 9 significa 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 na segunda regra de G_0.

subseqüentemente deriva a *string* 5 − 4. De forma semelhante, a *string* 3, que é derivada do *Termo* à direita do operador +, é seu operando direito.

2.1.4 Associatividade e Precedência

O exemplo da Seção 2.1.3 mostra que a estrutura de uma árvore de análise pode ser usada para interpretar o significado de uma *string* derivada. Aqui exploramos como a estrutura de uma gramática pode ser usada para definir a associatividade e a precedência de operadores em expressões.

> **Definição:** Um operador possui maior *precedência* do que outro se o primeiro precisar ser analisado antes em todas as expressões sem parênteses que envolverem apenas os dois operadores.

Na matemática comum, nas expressões 5 + 4 × 3 e 5 × 4 + 3, a multiplicação é sempre avaliada antes da adição e, assim, possui maior precedência. Assim, essas expressões são equivalentes a 5 + (4 × 3) e (5 × 4) + 3, respectivamente.

> **Definição:** *Associatividade* especifica se os operadores de precedência igual devem ser executados da esquerda para a direita ou da direita para a esquerda.

Na matemática comum, o operador binário "menos" possui associatividade esquerda, de modo que a expressão 5 − 4 − 3 é equivalente a (5 − 4) − 3.

Considere novamente a árvore de análise para a *string* 5 − 4 + 3 apresentada na Figura 2.2. Em um sentido muito real, a ocorrência dos operadores específicos + e − é irrelevante na determinação da estrutura da árvore de análise. Para a *string* 5 + 4 − 3 obtemos a mesma estrutura de árvore, exceto os operadores + e − que estão alternados. De fato, a mesma estrutura de árvore ocorre para as *strings* 5 + 4 + 3 e 5 − 4 − 3. A partir dessa observação, podemos concluir que os operadores + e − possuem a mesma precedência nessa gramática. Isso acontece porque cada um dos operadores + e − ocorre como uma alternativa na definição da *Expressão*.

Além disso, o uso de recursão esquerda nas regras de produção para a *Expressão* na gramática G_0 torna esses operadores associativos à esquerda. Se a recursão à direita tivesse sido usada na gramática, os operadores teriam sido de associação direita.

Com esta compreensão, podemos definir uma gramática G_1 um pouco mais interessante para os operadores binários +, −, *, /, % (resto) e ** (exponenciação)[2]. Em uso matemático normal, os primeiros quatro operadores são associativos pela esquerda, enquanto o operador de exponenciação é associativo pela direita (ou seja, 2^{3^4} significa 2^{81} em vez de 8^4).

$$
\begin{aligned}
\textit{Expressão} &\rightarrow \textit{Expressão} + \textit{Termo} \mid \textit{Expressão} - \textit{Termo} \mid \textit{Termo} \\
\textit{Termo} &\rightarrow \textit{Termo} * \textit{Fator} \mid \textit{Termo} / \textit{Fator} \mid \\
&\qquad \textit{Termo} \,\%\, \textit{Fator} \mid \textit{Fator} \\
\textit{Fator} &\rightarrow \textit{Primário} ** \textit{Fator} \mid \textit{Primário} \\
\textit{Primário} &\rightarrow 0 \mid \ldots \mid 9 \mid (\textit{Expressão})
\end{aligned}
$$

A precedência é determinada pelo comprimento da derivação mais curta a partir do símbolo inicial até o operador, enquanto a associatividade à esquerda ou à direita é determinada pelo uso de recursão à esquerda ou à direita, respectivamente. Essas propriedades são resumidas para a gramática G_1 na Tabela 2.1.

2. Linguagens do estilo de C, incluindo C++ e Java, não têm um operador de exponenciação. Embora o sinal de intercalação fosse uma escolha natural, já está usado para o *ou exclusivo de bit* nessas linguagens. Assim, adotamos o operador de exponenciação de Fortran (**).

Tabela 2.1 Associatividade e Precedência para a Gramática G_1

Precedência	Associatividade	Operadores
3	Direita	**
2	Esquerda	* / %
1	Esquerda	+ -

Considere a árvore de análise da Figura 2.3 para 4 ** 2 ** 3 + 5 * 6 + 7 que usa a gramática G_1. Esta árvore de análise interpreta a expressão como se tivesse sido escrita

((4 ** (2 ** 3)) + (5 * 6)) + 7.

A árvore de análise demonstra claramente que a exponenciação associa da direita para a esquerda. Embora não mostrado neste exemplo, a gramática G_1 permite que os operadores de multiplicação e divisão se associem da esquerda para a direita. Esta árvore também mostra a precedência de operadores da exponenciação *versus* adição e da adição *versus* multiplicação. Para a gramática G_1, os leitores devem se convencer que esta é a única árvore de análise que pode ser derivada da *string* 4 ** 2 ** 3 + 5 * 6 + 7. Isso pode ser provado experimentando-se diferentes estratégias e mostrando que nenhuma outra árvore com a raiz *Expressão* e folhas 4 ** 2 ** 3 + 5 * 6 + 7 pode ser derivada usando G_1.

Entender a relação entre a estrutura da gramática e a associatividade e a precedência de seus operadores é importante para projetistas de linguagem. Nem todas as linguagens seguem os mesmos princípios quanto a isso. Por exemplo, todos os operadores aritméticos de Smalltalk [Goldberg e Robson, 1989] associam estritamente da esquerda para a direita, sem qualquer precedência, enquanto os operadores APL [Iverson, 1962] associam da direita para a esquerda, sem qualquer precedência.

2.1.5 Gramáticas Ambíguas

Ao projetar uma gramática, é importante evitar fazer especificações ambíguas, ou seja, que possam ser interpretadas de duas ou mais formas diferentes.

> **Definição:** Uma gramática é *ambígua* se sua linguagem contiver pelo menos uma *string* com duas ou mais árvores de análise.

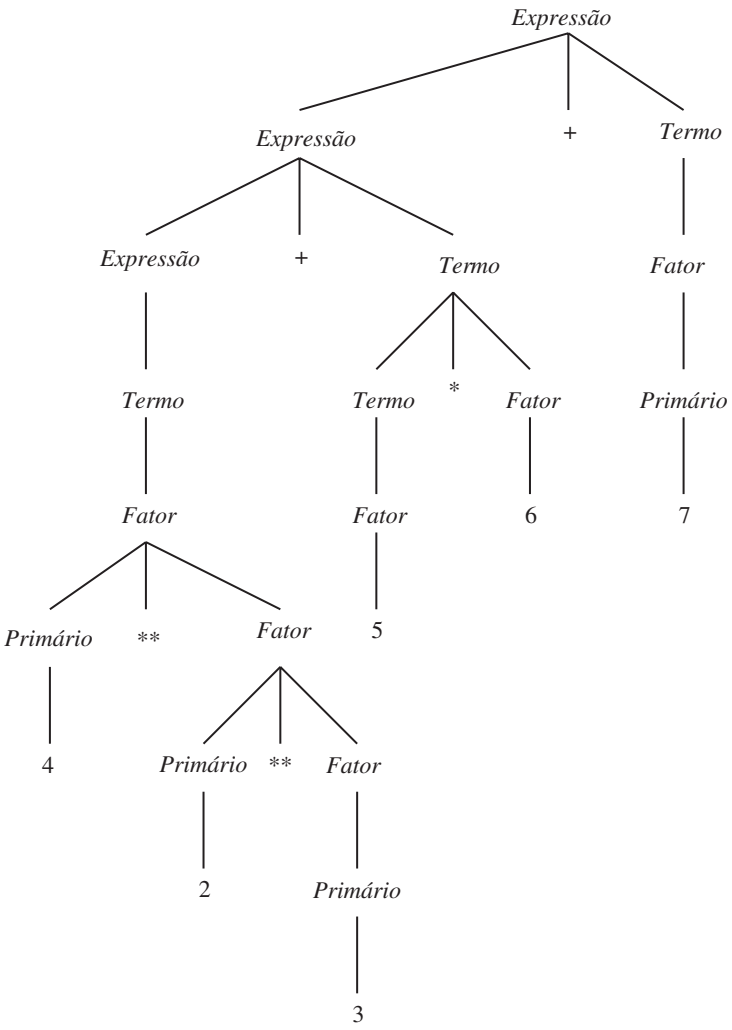
Normalmente, é preferível uma gramática não ambígua porque só deve haver uma interpretação válida para cada *string* em cada linguagem. Entretanto, a ambigüidade pode ser tolerável em alguns casos, especialmente quando sua introdução reduz significativamente o número de regras da gramática, ou seja, quando há um balanceamento entre o tamanho da gramática e as informações que ela está tentando expressar.

Analise a gramática G_2, que é uma versão ambígua de G_1:

Expressão → *Expressão Operador Expressão* | (*Expressão*) | *Inteiro*
Operador → + | − | * | / | % | **

A ambigüidade dessa gramática pode ser eliminada usando-se a Tabela 2.1.

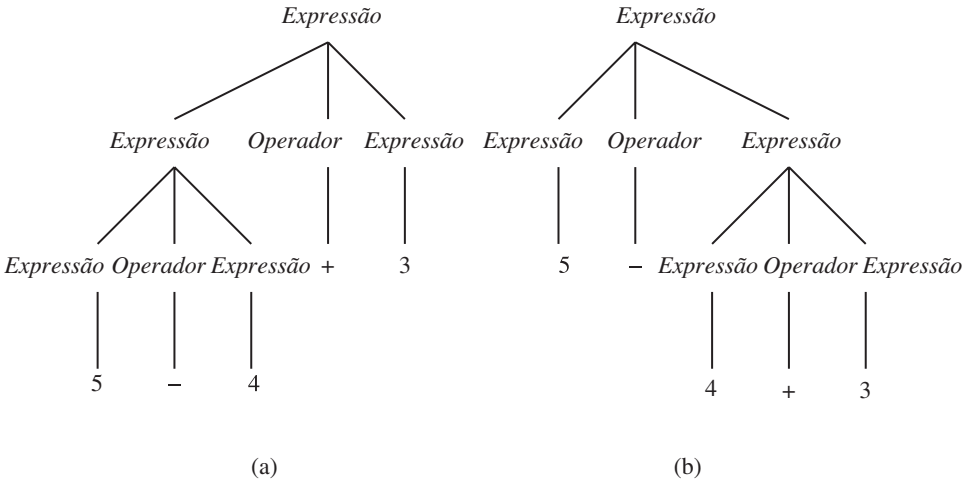
Observe que tal gramática se desenvolve apenas em uma nova alternativa quando cada novo operador é adicionado. Para linguagens como C, C++ e Java, o número de níveis de precedência e operadores torna suas gramáticas bastante grandes. Uma alternativa à inserção de precedência na gramática é usar uma gramática ambígua e então especificar a precedência e a associatividade separadamente, por exemplo em uma tabela (veja a Tabela 2.1).



| Figura 2.3 Análise de 4 ** 2 ** 3 + 5 * 6 + 7 para a Gramática G_1

O número de regras da gramática é ligeiramente maior para linguagens como C++ porque elas contêm operadores de prefixos e sufixos unários e diversos tipos diferentes de operandos. Entretanto, uma gramática completamente não ambígua para expressões C++ teria um não-terminal distinto para cada nível de precedência, aumentando assim demasiadamente seu tamanho e sua complexidade.

Para mostrar que G_2 por si só é ambígua, só precisamos encontrar alguma *string* que tenha duas árvores de análise distintas. Observando que a linguagem da gramática G_0 é um subconjunto da linguagem de G_2, usamos o exemplo $5 - 4 + 3$ na Figura 2.4 para mostrar duas árvores de análise distintas. Usando a interpretação natural de cada uma, a árvore da esquerda é avaliada como 4, enquanto a árvore da direita é avaliada como -2.



| Figura 2.4 Análise Ambígua de 5 − 4 + 3

Outro exemplo de ambigüidade sintática é conhecido como o problema do *senão pendente*. Analise o seguinte segmento de gramática G_{se}:

ComandoSe → `se` (*Expressão*) *Comando* |
`se` (*Expressão*) *Comando* `senão` *Comando*
Comando → *Atribuição* | *ComandoSe* | *Bloco*
Bloco → { *Comandos* }
Comandos → *Comandos Comando* | *Comando*

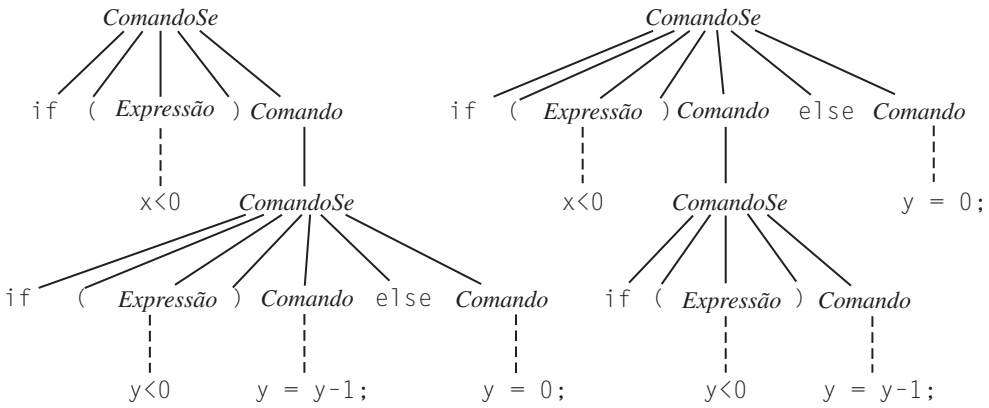
Não é difícil ver que essas regras específicas produzem o senão pendente, já que elas permitem duas ligações diferentes de uma cláusula `senão` a um `se` aninhado. Analise o seguinte exemplo:

```
se (x<0)
   se (y<0) y = y - 1;
   senão y = 0;
```

A gramática G_{se} permite que a cláusula `senão` seja ligada a qualquer um dos comandos `se`. Estas árvores de análise alternativas estão esboçadas na Figura 2.5, na qual a linha quebrada representa uma árvore omitida.

A questão aqui é decidir a qual `se` o `senão` deve ser ligado. Se a análise ligar a cláusula `senão` ao segundo comando `se`, `y` se tornará 0 sempre que `x<0` e `y<=0`. Todavia, se a análise atribuir a cláusula `senão` ao primeiro comando `se`, `y` se tornará 0 sempre que `x>=0`.

Algumas especificações de linguagem, como C e C++ (Stroustrup, 1997), resolvem esta ambigüidade dando uma descrição informal em língua inglesa sobre como a ligação deve ser feita. Por exemplo, tal descrição declararia que cada cláusula `senão` é associada ao comando `se` precedente textualmente mais próximo. Se uma ligação diferente for desejada, o programador deve inserir chaves { }. Por exemplo, para forçar



| Figura 2.5 A Ambigüidade do *Senão Pendente*

a ligação da cláusula `senão` com o primeiro `se` (mais externo) no exemplo anterior, o programador inseriria chaves da seguinte forma:

```
se (x<0)
   { se (y<0) y = y - 1 ; }
senão y = 0;
```

Agora a segunda linha identifica um *Bloco*, que é especificado na gramática G_{se} como uma lista de um ou mais comandos dentro de chaves. É claro que um *ComandoSe* é um comando perfeitamente bom para colocar entre chaves.[3]

Java resolve o problema do senão pendente expandindo a gramática BNF para um *ComandoSe* de uma forma interessante. A especificação Java (Gosling et al., 1996) separa a definição em duas categorias sintáticas diferentes, *ComandoSeEntão* e *ComandoSeEntãoSenão*, cada uma das quais é uma subcategoria da categoria geral *Comando*:

$$\begin{aligned} \textit{ComandoSeEntão} &\rightarrow \texttt{se}\ (\textit{Expressão})\ \textit{Comando} \\ \textit{ComandoSeEntãoSenão} &\rightarrow \texttt{se}\ (\textit{Expressão})\ \textit{ComandoSeNãoCurto} \\ &\qquad\qquad \texttt{senão}\ \textit{Comando} \end{aligned}$$

Observe que a segunda destas regras requer um *ComandoSeNãoCurto* antes da sua parte `senão`. Um *ComandoSeNãoCurto* inclui todas as declarações que não sejam *DeclaraçõesSeEntão*, mas pode incluir outras *DeclaraçõesSeEntãoSenão*. Essa condição elimina um *ComandoSeEntão* como a primeira alternativa antes de uma cláusula `senão`, eliminando assim qualquer ambigüidade sobre como associar a cláusula `senão` quando ela ocorrer em um programa. Se houver outro comando `se` aninhado neste *ComandoSeEntãoSenão*, ele deve ter uma cláusula `senão` explícita[4].

3. Este projeto de sintaxe da declaração `se` teve início com Algol-60 [Naur (ed.), 1963], que usava as palavras reservadas `begin` para a chave esquerda e `end` para a direita. Pascal também segue essa convenção. Linguagens de estilo C, começando com C, usam a chave esquerda para `begin` e a chave direita para `end`.

4. Esta estratégia pode ser consideravelmente mais complicada do que o resumo acima sugeriria. Por exemplo, cada comando de laço derivável de *ComandoSeNãoCurto* deve ter um não-terminal especial que restrinja um corpo de laço para u*m ComandoSeNãoCurto.*

Uma terceira abordagem ao problema do senão pendente se originou com Algol-68 [van Wijngaarden (ed.) *et al.*, 1969], na qual construções naturalmente aninhadas como comandos "se", laços e funções/procedimentos terminam cada uma com uma palavra-chave única. A convenção em Algol-68 era que a palavra-chave que servisse como término fosse a palavra-chave da abertura escrita ao contrário. Assim, o problema do senão pendente foi resolvido porque cada comando `se`, com ou sem uma cláusula `senão`, tinha que ser terminado com a palavra-chave `se`. Por exemplo, a ambigüidade mostrada na Figura 2.5 seria resolvida pelo programador Algol-68 de uma das duas formas a seguir, respectivamente:

```
se (x<0)
   se (y<0)
      y := y-1;
   senão  y := 0;
   es
es
se (x<0)
   se (y<0)
      y := y-1;
   es
senão y := 0;
es
```

Algumas linguagens recentes adotaram alguma forma dessa convenção, incluindo *shell* Bourne [Kernighan e Pike, 1984], Modula [Wirth, 1982], SR [Andrews e Olsson, 1993] e Ada [Cohen, 1996].

2.2 BNF ESTENDIDA

Desde Algol 60, alguma forma de BNF tem feito o trabalho pesado de descrever a sintaxe de linguagens de programação. Diversas variações menores foram introduzidas, mas não afetaram o poder expressivo básico da BNF. Essas variações foram introduzidas principalmente para melhorar a clareza e a concisão das descrições de sintaxe.

A *BNF Estendida* (abreviada EBNF) simplifica a escrita de uma gramática introduzindo metassímbolos para iteração, opção e escolha. Considere a gramática BNF G_0 da Seção 2.1.3, por exemplo:

$$\textit{Expressão} \rightarrow \textit{Expressão} + \textit{Termo} \mid \textit{Expressão} - \textit{Termo} \mid \textit{Termo}$$
$$\textit{Termo} \rightarrow 0 \mid ... \mid 9 \mid (\textit{Expressão})$$

A primeira dessas regras define uma *Expressão* como uma série (ou lista) de um ou mais *Termos* separados por sinais + ou −. A EBNF permite a eliminação de recursão à esquerda nesta regra da seguinte maneira:

$$\textit{Expressão} \rightarrow \textit{Termo} \{ (+ \mid -) \textit{Termo} \}$$

Nessa regra tanto as chaves quanto os parênteses são metassímbolos. As chaves { } denotam zero ou mais ocorrências dos símbolos que estão dentro delas. Os parênteses ( ) envolvem uma série de alternativas das quais uma deve ser escolhida. Colchetes [ ],

quando usados como metassímbolos EBNF, especificam uma seqüência opcional de símbolos (colchetes não são usados neste exemplo).

Uma complicação adicional com EBNF é a distinção do uso de chaves, parênteses e colchetes como metassímbolos do seu uso como símbolos terminais comuns. Wirth (1977) propôs envolver todos os símbolos terminais entre aspas (“ ”). Uma convenção mais moderna é que todos os símbolos terminais sejam escritos com uma fonte especial. Aqui, convencionamos usar uma fonte de `largura fixa` (por exemplo, os símbolos (,), + e − na gramática anterior) e os metassímbolos (outros que não a seta para a direita), em negrito.

Um exemplo do uso de colchetes como metassímbolo ocorre na versão EBNF da gramática G_{se} da Seção 2.1.5:

ComandoSe → `se` (*Expressão*) *Comando* **[** `senão` *Comando* **]**

Esta regra deve ser lida como se um *ComandoSe* fosse definido para ser um `se` terminal, seguido por uma *Expressão* entre parênteses, seguida por um *Comando*, seguido por uma parte `senão` opcional entre parênteses consistindo de um `senão` terminal seguido por um *Comando*. Como sua contraparte na Seção 2.1.5, esta gramática é ambígua.

Preferimos a notação EBNF, que muitas vezes é chamada de *EBNF estilo-Wirth*, para descrever e ilustrar sintaxe de linguagem no restante deste livro. Essa preferência é guiada pelo fato de que as definições de sintaxe de linguagem em EBNF tendem a ser ligeiramente mais claras e mais breves do que as definições BNF. Em especial, a EBNF elimina o uso desnecessário de definições recursivas para descrever listas, que são fundamentais na sintaxe de linguagens de programação. Existem listas de declarações, listas de comandos, listas de parâmetros, e assim por diante.

Uma variante da EBNF, que chamamos de *EBNF estilo C*, é usada por Kernighan e Ritchie (1988) na definição de ANSI C, por Stroustrup (1977) na definição de C++ e por Gosling et al. na definição de Java. Uma descrição desta notação é fornecida por Stroustrup (1991, p. 478):

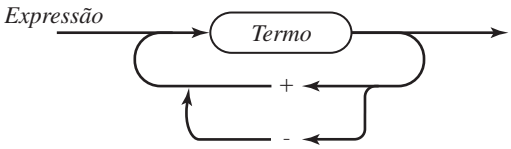
> Na sintaxe usada neste manual, as categorias sintáticas são indicadas pelo tipo *itálico* e palavras literais e caracteres em tipo de `largura constante`. As alternativas são listadas em linhas separadas, exceto em alguns casos em que um conjunto grande de alternativas é apresentado em uma linha, marcado pela expressão “um de”. Um não-terminal ou terminal opcional é indicado por “opc”...

O subscrito “opc” corresponde aos nossos metacolchetes, porém sem a capacidade de agrupar uma seqüência opcional de diversos símbolos terminais e não-terminais. Assim, a gramática G_{se} seria escrita na notação que se segue:

comandose :
`se` (*expressão) comando partesenão*$_{opc}$

partesenão :
`senão` *comando*

Outra variação da EBNF é o *diagrama de sintaxe*, que foi popularizado pelo seu uso na definição sintática da linguagem Pascal (Jensen e Wirth, 1975). Diagramas de sintaxe podem auxiliar a esclarecer a sintaxe de diversas construções de linguagens quando estas são ensinadas a programadores iniciantes. Eles foram amplamente usados em livros-textos de Pascal



| Figura 2.6 Diagrama de Sintaxe para *Expressões* com Adição e Subtração

escritos para cursos introdutórios de programação; veja, por exemplo, (Cooper e Clancy, 1985). A Figura 2.6 mostra um exemplo de diagrama de sintaxe para a idéia de *Expressão* com qualquer número de operadores adicionais.

O significado desse diagrama de sintaxe é auto-explicativo de acordo com seu formato. Lemos o diagrama da esquerda para a direita, e todos os caminhos que levam para a seta mais à direita definem *Expressões* legais, ou seja, uma *Expressão* consiste de um *Termo* seguido por uma seqüência de zero ou mais ocorrências de + junto a um *Termo*. Em outras palavras, uma *Expressão* é uma seqüência ou uma lista de um ou mais *termos* separados por símbolos +.

Nem a EBNF nem o diagrama de sintaxe são mais poderosos do que a BNF para descrever formalmente uma sintaxe de linguagem. Podemos demonstrar isso facilmente para a EBNF estilo Wirth da seguinte forma. Suponha que A seja um não-terminal e x, y, z sejam seqüências arbitrárias de terminais e não-terminais. Qualquer regra EBNF que tenha metachaves:

$$A \rightarrow x\{y\}z$$

pode ser reescrita de forma equivalente sem metachaves da seguinte forma:

$$A \rightarrow x\,A'\,z$$
$$A' \rightarrow \in \mid y\,A'$$

em que A' é um novo não-terminal único. Observe que a primeira alternativa na regra para A' é uma *string* vazia $\in$, o que significa que A' pode ser substituído por nada. As estratégias para a eliminação de metachaves (opção) e metaparênteses (grupo alternativo) das regras EBNF são semelhantes e deixadas como exercício.

Assim, a alternativa EBNF é, muitas vezes, usada na descrição da sintaxe de uma linguagem porque produz regras mais simples do que a BNF. Demonstramos isso na Seção 2.3 descrevendo a sintaxe de uma linguagem simples que é um subconjunto de C/C++.

2.3 A SINTAXE DE UMA PEQUENA LINGUAGEM: CLITE

As idéias discutidas neste capítulo fornecem uma base para a descrição da sintaxe de uma linguagem de programação completa. Entretanto, para evitar o excesso de detalhes, decidimos definir um subconjunto de C, uma linguagem que chamamos de *Clite*. Esta linguagem deve ser familiar a qualquer pessoa que tenha escrito um programa em C, C++ ou Java.

Uma gramática completa para Clite, denominada G_{Clite}, é mostrada na Figura 2.7. Essa gramática é muito menor do que as de diversas linguagens reais, conforme mostrado na Tabela 2.2.

Nesta discussão, não imitaremos o estilo comumente usado para definir uma linguagem (por exemplo, Jensen e Wirth, 1975), em que cada construção importante é introduzida primeiramente apresentando-se sua sintaxe, depois sua semântica de tempo de compilação e, finalmente, sua semântica de tempo de execução.

Programa → `int main ( )` { *Declarações Comandos*}
Declarações → { *Declaração* }
Declaração → *Tipo Identificador* **[** [*Inteiro*]] **{ ,** *Identificador* **[** [*Inteiro*]] **}** ;
Tipo → `int` | `bool` | `float` | `char`
Comandos → **{** *Comando* **}**
Comando → ; | *Bloco* | *Atribuição* | *ComandoSe* | *ComandoEnquanto*
Bloco → {*Comandos*}
Atribuição → *Identificador* **[** [*Expressão*]] **=** *Expressão* ;
ComandoSe → `se` (*Expressão*) *Comando* **[senão** *Comando***]**
ComandoEnquanto → `enquanto` (*Expressão*) *Comando*

Expressão → *Conjunção* **{** || *Conjunção* **}**
Conjunção → *Igualdade* **{** && *Igualdade* **}**
Igualdade → *Relação* **[** *OpIgual Relação* **]**
OpIgual → == | !=
Relação → *Adição* **[** *OpRel Adição* **]**
OpRel → < | <= | > | >=
Adição → *Termo* **{** *OpAdição Termo* **}**
OpAdição → + | -
Termo → *Fator* **{** *OpMult Fator* **}**
OpMult → * | / | %
Fator → **[** *OpUn* **]** *Primário*
OpUn → - | !
Primário → *Identificador* **[** [*Expressão*]] | *Literal* | (*Expressão*)
| *Tipo* (*Expressão*)

Identificador → *Letra* **{** *Letra* | *Dígito* **}**
Letra → a | b | ... | z | A | B | ... | Z
Dígito → 0 | 1 | ... | 9
Literal → *Inteiro* | *Boleano* | *Ponto Flutuante* | *Caractere*
Inteiro → *Dígito* **{** *Dígito* **}**
Boleano → `verdadeiro` | `falso`
Ponto Flutuante → *Inteiro* . *Inteiro*
Caractere → ' *CaractereASCII* '

| Figura 2.7 A sintaxe de Clite

| Tabela 2.2 Tamanhos de Gramáticas de Diversas Linguagens

Linguagem	Páginas Aproximadas	Referência
Pascal	5	(Jensen e Wirth, 1975)
C	6	(Kernighan e Ritchie, 1988)
C++	22	(Stroustrup, 1997)
Java	14	(Gosling et al., 1996)

Em vez disso, nosso propósito aqui é examinar em maior profundidade a sintaxe, deixando a discussão a respeito da semântica e outras questões de projeto para capítulos posteriores.

Embora a gramática G_{Clite} realize um bom trabalho na apresentação da sintaxe de diversas construções, um bom número de questões sintáticas relacionadas não são abordadas:

1. Comentários.
2. O papel do espaço em branco.
3. A distinção entre símbolos como o $<=$ dos dois símbolos $<$ e $=$.
4. A distinção entre os identificadores e as palavras-chave como `se`.

O restante desta seção abordará estas e outras questões sintáticas.

A gramática G_{Clite} possui dois níveis: o léxico e o sintático. Como veremos na Seção 2.4, estes níveis correspondem a partes distintas de um compilador. Todas as questões listadas anteriormente são léxicas.

2.3.1 Sintaxe Léxica

As últimas oito regras de gramática da Figura 2.7, começando com a definição de um *Identificador*, abordam questões léxicas. Elas definem a sintaxe de um *Identificador* e diversas formas de *Literal*. Além disso, todas as palavras-chave e outros símbolos terminais nas primeiras 23 regras de produção de G_{Clite} também fazem parte do nível léxico.

O alfabeto de entrada da sintaxe léxica é o conjunto de caracteres ASCII que podem ser impressos, o que exclui o caractere NUL (código ASCII 0), os caracteres de controle (códigos ASCII 1-26), excetuando o caractere de tabulação (9), DEL (127) e todos os caracteres cujos códigos ultrapassarem 127. O conjunto de caracteres ASCII é descrito em detalhes na Seção 5.3. As *strings* terminais que podem ser derivadas na sintaxe léxica são chamadas de *tokens* e classificadas nos seguintes grupos:

- *Identificadores*
- *Literais,* incluindo *Inteiros*, `true` e `false`, *Pontos Flutuantes* e *Caracteres*
- Palavras-chave: `bool char else false float if int main true while`
- Operadores: `= || && == != < <= > >= + - * / % ! [ ]`
- Pontuação: `; , { } ( )`

Normalmente, os *tokens* não podem conter espaços em branco. A definição de *espaço em branco* comumente inclui os caracteres de espaço e tabulação, o carac-

tere ou os caracteres delimitando o final de linha e os comentários. Por exemplo, quando >= é digitado sem um espaço em branco, é tratado como um *token*, mas o texto > = constitui dois *tokens*. De forma semelhante, analise o seguinte fragmento de código de programa em Pascal:

```
while a < b do
```

Aqui, o espaço em branco é necessário entre a palavra-chave `while` e o identificador `a` e também entre `b` e `do`. O espaço em branco não é necessário para separar `a` de `<` ou `<` de `b`.

Os *tokens* são analisados pelos compiladores com o uso de um algoritmo ávido que procura pela maior seqüência de caracteres que constituam um *token* válido independente do seu contexto. No exemplo anterior, sem o espaço em branco após o `while`, os caracteres `whilea` formariam um *token* válido do tipo *Identificador*, embora tal identificador possa não ter sido declarado e `whilea <` não constitua um prefixo de comando válido. De forma oposta, a seqüência de caracteres `a<` não forma um *token* válido. Assim, espaços em branco entre estes dois caracteres são desnecessários.

Observe que a gramática G_{Clite} não define a sintaxe de um comentário, deixando-o para ser definido fora da gramática, como no caso das definições de Pascal (Jensen e Wirth, 1975) e C (Kernighan e Ritchie, 1988). Clite usa a convenção de comentários // de C++.

Um *Identificador* é constituído de letras (A–Z, a–z) e dígitos (0–9), e começa com uma letra. Assim, por essa definição, a *string se* é tanto um *Identificador* quanto uma *palavra-chave* (veja a definição de *ComandoSe*). Como, então, os identificadores são distinguidos das palavras-chave? Na maioria das linguagens, todas as palavras-chave são as assim chamadas *palavras reservadas*, no sentido de que nenhum identificador pode ser escrito da mesma forma que uma palavra-chave. Clite segue esta convenção e suas palavras reservadas são:

```
bool   else      float   int     true
char   false     if      main    while
```

Observe que `main` não é reservada em C ou C++. Em vez disso, é um identificador especial que nomeia a função na qual a execução do programa começa.

A gramática de Clite na Figura 2.7 define quatro tipos de literais. Literais *Inteiro* são convencionais, consistindo de uma seqüência de um ou mais dígitos decimais. Algumas linguagens permitem que literais inteiros sejam expressos em outras bases além de 10, usando uma diversidade de convenções.

Um literal de ponto flutuante possui uma seqüência de dígitos com um ponto decimal interno. Pelo menos um dígito é requerido antes e depois do ponto decimal. Esta convenção foi adotada pelos autores há muitos anos, ao aprenderem que o número `5.` era permitido em Fortran, mas o número `.5` não era. A convenção oposta era verdadeira em Algol. Todavia, os números `5.0` e `0.5` são permitidos em ambas as linguagens. A maioria das linguagens fornece números de ponto flutuante, tanto em notação científica quanto com precisão simples ou dupla.

A gramática Clite também não define a faixa de valores válidos para números inteiros ou de ponto flutuante e não diz como eles são armazenados. Estas são questões de tipo e semântica, que discutimos com mais detalhes nos Capítulos 5–8.

Os literais *Boleanos* de Clite são as palavras reservadas `true` e `false`. Literais do tipo *Caractere* são compostos de um único caractere ASCII imprimível (veja a Seção 5.1) entre aspas simples ('), como '`a`' e '`b`'.

2.3.2 Sintaxe Concreta

A estrutura sintática de um programa completo é uma árvore derivada da seqüência de *tokens* do programa e da sua gramática. A *sintaxe concreta* de um programa é a árvore que representa uma análise da sua seqüência de *tokens*, começando com o símbolo inicial da sua gramática. Por exemplo, a raiz de qualquer árvore de análise de um programa em Clite é a categoria sintática *Programa*, tendo como descendentes diretos uma série de *tokens* incluindo `main`, uma subárvore de *Declarações* e uma subárvore de *Comandos*.

As linguagens de estilo C usam ponto-e-vírgula ; como terminador de declarações e comandos, em vez de um separador de comandos como em Algol e Pascal. Apenas o comando *Bloco* não termina com um ponto-e-vírgula, mas, sim, com uma chave.

Uma *Declaração* em Clite consiste de um tipo seguido de uma lista de identificadores separados por vírgulas. Os tipos internos são: `int`, `float`, `bool` e `char`. Observe que este estilo de declaração é o oposto da convenção em Pascal, Ada e UML, nos quais a lista de identificadores precede o tipo. Aqui estão dois exemplos:

```
int i, j;
float x;
```

Comandos em Clite incluem *Atribuição* usando = como operador de atribuição, *Bloco* (uma série de comandos entre chaves { }), *DeclaraçãoSe* e *DeclaraçãoEnquanto*. Observe que a regra para *DeclaraçãoSe* torna a gramática ambígua, sofrendo do problema do senão pendente. Isto é resolvido da forma habitual (Stroustrup, 1991):

> A ambigüidade do `senão` é resolvida conectando-se um `senão` ao último `se` sem o `senão` correspondente que for encontrado.

Essa é a única ambigüidade na gramática de Clite da Figura 2.7.

A sintaxe concreta de Clite permite expressões sobre diversos operadores. O uso de metachaves { } em diversos níveis de precedência, como na regra da *Adição*, é feito para inferir associatividade à esquerda. Observe que o uso de metachaves (inferindo opcional) é usado para definir tanto *Igualdade* quanto *Relação*, tornando os operadores *OpIgual* e *OpRel* não-associativos. Metaparênteses não são usados na gramática Clite da Figura 2.7.

Observe que a seção intermediária da Figura 2.7 é dedicada à definição não-ambígua de sete níveis de precedência, seus operadores associados e a associatividade. Para C++ completo, quatro páginas de Stroustrup, 1977, são dedicadas a definir a sintaxe concreta de expressões. Isso resulta do grande número de operadores e dos níveis de precedência. Em contraste, a definição de C (Kernighan e Ritchie, 1988) usa uma gramática de expressão ambígua, deixando a precedência e a associatividade de operadores para serem especificadas fora da gramática.

Mesmo quando a subgramática da expressão não é ambígua, a compreensão é melhorada especificando-se a precedência e a associatividade de operadores de expressão em uma tabela, que para Clite é dada na Tabela 2.3, da maior para a menor precedência.

Essa especificação difere de C/C++ de duas formas importantes. Primeiro, Clite possui muito menos operadores e níveis de precedência resultantes. Para ver uma comparação de C, C++ e Java, observe a Tabela 2.4. Na tabela, os níveis de precedência são mostrados do mais alto para o mais baixo, separados do nível seguinte por uma linha horizontal.

Tabela 2.3

Precedência de Operadores e Associatividade em Clite

Operadores	Associatividade
Unário – !	nenhuma
* / %	à esquerda
+ –	à esquerda
< <= > >=	nenhuma
== !=	nenhuma
&&	à esquerda
||	à esquerda

Em segundo lugar, nem os operadores de igualdade nem os relacionais são associativos, o que é uma idéia emprestada de Ada (The Department of Defense, 1983). Em C/C++ você pode escrever expressões como:

```
se (a < x < b)
```

que presumivelmente está tentando testar se `x` possui um valor estritamente entre `a` e `b`. Entretanto, enquanto `b > 1`, a expressão será avaliada como `true`. Isso resulta do fato de que `a < x` é avaliado como 0 ou 1. Assim, tornar esses operadores associativos é uma decisão de projeto questionável.

Nada na gramática de Clite especifica os tipos de operandos requeridos por cada operador ou o tipo do resultado retornado. Estas são questões de tipo e semântica não abordadas pela sintaxe concreta e são discutidas mais integralmente nos Capítulos 5–8.

2.4 COMPILADORES E INTERPRETADORES

A estrutura lógica de um compilador é mostrada na Figura 2.8. Cada caixa na figura representa uma *fase*, que é uma operação logicamente coesa que transforma o programa-fonte de uma representação para outra, conforme explicado a seguir. Em contraste está a noção de um *passo*, que é uma leitura completa da representação corrente do programa-fonte. Muitos compiladores de um passo foram escritos; em contraste, muitos compiladores PL/I usavam múltiplos passos na fase semântica. Compiladores de otimização geralmente são de múltiplos passos.

Um programa-fonte começa como um fluxo de caracteres. A função do *analisador léxico* (ou *lexer*) é varrer o programa e transformá-lo em um fluxo de *tokens*. Como parte dessa transformação, todos os espaços em branco, incluindo comentários, são descartados. De forma semelhante, todas as seqüências que não formarem *tokens* válidos são descartadas, após ser gerada uma mensagem de erro.

Já que as fases de análise léxica e sintática são baseadas em uma gramática, os motivos para se ter uma fase de análise léxica separada precisam ser explicados. Historicamente, estes motivos são:

1. O projeto do analisador léxico é fundamentado em um modelo de máquina mais rápido e muito mais simples do que o modelo livre de contexto usado para análises sintáticas.
2. Para um compilador que não seja de otimização, aproximadamente 75% do tempo total de compilação, muitas vezes, é consumido na análise léxica. Qualquer melhoria que faça com que esta fase seja executada mais rapidamente possui um impacto considerável sobre o tempo total de compilação.

| **Tabela 2.4.** Precedência e Operadores em C, C++ e Java

Operação		**Operadores**		
(precedência descendente)	**Associatividade**	**C**	**C++**	**Java**
Resolução de escopo	esquerda		`::`	
Seleção de membro	esquerda	`.`	`.`	`.`
Seleção de membro ponteiro	esquerda	`->`	`->`	
Subscrito	esquerda	`[]`	`[]`	`[]`
Chamada de função	esquerda	`()`	`()`	`()`
Construção de valor	esquerda	`()`	`()`	`()`
Pós-incremento	esquerda	`++`	`++`	`++`
Pós-decremento	esquerda	`--`	`--`	`--`
Tamanho do objeto	direita	`sizeof`	`sizeof`	
Tamanho do tipo	direita	`sizeof`	`sizeof`	
Pré-incremento	direita	`++`	`++`	`++`
Pré-decremento	direita	`--`	`--`	`--`
Complemento	direita	`~`	`~`	`~`
Menos unário	direita	`-`	`-`	`-`
Mais unário	direita	`+`	`+`	`+`
Endereço de	direita	`&`	`&`	
Desreferência	direita	`*`	`*`	
Aloca	direita		`new`	`new`
Desaloca	direita		`delete`	
Desaloca matriz	direita		`delete[]`	
Conversão	direita	`()`	`()`	`()`
Seleção de membro	esquerda	`.*`	`.*`	
Seleção de membro ponteiro	esquerda	`->*`	`->*`	
Multiplica	esquerda	`*`	`*`	`*`
Divide	esquerda	`/`	`/`	`/`
Módulo	esquerda	`%`	`%`	`%`
Adição	esquerda	`+`	`+`	`+`
Subtração	esquerda	`-`	`-`	`-`
Shift para esquerda	esquerda	`<<`	`<<`	`<<`
Shift para direita	esquerda	`>>`	`>>`	`>>`
Menor que	esquerda	`<`	`<`	`<`
Menor ou igual a	esquerda	`<=`	`<=`	`<=`
Maior que	esquerda	`>`	`>`	`>`
Maior ou igual a	esquerda	`>=`	`>=`	`>=`

| **Tabela 2.4.** Precedência e Operadores em C, C++ e Java

Operação		**Operadores**		
(precedência descendente)	**Associatividade**	**C**	**C++**	**Java**
Igual	esquerda	==	==	==
Não igual	esquerda	!=	!=	!=
"E" de bit	esquerda	&	&	&
"OU" exclusivo de bit	esquerda	^	^	^
"OU" inclusivo de bit	esquerda	|	|	|
"E" lógico	esquerda	&&	&&	&&
"OU" inclusivo lógico	esquerda	||	||	||
Atribuição condicional	esquerda	? :	? :	? :
Atribuição	direita	=	=	=
Atribuição com multiplicação	direita	*=	*=	*=
Atribuição com divisão	direita	/=	/=	/=
Atribuição com módulo	direita	%=	%=	%=
Atribuição com adição	direita	+=	+=	+=
Atribuição com subtração	direita	-=	-=	-=
Atribuição com shift para esquerda	direita	<<=	<<=	<<=
Atribuição com shift para direita	direita	>>=	>>=	>>=
Atribuição com "E"	direita	&=	&=	&=
Atribuição com "OU" inclusivo	direita	|=	|=	|=
Atribuição com "OU" exclusivo	direita	^=	^=	^=
Gera exceção	esquerda		throw	throw
Seqüenciamento de vírgulas	esquerda	,	,	,

3 Antes da criação de ASCII, cada computador tinha seus próprios caracteres idiossincráticos. Esconder esse conjunto de caracteres no analisador léxico tornou mais fácil portar um compilador para uma nova máquina. Mesmo hoje ainda existem vestígios desse problema, com os conjuntos de caracteres ASCII, Unicode e EBCDIC sendo usados.

4 A convenção de fim de linha também varia de um sistema operacional para outro. Por exemplo, o Apple Macintosh (antes do Mac OS/X) usa o retorno de carro (ASCII 13) como um caractere de fim de linha, Linux/Unix usa um caractere de nova (ou alimentação de) linha (ASCII 10), e o Microsoft Windows usa uma combinação retorno de carro/nova linha. Para muitas linguagens modernas, mas não todas, este problema é manipulado pela biblioteca de suporte de tempo de execução da linguagem. Antigamente isso não ocorria, de modo que essa dependência do sistema operacional foi escondida dentro do analisador léxico.

A segunda fase de um compilador é o *analisador*, ou fase da *análise sintática*. O *analisador sintático* lê um fluxo de *tokens* e constrói uma árvore de análise, de acordo com

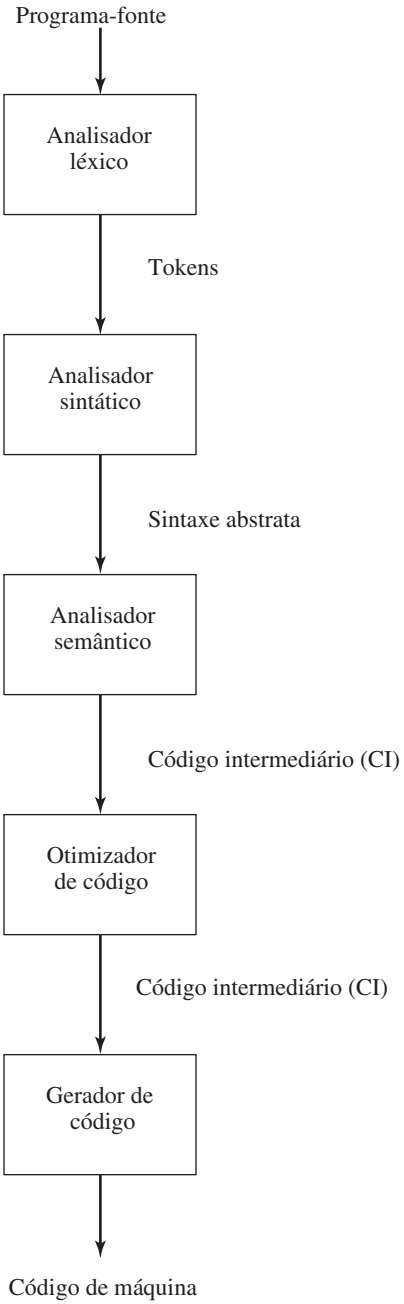
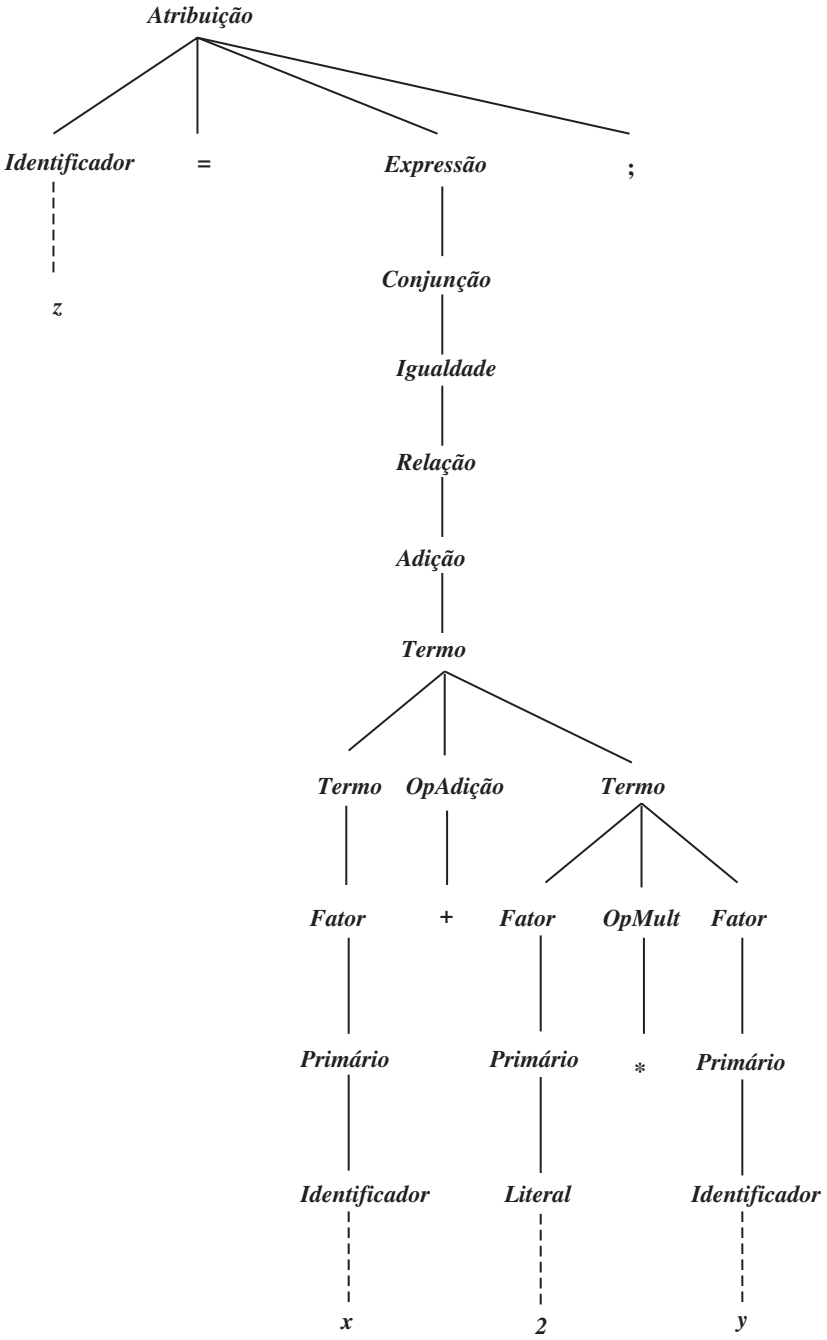


Figura 2.8 Principais Etapas no Processo de Compilação

as regras da gramática. Um analisador completo é desenvolvido para Clite na Seção 3.3. A saída de um analisador pode ser uma árvore de análise, conforme mostrado na Figura 2.9. De forma alternativa, a saída pode ser uma árvore de sintaxe abstrata, cujas vantagens serão discutidas mais integralmente na Seção 2.5.



| Figura 2.9 Árvore de Análise para `z = x + 2 * y;`

Como exemplo de análise, considere a análise da entrada `z = x + 2 * y;` , com o objetivo de obter uma árvore de análise para *Atribuição,* usando a gramática G_{Clite}. A regra da gramática *Atribuição* requer um *Identificador* seguido de um =, uma *Expressão,* e ;. Para que essa *string* seja analisada com sucesso, `z` deve ser uma instância da categoria gramatical *Identificador* e `x + 2 * y` deve ser uma instância da categoria *Expressão*. Assim, a subárvore com raiz em *Expressão* na Figura 2.9 deve ser desenvolvida como uma tarefa separada, usando diferentes regras de gramática, ou seja, regras adicionais para *Termo, Fator* e *Primário* entram em ação quando estas subárvores são desenvolvidas.

A terceira fase do compilador é a fase da *análise semântica* ou *verificador de tipo*. Esta fase é responsável por garantir que as regras de semântica de tempo de compilação da linguagem sejam impostas com as seguintes condições:

- Que todos os identificadores referenciados no programa sejam declarados.
- Que os operandos para cada operador possuam um tipo apropriado.
- Que as operações de conversões envolvidas, por exemplo, inteiro para float, sejam inseridas onde for necessário.

Se alguma regra de semântica de tempo de compilação for violada, uma mensagem de erro apropriada será gerada. A saída dessa fase é uma árvore de código intermediário (CI), que é basicamente uma árvore de sintaxe abstrata transformada em um nível mais baixo de abstração. Por exemplo, em Clite, o *token* + é usado para indicar a adição de inteiros e pontos flutuantes. Na árvore CI poderia haver operadores separados para cada tipo de adição. As principais questões de verificação de tipos, assim como o verificador de tipos de Clite, são discutidas com mais detalhes em um capítulo posterior.

O propósito da fase de *otimização de código* de um compilador é melhorar o código intermediário baseado na arquitetura da máquina-alvo. Melhorias comuns incluem encontrar expressões constantes e avaliá-las em tempo de compilação, reordenar código para melhorar o desempenho da cache, encontrar subexpressões comuns e substituí-las por uma referência a um temporário, e assim por diante. Um bom otimizador de código pode aumentar bastante a velocidade de execução de um programa.

A fase final do compilador é a fase de *geração de código*, na qual é elaborado o código específico para a máquina-alvo. Esta fase é responsável por decidir quais instruções de máquina usar, como alocar registradores e outros detalhes dependentes da máquina.

Em contraste com um compilador, um *interpretador* é uma forma de tradutor no qual as duas últimas fases do compilador são substituídas por um programa que executa diretamente (ou interpreta) o código intermediário. Há dois tipos gerais de interpretadores: puros e mistos. Em um *interpretador puro*, cada declaração é colocada em *tokens*, analisada, verificada semanticamente e interpretada cada vez que é executada. A maioria dos tradutores Basic e virtualmente todos os interpretadores de comandos de *shell*, como por exemplo o *shell* Bourne, são interpretadores puros. Em contraste, a maioria das linguagens de *scripting*, como por exemplo Perl, é traduzida por intermédio dos *interpretadores mistos*, que, a cada execução, primeiro traduzem todo o *script* (ou programa) em código intermediário e depois, repetidamente, interpretam esse código intermediário sem mais traduções.

O JDK Java da Sun é um exemplo incomum de um sistema interpretador misto no qual o compilador Java `javac` *compila* código-fonte em Java para uma forma externa de código intermediário, conhecido como um arquivo de *byte code,* que é interpretado com o uso de uma máquina virtual chamada de "Java Virtual Machine" (JVM). O interpretador de *byte code* Java é denominado `java` e simula o comportamento da JVM. Complicando mais a situação, nos últimos anos a Sun introduziu um gerador de código

"*on the fly*" para o interpretador que traduz partes freqüentemente executadas do programa em código de máquina nativo.

Capítulos posteriores usam Clite como base para discutir sistemas de tipo e semântica assim como para representar expressões em um exemplo recorrente sobre diferenciação simbólica. Este capítulo continua a discussão de sintaxe examinando a transição de sintaxe para semântica.

2.5 RELACIONANDO SINTAXE E SEMÂNTICA

Até agora, exploramos ferramentas para definir os níveis sintático e semântico de uma linguagem incluindo uma visão geral das fases de análise léxica e sintática de um compilador. Pareceria que a saída do analisador (fase de análise sintática) deveria ser uma árvore de análise. Nesta seção, analisamos a idéia de uma alternativa à arvore de análise para representar a estrutura e o conteúdo de um programa.

A necessidade de tal alternativa é múltipla. Examine novamente a árvore de análise dada na Figura 2.9. Já que muitos dos não-terminais nessa árvore não expressam informações, seria útil eliminá-los. Suponha que executemos as seguintes transformações nessa árvore de análise:

1. Descarte todos os símbolos terminais separadores (ou pontuação), como ponto e vírgula.
2. Descarte todos os não-terminais que sejam raízes triviais, ou seja, os que tenham apenas uma única subárvore. Um exemplo é o símbolo *Termo* que é o nó interior mais à esquerda derivável de *Expressão*.
3. Finalmente, substitua os não-terminais remanescentes por operadores que sejam uma folha de uma de suas subárvores imediatas. Um exemplo é substituir o símbolo *Atribuição* pelo operador =.

O que resulta é uma árvore como a mostrada na Figura 2.10, chamada de árvore de *sintaxe abstrata*. É útil comparar esta árvore com a árvore de análise original da Figura 2.9. A árvore de sintaxe abstrata contém menos nós intermediários e subárvores, embora tenha a mesma estrutura da expressão original. Assim, esta árvore contém todas as informações importantes da árvore de análise original, mas é muito menor. Esses ganhos seriam muito maiores durante a análise de um programa completo que usasse a gramática inteira para Clite.

Um segundo argumento para usar uma sintaxe abstrata é que ela pode revelar os elementos sintáticos essenciais em um programa sem especificar em excesso os detalhes de como eles são escritos pelo programador. Considere, por exemplo, os seguintes comandos de laço Pascal e C/C++:

Pascal

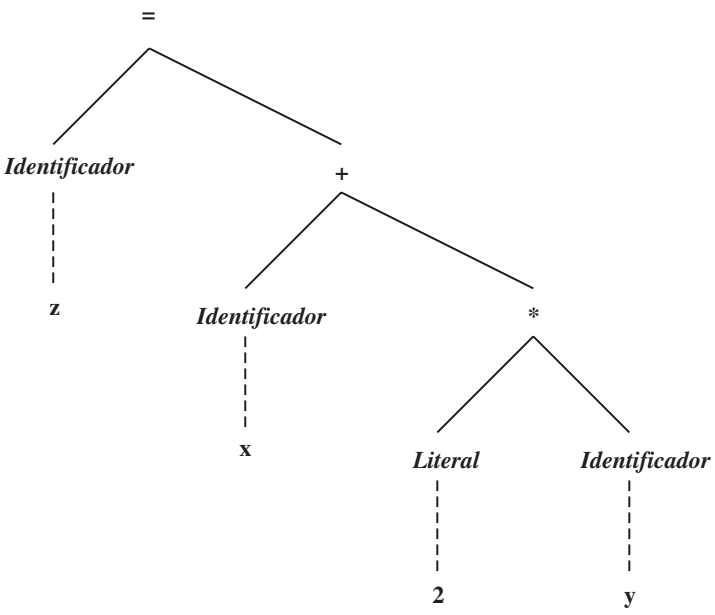
```
while i<n do begin
      i := i + 1
end
```

C/C++

```
while (i<n) {
      i = i + 1;
}
```

Estes dois laços obviamente obtêm o mesmo resultado; suas diferenças sintáticas não são essenciais para o processo fundamental do laço que elas representam.

Ao pensar abstratamente sobre um laço, descobrimos que os únicos elementos essenciais são uma expressão de *teste* (`i < n` neste caso) para continuar o laço e um comando



| Figura 2.10 Árvore de Sintaxe Abstrata para `z = x + 2 * y;`

que seja o *corpo* do laço a ser repetido (incrementando a variável `i`, neste caso). Todos os outros elementos que aparecem nesses dois comandos de laço constituem "*perfumaria" sintática* não essencial.

2.5.1 Sintaxe Abstrata

Assim, é valioso definir a estrutura de uma linguagem de programação em um nível mais funcional do que aquele que é oferecido pela sua sintaxe EBNF. *Sintaxe abstrata* é uma notação que permite a um analisador retirar a *perfumaria* sintática e gerar uma árvore que contenha apenas os elementos essenciais da computação.

A sintaxe abstrata de uma linguagem de programação pode ser definida, por meio de um conjunto de regras, da seguinte forma:

Lhs = *Rhs*

em que *Lhs* é o nome de uma classe de sintaxe abstrata e *Rhs* define a classe como:

1. Uma lista de uma ou mais alternativas. Por exemplo, na sintaxe concreta da gramática Clite da Figura 2.7, há apenas quatro tipos de *Expressão*: identificadores, literais ou valores, aquelas com um operador unário e aquelas com um operador binário.
2. Uma lista de componentes essenciais que definem um membro dessa classe, separados por ponto-e-vírgula (`;`). Cada componente desses possui a forma de uma declaração comum, identificando uma classe de sintaxe abstrata e uma lista de um ou mais nomes de campos, separados por vírgulas (`,`).

```
Atribuição = Variável target; Expressão source
Expressão = Variável | Valor | Binária | Unária
Binária = Operador op; Expressão term1, term2
Unária = Operador op; Expressão term
Variável = String id
Valor = Inteiro value
Operador = + | - | * | / | !
```

Figura 2.11 Sintaxe Abstrata Parcial para a Gramática da Figura 2.7

Uma classe de sintaxe abstrata, em ambas as alternativas, pode corresponder a um domínio matemático bem conhecido como *Inteiro, String* ou *Boleano*. O propósito dos nomes dos campos é permitir que o processamento semântico referencie componentes abstratos por nome.

Considere, por exemplo, as regras de sintaxe abstrata para *Atribuição* e *Expressão* que aparecem na Figura 2.7; uma sintaxe abstrata para esta gramática é demonstrada na Figura 2.11. A regra para *Expressão* é uma instância do primeiro tipo de regra de sintaxe abstrata, enquanto *Atribuição* é uma instância do segundo tipo de regra. A definição de uma expressão *Binária* indica a potencial utilidade de se ter nomes de campos, já que contém duas *Expressões*. Não é surpreendente que, já que a gramática concreta é recursiva, também o é a definição de sintaxe abstrata.

Por exemplo, analise a sintaxe abstrata de uma expressão *Binária*. A Figura 2.11 define a classe de sintaxe abstrata *Expressão* como tendo três componentes: `op`, que é um membro da classe de sintaxe abstrata *Operador*; e `term1` e `term2`, que são membros da classe de sintaxe abstrata *Expressão*.

Um subproduto imediato do uso de sintaxe abstrata é que ela fornece uma base para definir a estrutura abstrata de uma linguagem como um conjunto de classes Java. A seguir, por exemplo, está uma definição de protótipo de classe Java para a sintaxe abstrata de *Expressão*:

```
class Binária extends Expressão {
   Operador op;
   Expressão term1, term2;
}
```

Embora esta definição de protótipo ignore muitas questões de implementação (visibilidade, construtores, métodos de acesso e assim por diante), ela contém todas as variáveis de instância que caracterizam a classe *Binária*, ou seja, uma `Binária` é um tipo de `Expressão` e possui um `op`, que é um `Operador`, e o par `term1` e `term2`, que são membros da classe `Expressão`[5].

Na Figura 2.11, a classe de sintaxe abstrata *Expressão* representa um grupo que consiste de múltiplas classes de sintaxe abstrata, incluindo variáveis, valores inteiros (literais) e expressões que envolvem operadores binários e unários e o número apropriado de operandos.

5. A noção de hierarquia de classe que foi construída em linguagens orientadas a objeto como Java torna direta a implementação dessa especificação em particular. Outra característica de objetos Java que surge com esta abordagem de sintaxe abstrata é a capacidade de consultar um objeto para determinar sua classe. Exploraremos esses recursos de Java quando discutirmos os sistemas de tipo e semântica de tempo de execução de linguagens em capítulos posteriores.

Em termos de implementação de Java, grupos como *Expressão* seriam implementados como uma classe abstrata, já que nenhum objeto poderia ser criado como uma `nova Expressão()`.

```
abstract class Expressão { }
```

Esperamos que agora já tenhamos justificado a separação da sintaxe abstrata da concreta. Embora as duas sejam, de certa forma, redundantes, de outra, elas não o são, ou seja, a sintaxe concreta informa ao programador concretamente o que deve escrever para ter um programa válido na linguagem X. Entretanto, a sintaxe abstrata permite que programas válidos na linguagem X e na Y compartilhem representações abstratas comuns. De forma ideal, um interpretador ou sistema em tempo de execução se preocupa menos sobre como algumas idéias, como laços, são expressas concretamente, e mais sobre o que determinados requisitos computacionais devem ser comunicados da expressão para o ambiente em tempo de execução para o programa.

Este é o papel da sintaxe abstrata: fornecer uma ligação entre a sintaxe e a semântica, entre forma e função. Exploramos o uso dessa ligação em capítulos posteriores.

2.5.2 Árvores de Sintaxe Abstrata

Analise as regras na Figura 2.11 que definem a classe sintática abstrata *Expressão* e suas subclasses *Binária* e *Unária*. Estas regras mencionam os principais elementos de uma *Expressão* sem considerar como ela é formada concretamente pelo analisador, ou seja, a sintaxe concreta fornece a definição da forma de uma expressão, enquanto a sintaxe abstrata fornece a definição dos seus elementos essenciais.

Uma árvore de sintaxe abstrata fornece um meio conciso para descrever os elementos individuais de uma *Expressão*, eliminando as subárvores e os nós intermediários que não contiverem informações essenciais. Para construir uma árvore de sintaxe abstrata, cada categoria sintática abstrata – como *Binária* – possui um nó associado com tantos campos quanto houver elementos distintos no lado direito da sua regra. Por exemplo, o nó associado à *Binária* possui três campos e se parece com a Figura 2.12.

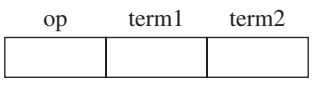
Em contraste, a categoria de sintaxe abstrata *Expressão* não possui campos de informação; ela serve apenas como um mecanismo de agrupamento. Assim, uma *Expressão* não precisa ser implementada como um nó; como já foi observado, uma implementação em Java normalmente implementaria uma *Expressão* como uma classe abstrata. Como exemplo, a Figura 2.13 mostra uma implementação em Java da árvore de sintaxe abstrata para a declaração de atribuição `z = x + 2 * y;`.

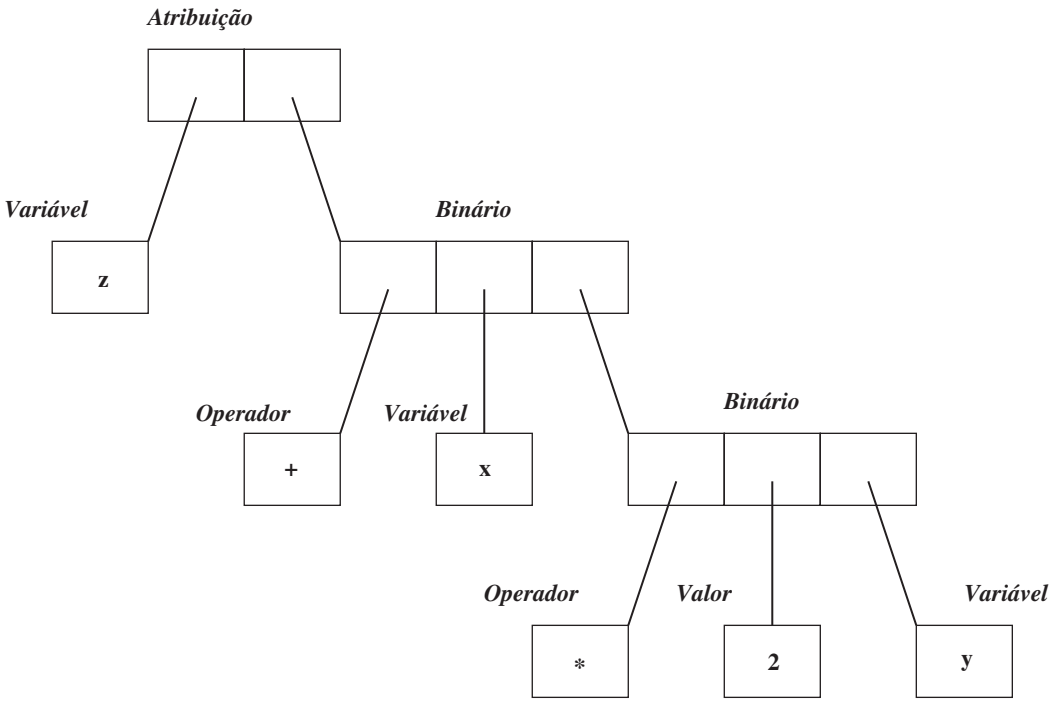
2.5.3 A Sintaxe Abstrata de Clite

A sintaxe abstrata, como discutido na Seção 2.5, forma uma base para conectar as fases de análise sintática e semântica de um compilador. Conforme vimos, ela possibilita uma árvore de análise consideravelmente mais compacta do que a sintaxe concreta, mas, ainda assim, expressa todas as informações requeridas para o processamento semântico, ou seja, a sintaxe abstrata é adequada para determinar o significado de um programa.

Nos próximos capítulos, Clite será usado como um veículo para discutir muitos aspectos técnicos de linguagens de programação orientadas a objeto e imperativas modernas. Por ser um subconjunto de C/C++, Clite será familiar para os leitores que tiverem escrito

Figura 2.12 A Estrutura de um Nó Binário





| Figura 2.13 Árvore de Sintaxe Abstrata para a *Atribuição* `z = x + 2 * y;`

alguns programas em C, C++, Ada ou Java. Em capítulos posteriores, a sintaxe abstrata de Clite servirá como um ponto de partida para a discussão de características de sistemas de tipo, semântica de tempo de execução e gerenciamento de memória. Esta seção introduz a sintaxe abstrata de Clite.

A sintaxe abstrata de Clite estende as idéias básicas apresentadas na seção anterior. Um resumo informal das categorias e de seus significados é apresentado a seguir:

Programa: tendo *Declarações* e um *Composto*.

Declarações: uma seqüência de *Declarações* individuais.

Declaração: uma variável individual, seu tipo e tamanho (se for uma matriz).

Tipo: um membro do conjunto {`int`, `bool`, `float`, `char`}.

Comando: tendo subclasses *Composto, Pular, Atribuição, Condicional* e *Laço.*

Composto: uma seqüência de *Comandos* individuais que são executados na ordem em que aparecem. Cada um aparece no mesmo nível em uma árvore de sintaxe abstrata.

Pular: o comando vazio.

Atribuição: um comando que atribui o valor de uma *Expressão* a uma variável.

Condicional: tendo uma *Expressão* (o teste) e dois *Comandos* (thenbranch e elsebranch), um dos quais é executado e o outro, pulado. No caso de um comando `se-então`, a ramificação elsebranch é instanciada como um comando *Pular*.

```
Programa = Declarações decpart; Comandos body ;
Declarações = Declaração*
Declaração = DecVariável | DecMatriz
DecVariável = Variável v ; Tipo t
DecMatriz = Variável v ; Tipo t ; Inteiro size
Tipo = int | bool | float | char
Comandos = Comando*
Comando = Pular | Bloco | Atribuição | Condicional | Laço
Pular =
Bloco = Comandos
Condicional = Expressão test ; Comando thenbranch, elsebranch
Laço = Expressão test ; Comando body
Atribuição = RefVariável target ; Expressão source
Expressão = RefVariável | Valor | Binária | Unária
RefVariável = Variável | RefMatriz
Binária = Operador op ; Expressão term1, term2
Unária = OpUnário op ; Expressão term
Operador = OpBoleano | OperadorRel | OperadorAritm
OpBoleano = && | ||
OpRelacional = = | != | < | <= | > | >=
OpAritmético = + | - | * | /
OpUnário = ! | -
Variável = String id
RefMatriz = String id ; Expressão index
Valor = ValorInt | ValorBool | ValorFloat | ValorChar
ValorInt = Integer intValue
ValorFloat = Float floatValue
ValorBool = Bolean boolValue
ValorChar = Character charValue
```

| Figura 2.14 Sintaxe Abstrata de Clite

Laço: tendo uma *Expressão* (o teste) e um *Comando* (o corpo do laço) que é repetido enquanto o teste continuar sendo *verdadeiro*.

Expressão: tendo subclasses *Variável, Valor, Binária* e *Unária*.

Binária: tendo um *Operador* e duas *Expressões*.

Unário: tendo um *Operador* e uma única *Expressão.*

Operador: &&, ||, !, <, <=, ==, !=, >, >=, +, −, * e /, representando:

Os operadores boleanos: `e`, `ou`, `não`;

Os operadores relacionais: `menor`, `menor ou igual`, `igual`, `não igual`, `maior`, `maior ou igual`; e

Os operadores aritméticos: `mais`, `menos`, `vezes`, `dividir`.

Valor: tendo subclasses *ValorInt, ValorFloat, ValorBool* e *ValorChar.*

ValorInt: tendo um valor inteiro.

ValorFloat: tendo um valor em ponto flutuante, representando uma aproximação de computador para um valor não-inteiro.

ValorBool: tendo um valor boleano, ou seja, `verdadeiro` ou `falso`.

ValorChar: tendo um único valor do tipo caractere.

Uma descrição formal da sintaxe abstrata de Clite é apresentada na Figura 2.14. Esta descrição fornece uma base adequada, a partir da qual uma árvore de sintaxe abstrata para um programa completo pode ser construída.

Um analisador completo para Clite, capaz de produzir árvores de sintaxe abstrata para programas Clite (como o fragmento mostrado na Figura 2.13), está disponível no site do livro. Esta é uma ferramenta útil para leitores que queiram trabalhar com os aspectos práticos de sintaxe que aparecem nos muitos exercícios no final dos próximos capítulos.

2.6 RESUMO

Este capítulo introduziu a idéia básica de sintaxe e sua importância para o projeto de linguagens atuais de programação. A *sintaxe* se refere à estrutura do texto de um programa, enquanto a *semântica* se refere ao seu significado ou comportamento em tempo de execução. Os programadores se beneficiam de ter uma linguagem cuja sintaxe seja facilmente aprendida e claramente compreendida.

A ambigüidade na descrição sintática ocorre quando um fragmento de programa possui duas ou mais interpretações diferentes. A ambigüidade sintática geralmente pode ser evitada, especialmente quando uma metalinguagem formal como a EBNF é usada. Muitas características de linguagens, como a *associatividade* e a *precedência* de operadores, podem ser claras e concisamente definidas por tal descrição sintática.

A sintaxe de uma linguagem é descrita convenientemente em dois níveis – o nível *léxico* e o *concreto*. A sintaxe léxica ajuda a definir diversas classes fundamentais de *tokens* de uma linguagem, como identificadores e constantes.

Um compilador possui diversas fases, sendo que as duas primeiras são as análises léxica e sintática. As fases restantes de um compilador – análise semântica, otimização de código e geração de código – dependem do término bem-sucedido das duas primeiras. A noção de *sintaxe abstrata* fornece um meio pelo qual os elementos essenciais de um programa podem ser passados para essas fases posteriores para uma análise eficiente.

Uma pequena minilinguagem denominada Clite é introduzida neste capítulo para auxiliar a elucidar diversas questões de sintaxe. Retornaremos a esta linguagem em capítulos posteriores, quando explorarmos questões de projeto de linguagens além da sintaxe – tipos, nomes, funções e semântica.

EXERCÍCIOS

2.1 Usando a gramática de $G_{inteiro}$, desenvolva uma derivação mais à esquerda para o inteiro 4520. Quantos passos são necessários para essa derivação? De modo geral, quantos passos são requeridos para derivar um inteiro com um número arbitrário, digamos *d*, de *Dígitos*?

2.2 Usando a gramática $G_{inteiro}$, desenvolva uma derivação mais à direita para o inteiro 4520.

2.3 Desenvolva uma derivação mais à esquerda para o valor *identificador a2i*, usando a sintaxe BNF dada na Figura 2.7.

2.4 Desenvolva uma derivação mais à direita para o valor *identificador a2i*, usando a sintaxe BNF dada na Figura 2.7.

2.5 Usando a gramática da Figura 2.7, desenhe uma árvore de análise para cada um dos seguintes:

(a) `x = x + a - 1;`

(b) `a = b * c / d;`

(c) `i = i + j * k - 3;`

2.6 Usando a seguinte gramática:

Expressão → *Expressão* + *Termo* | *Expressão* * *Termo* | *Termo*
Termo → 0 | ... | 9 | (*Expressão*)

desenhe uma árvore de análise para cada um dos seguintes:

(a) `5 + 4 * 3`

(b) `5 * 4 + 3`

2.7 Usando a seguinte gramática:

Expressão → *Termo* + *Expressão* | *Termo* * *Expressão* | *Termo*
Termo → 0 | ... | 9 | (*Expressão*)

desenhe uma árvore de análise de cada um dos seguintes:

(a) `5 + 4 * 3`

(b) `5 * 4 + 3`

2.8 Usando a seguinte gramática:

Expressão → *Expressão*+ *Termo* | *Termo*
Termo → *Termo* + *Fator* | *Fator*
Fator → 0 | ... | 9 | (*Expressão*)

desenhe uma árvore de análise para cada um dos seguintes:

(a) `5 + 4 * 3`

(b) `5 * 4 + 3`

2.9 Usando a seguinte gramática:

Expressão → *Expressão* + *Expressão* | *Expressão* * *Expressão* |
0 | ... | 9 | (*Expressão*)

desenhe uma árvore de análise para cada um dos seguintes:

(a) `5 + 4 * 3`

(b) `5 * 4 + 3`

2.10 Usando a seguinte gramática:

$$\textit{Expressão} \rightarrow + \textit{ Expressão Expressão} \mid * \textit{ Expressão Expressão} \mid 0 \mid \ldots \mid 9$$

desenhe uma árvore de análise para cada um dos seguintes:
(a) `+ 5 * 4 3`
(b) `+ * 5 4 3`

2.11 Argumente de forma convincente que a árvore de análise apresentada na Figura 2.2 para a expressão $5 - 4 + 3$ é a única árvore de análise possível. Dica: enumere as outras possibilidades e mostre que elas não funcionam.

2.12 Mostre como a subgramática Java para *ComandoSeEntão* elimina qualquer ambigüidade no comando `se`, ou seja, esboce a árvore de análise usando as regras BNF Java e então argumente que nenhuma outra árvore de análise pode ser encontrada usando estas regras.

2.13 Analise as regras de gramática para *ComandoSe* e *Comando* dadas na Seção 2.1.5. Mostre como elas podem ser alteradas para eliminar sua ambigüidade ilustrada no texto, ou seja, mostre como a gramática pode ser alterada de modo que o programador possa distinguir explicitamente as duas associações possíveis do "senão pendente", conforme mostrado a seguir:

```
se (x<0) se (x==0) y = y - 1; senão y = 0; es
se (x<0) se (x==0) y = y - 1; es senão y = 0; es
```

Aqui, a nova palavra-chave `es` é usada para terminar uma declaração que começa com a palavra-chave `se`. (Isso funciona como um colchete direito, onde `se` é o esquerdo).

2.14 Dê uma gramática e um exemplo de comando `se` para cada uma das seguintes linguagens: (a) Perl, (b)Python, (c) Ada.

2.15 Dê regras de tradução para alternativas EBNF envolvendo metacolchetes e metaparênteses para BNF padrão.

2.16 Reescreva a gramática G_2 como um conjunto de diagramas de sintaxe.

2.17 Escolha uma das seguintes linguagens: Perl, Python, Ada, C, C++, Java ou outra linguagem aprovada pelo professor. Consulte uma fonte autorizada e escreva um relatório que resuma a definição da linguagem quanto ao tamanho e ao tipo da sua gramática, a aparição de ambigüidade na gramática, seu número de palavras reservadas, a precedência e a associatividade de operadores e assim por diante. Cite suas fontes.

2.18 Pesquise o número de palavras reservadas em cada uma das seguintes linguagens: (a) Perl, (b) Python, (c) Ada, (d) C, (e) C++, (f) Java.

2.19 Dê um conjunto de regras de gramática que defina a sintaxe de uma declaração de variável em Perl. Exemplifique.

2.20 Desenhe uma árvore de sintaxe abstrata para cada uma das árvores de análise que você definiu para o Exercício 2.5, usando a sintaxe abstrata para *Atribuição* apresentada na Figura 2.11.

Análise Léxica e Sintática

3

"Perfumaria sintática causa câncer no ponto-e-vírgula."

Alan Perlis

VISÃO GERAL DO CAPÍTULO

O propósito deste capítulo é fornecer uma visão geral da implementação de um analisador léxico (lexer) e um analisador (sintático). Devido ao fato de o analisador léxico estar fundamentado em um modelo mais simples do que uma gramática livre de contexto, o capítulo começa com um breve exame na hierarquia de gramáticas de Chomsky.

Na Seção 3.2 examinamos especificações alternativas de *tokens* de Clite, incluindo conjuntos regulares e autômatos de estados finitos determinísticos. Este último fornece um modelo que é facilmente implementado, conforme demonstramos com uma implementação parcial para a parte léxica da gramática de Clite.

A seguir, desenvolvemos um analisador descendente recursivo para uma pequena parte da sintaxe concreta de Clite. Para fazer isso, desenvolvemos alguns algoritmos auxiliares. Depois mostramos como converter uma gramática EBNF para um analisador descendente recursivo.

Este capítulo não pretende substituir um curso sobre construção de compiladores. Contudo, é útil para as escolas que não oferecem tal curso e para aqueles alunos que não

tenham feito tal curso. Além disso, ao aprendermos como construir um analisador descendente recursivo, teremos um algoritmo muito útil em nosso repertório, especialmente para "pequenas linguagens". Por exemplo, algumas das soluções dos exercícios neste texto usam a técnica de descrição de dados, utilizando uma gramática simples e depois construindo uma representação interna para os dados por meio do uso de um analisador descendente recursivo (veja o Exercício 3.21).

3.1 A HIERARQUIA DE CHOMSKY

Chomsky (1957) definiu quatro classes de gramáticas, cada uma correspondendo a uma classe única de linguagem e máquina teórica. Da mais simples para a mais complexa, estes tipos de gramática são:

- Gramáticas regulares.
- Gramáticas livres de contexto (equivalentes a gramáticas BNF).
- Gramáticas sensíveis ao contexto.
- Gramáticas irrestritas.

Apenas as duas primeiras categorias são usadas em linguagens de programação; as duas últimas foram incluídas apenas para que todas fossem mencionadas. As gramáticas livres de contexto foram discutidas na Seção 2.1.

Lembre-se do que foi dito na Seção 2.1, que uma gramática consiste de um conjunto de produções, um conjunto de símbolos não-terminais N, um conjunto de símbolos terminais T e um símbolo inicial S que deve ser não-terminal. Os símbolos terminais são o alfabeto da linguagem sendo definida, enquanto os símbolos não-terminais representam conceitos da linguagem. Usando a sintaxe concreta de Clite como exemplo, os símbolos terminais incluem as palavras reservadas (por exemplo, `while`, `if`), literais (por exemplo, 542, 1.5, `true`), operadores etc. Os não-terminais representam conceitos de Clite como uma *Declaração,* uma *Expressão* etc.

Na hierarquia de Chomsky, apenas a forma de uma produção muda de um tipo de gramática para outro. O conceito de terminais, não-terminais e o símbolo inicial permanecem os mesmos.

Gramáticas regulares são a classe mais simples e equivalem às assim denominadas *expressões regulares* e aos *autômatos de estados finitos* (ambos discutidos na Seção 3.2) no seu poder expressivo. Uma *gramática regular à direita* (ou gramática linear à direita) só pode ter produções na seguinte forma:

$$A \rightarrow \omega\, B$$
$$A \rightarrow \omega$$

em que $\omega \in T^*$ (uma *string* de zero ou mais terminais[1]) e $B \in N$ (um único não-terminal). Em outras palavras, cada alternativa de uma produção pode ter no máximo um não-terminal e, caso exista, ele deve ser o símbolo mais à direita. O não-terminal B pode ser o mesmo não-terminal A, ou seja, a recursão direta é permitida. Reescrevendo a produção para *Inteiro* da Figura 2.7 (Clite) como uma gramática regular à direita, obtemos:

Inteiro → 0 *Inteiro* | ... | 9 *Inteiro* |
0 | ... | 9

1. T^* é outro exemplo de uso da notação de estrela de Kleene.

ou o equivalente em EBNF:

$$\begin{aligned} \textit{Inteiro} \rightarrow\ & (0 \mid ... \mid 9)\ \textit{Inteiro} \mid \\ & 0 \mid ... \mid 9 \end{aligned}$$

Uma alternativa define *gramáticas regulares à esquerda* (ou gramáticas lineares à esquerda) de forma equivalente, exceto que o único não-terminal em uma alternativa, caso exista, deve ser o símbolo mais à esquerda. Reescrevendo a produção para *Inteiro* da Figura 2.7 (Clite) como uma gramática regular à esquerda, obtemos:

$$\begin{aligned} \textit{Inteiro} \rightarrow\ & \textit{Inteiro}\ 0 \mid ... \mid \textit{Inteiro}\ 9 \mid \\ & 0 \mid ... \mid 9 \end{aligned}$$

ou o equivalente em EBNF:

$$\begin{aligned} \textit{Inteiro} \rightarrow\ & \textit{Inteiro}\ (0 \mid ... \mid 9) \mid \\ & 0 \mid ... \mid 9 \end{aligned}$$

Gramáticas regulares e suas formas equivalentes de expressões regulares e autômatos de estado finito são usadas na construção de *tokenizadores* para converter fluxos de caracteres em *tokens* (ou símbolos básicos). Isto é discutido em detalhes na Seção 3.2.

Gramáticas regulares são consideravelmente menos poderosas do que gramáticas livres de contexto ou BNF. Por exemplo, um teorema bem conhecido é que a linguagem

$$\{a^n b^n \mid n \geq 1\}$$

não é uma linguagem regular, ou seja, a linguagem não pode ser gerada por uma gramática regular. Em uma gramática de linguagem de programação como Clite, alguns símbolos terminais vêm em pares correspondentes: parênteses em expressões, colchetes em listas de declarações etc. Linguagens regulares não conseguem lidar com o caso mais simples de balanceamento de parênteses, em que todos os parênteses à esquerda precedem os parênteses à direita. Portanto, gramáticas regulares são inadequadas para descrever apropriadamente a sintaxe de uma linguagem de programação.

Gramáticas livres de contexto, na forma de gramáticas BNF, já foram discutidas com algum detalhe. Uma vantagem das gramáticas livres de contexto é que para uma classe ampla de gramáticas não ambíguas há analisadores com base em tabelas (veja a Observação na página 81). Gramáticas livres de contexto correspondem a autômatos *push-down*, que são discutidos na Seção 3.3.

Gramáticas sensíveis ao contexto possuem produções definidas da seguinte maneira [Hopcroft e Ullman, 1979]:

$$\alpha \rightarrow \beta$$

de modo que $|\alpha| \leq |\beta|$, em que $\alpha, \beta \in (N \cup T)^*$. Ou seja, diferentemente das gramáticas livres de contexto, o símbolo à esquerda não se re*string*e a ser um único não-terminal, mas, em vez disso, pode consistir de uma *string* que contenha tanto símbolos terminais quanto não-terminais. A restrição sobre gramáticas sensíveis ao contexto é que o comprimento do lado direito de uma alternativa não pode ser menor do que o do lado esquerdo, exceto que o símbolo inicial pode derivar a *string* vazia. Em uma derivação, a forma seqüencial não pode encolher de um passo de derivação para outro. Gramáticas sensíveis ao contexto podem assegurar que os identificadores de um programa sejam declarados e tenham o tipo apropriado para o contexto.

Do ponto de vista da definição de sintaxe e de semântica de uma linguagem de programação, linguagens sensíveis ao contexto (como as linguagens definidas por uma gramática G sensível ao contexto) possuem muitas propriedades indesejadas; por exemplo:

- Dada uma *string* terminal ω e uma gramática G, não é possível determinar se $\omega \in L(G)$, ou seja, se ω é derivável a partir de G. Nos nossos termos, não é possível determinar se um programa é válido de acordo com G.
- Dadas uma *string* terminal ω e uma gramática G, não é possível determinar se $L(G)$ possui alguma *string* válida. Em outras palavras, não é possível determinar se uma gramática sensível ao contexto para Clite define algum programa Clite válido.

Para os nossos propósitos, o termo *indeterminável* significa que você não pode escrever um programa de computador que garantidamente pare (termine a execução) para todas as suas entradas e determine a solução do problema. Por esse motivo, gramáticas sensíveis ao contexto não são usadas no tratamento de sintaxes de linguagens de programação.[2]

Gramáticas irrestritas cessam as restrições de comprimento do lado direito. Gramáticas irrestritas são equivalentes a máquinas de Turing ou, de forma equivalente, C/C++ integrais. Gramáticas irrestritas podem calcular qualquer função calculável [Hopcroft e Ullman, 1979]. Do ponto de vista da definição de sintaxe de uma linguagem de programação, as gramáticas irrestritas sofrem das mesmas propriedades indesejáveis das gramáticas sensíveis ao contexto.

3.2 ANÁLISE LÉXICA

Conforme já observado na Seção 2.4, o propósito da fase da análise léxica de um compilador é transformar o programa-fonte de uma seqüência de caracteres em uma seqüência de *tokens*. No processo, os espaços em branco e os comentários são descartados. Nesta seção, mostramos como um *lexer* é projetado e implementado para a parte léxica da gramática Clite da Figura 2.7, ampliada pelas definições de espaços em branco e pelos comentários.

> **Definição:** Um *token* é uma seqüência logicamente coesa de caracteres que representam um único símbolo.

Exemplos incluem um identificador, uma palavra reservada (como `while`), um literal (como `3.1416`), um operador (por exemplo, `!=`) e uma pontuação (por exemplo, ;).

Conforme discutido na Seção 2.4, historicamente o *lexer* é uma fase de compilação separada pelos seguintes motivos:

1. O projeto do analisador léxico é baseado em um modelo de máquina muito mais simples e rápido do que o modelo livre de contexto usado para análise sintática.
2. Para um compilador não-otimizador, aproximadamente 75% do tempo total de compilação é consumido na análise léxica. Qualquer melhoria que faça com que essa fase seja executada mais rapidamente possui um impacto comparável no tempo total de compilação.
3. Antes da criação de ASCII, cada computador tinha seu próprio conjunto idiossincrático de caracteres. Esconder este conjunto de caracteres no *lexer* fez com que portar um compilador para uma nova máquina fosse muito mais fácil. Mesmo hoje,

2. Questões sensíveis ao contexto como a resolução de nomes e a verificação de tipos podem ser lidadas por uma gramática de atributos. Uma alternativa explorada no Capítulo 8 são as semânticas denotacionais.

vestígios desse problema ainda existem quando são usados os conjuntos de caracteres ASCII, Unicode e EBCDIC.

4 A convenção do final de linha também varia de um sistema operacional para outro. Por exemplo, o Apple Macintosh usa o retorno de carro (ASCII código 13) como caractere de final de linha, Linux/Unix usa um caractere de nova linha (ASCII código 10), e o Microsoft Windows usa uma combinação retorno de carro/nova linha. Para muitas linguagens modernas, mas não todas, esse problema é manipulado pela biblioteca de suporte de tempo de execução da linguagem. No início não era assim, de modo que a dependência do sistema operacional foi escondida dentro do analisador léxico.

O léxico de Clite é um conjunto de *tokens* que são definidos pela gramática e pelas seguintes categorias gramaticais:

- Identificadores (por exemplo: `token`)
- Literais inteiros (por exemplo: `80`)
- Literais de ponto flutuante (por exemplo: `3.1415`)
- Literais de caracteres (por exemplo `'a'`)

Isso significa que, por exemplo, quando o analisador está esperando um identificador, ele não se importa com qual é especificamente encontrado, mas apenas que um seja encontrado. Isso é muito diferente de uma palavra reservada como `while` que, quando vista no início de uma declaração, determina o tipo, a declaração e sua estrutura. Literais boleanos são deixados de fora dessa lista porque apenas esses literais são palavras reservadas.

Precisamos adicionar à lista anterior os espaços em branco, os comentários, o indicador de final de linha (necessário para comentários `//`) e um indicador de final de arquivo. Estes itens acrescentados, exceto o último, são descartados pelo *lexer*, já que o analisador não precisa conhecê-los. Assim, a lista completa de seqüências de caracteres a serem reconhecidos é:

- *Identificadores*: uma letra seguida de uma ou mais letras e dígitos
- *Literais*, incluindo:
  - literais inteiros: uma seqüência de um ou mais dígitos
  - literais de ponto flutuante: uma seqüência de um ou mais dígitos seguidos por um ponto seguido por uma seqüência de um ou mais dígitos
  - literais de caracteres: 'um único caractere imprimível'
- Palavras-chave: `bool char else false float if int main true while`
- Operadores: `= || && == != < <= > >= + - * / ! [ ]`
- Pontuação: `; , { } ( )`
- Espaço em branco: caracteres de espaço ou tabulação
- Comentários:
  - de uma única linha: `//` quaisquer caracteres de final de linha
- Final de linha
- Final de arquivo

Qualquer outro caractere é ilegal.

Na Seção 3.2.1, uma forma alternativa de especificar o léxico de uma linguagem de programação é apresentada.

3.2.1 Expressões Regulares

Uma alternativa a gramáticas regulares para especificação formal de uma linguagem de nível léxico é a *expressão regular*. A linguagem de expressões regulares é resumida a seguir, usando sintaxe adaptada da família Lex (Mason *et al.*, 1992) de geradores léxicos:

Expressão Regular	Significado
`x`	Um caractere x (significa ele próprio)
`\x`	Um caractere de escape, por exemplo, `\n`
{ `nome` **}**	Uma referência a um nome
`M` **|** `N`	M ou N
`M N`	M seguido de N (concatenação)
`M*`	Zero ou mais ocorrências de M
`M+`	Uma ou mais ocorrências de M
`M?`	Zero ou uma ocorrência de M
[`aeiou`**]**	O conjunto de vogais: a, e, i, o , u
[`0-9`**]**	O conjunto de dígitos, 0 a 9
.	Qualquer caractere (único)

Observe que Lex usa `\` para caracteres de escape, que de outra forma seriam interpretados como metacaracteres (como chaves, `\{` e `\}`), enquanto aqui usamos a fonte de `largura fixa` para todos os não-metacaracteres.

Com essas convenções, agora redefinimos as diversas classes sintáticas léxicas de Clite da Figura 3.1. Observe que podem ser usados parênteses para agrupar elementos de uma expressão regular.

As primeiras seis definições são auxiliares. A primeira definição indica que o conjunto de caracteres imprimíveis em ASCII consiste de todos os caracteres, indo do caractere de espaço

Categoria	Definição
qualquerCaractere	**[**-~**]**
letra	**[**a-zA-Z**]**
dígito	**[**0-9**]**
espaçoembranco	**[**\t**]**
eol	\n
eof	\004
palavra-chave	`bool` **|** `char` **|** `else` **|** `false` **|** `float` **|** `if` **|** `int` **|** `main` **|** `true` **|** `while`
identificador	**{**letra**}({**letra**} | {**dígito**})+**
litInteiro	**{**dígito**}+**
litPontoFlutuante	**{**dígito**}+**\ .**{**dígito**}+**
litCaractere	‘**{**qualquerCaractere**}**’
operador	= **|** || **|** && **|** == **|** != **|** < **|** <= **|** > **|** >= **|** + **|** - **|** * **|** / **|** ! **|** [**|**]
separador	; **|** , **|** { **|** } **|** (**|**)
comentário	//(**{**qualquerCaractere**} | {**espaçoembranco**})***{eol}

| Figura 3.1 Expressões Regulares para Sintaxe Léxica Clite

(código ASCII 27) até o caractere de til (código ASCII 126). A segunda linha define um conjunto de letras, incluindo as maiúsculas e minúsculas, enquanto a terceira linha define o conjunto de dígitos. A seguir, os caracteres de espaço em branco são definidos, excluindo o final de linha. As duas definições auxiliares finais são os caracteres de final de linha e de final de arquivo que utilizam convenções adotadas de Linux/Unix.

As linhas restantes, exceto a última, definem os diversos *tokens*: identificadores; palavras-chave; literais de inteiros, ponto flutuante e caractere; operadores; e separadores (ou pontuação). A última definição especifica um comentário `//`, que pode incluir caracteres imprimíveis e espaço em branco, terminando com o final de linha (representado por um caractere de nova linha).

Expressões regulares são uma ferramenta popular no projeto de linguagens porque suportam facilmente a geração automática de analisadores léxicos, ou seja, em vez de projetar manualmente e escrever código para um analisador léxico, podem-se submeter as expressões regulares diretamente a um gerador de analisador léxico. Dois geradores comumente usados são da família Lex (Lex, Flex etc.) para gerar código C/C++ e JLex para gerar código Java. Mais informações sobre isso podem ser encontradas em diversos textos sobre projeto de compiladores (Aho et al., 1986; Appel, 1998).

Apresentadas as definições anteriores do espaço de possíveis entradas, como projetamos um programa para reconhecer essas seqüências? Exploramos esta questão a seguir.

3.2.2 Autômatos de Estados Finitos

Como já foi observado, há diversas formas equivalentes de se especificar o espaço de possíveis entradas de um *lexer*: linguagem natural (como a mostrada anteriormente), uma gramática regular (preferencialmente regular à direita) ou expressões regulares. Tradutores dessas últimas convertem-nas primeiro para um *autômato de estado finito determinístico* (DFSA – *deterministic finite state automaton*) e depois extraem um *lexer* do DFSA.

Assim que se tiver adquirido a compreensão sobre como um DFSA, para reconhecer *tokens* e outras seqüências de caracteres (como comentários), é construído e convertido, para código, para seqüências simples de caracteres, raramente se terá de passar por esse exercício novamente. Todavia, em uma tentativa de evitar essa discussão de DFSAs, a primeira edição deste texto apresentou um projeto *ad hoc*. As soluções dos Exercícios 3.14 e 3.15 apresentam tipos de problemas nos quais o material desta seção deve sempre ser usado; os autores encontraram compiladores nos quais estes problemas estavam incorretos.

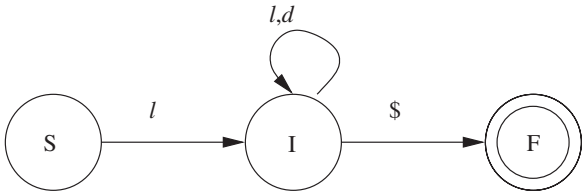
Definição: Um *autômato de estado finito* possui:

1. Um conjunto de regras, representado por nodos em um grafo.
2. Um alfabeto de entrada, aumentado por um símbolo único que representa o final da entrada.
3. Uma função de transição de estado, representada por arcos direcionados de um nodo para outro, rotulados por um ou mais símbolos do alfabeto.
4. Um estado inicial único.
5. Um conjunto de um ou mais *estados finais* (estados sem arestas de saída).

Um autômato de estado finito é *determinístico* se, para cada estado e símbolo de entrada, houver no máximo um arco de saída do estado rotulado com esse símbolo de entrada.

Por exemplo, um autômato de estado finito que reconheça identificadores é mostrado na Figura 3.2, em que *S* é o estado inicial e *F*, o estado final. O símbolo *l* representa o conjunto de letras, enquanto o símbolo *d* representa o conjunto de dígitos; $ é o marcador de final de entrada. O autômato da Figura 3.2 é determinístico.

Figura 3.2
Um Autômato de Estado Finito para Identificadores



Uma *configuração* de um autômato de estado finito consiste de um estado e a entrada restante, terminada pelo símbolo especial de final de entrada. Um *movimento* significa atravessar o arco que sai do estado que corresponde ao símbolo de entrada mais à esquerda, dessa forma, consumindo-o; se nenhum movimento estiver definido para o par de símbolos estado-entrada, o autômato pára no erro, rejeitando a entrada. Uma entrada é *aceita* se, partindo do estado inicial, o autômato consumir todos os seus símbolos e parar em um estado final.

Usando o autômato da Figura 3.2, os seguintes passos são dados para aceitar a entrada `a2i$` começando do estado *S*:

1. Mover-se para o estado *I* com entrada `2i$`.
2. Mover-se para o estado *I* com entrada `i$`.
3. Mover-se para o estado *I* com entrada `$`.
4. Mover-se para o estado *F* sem entrada restante.

Já que *F* é um estado final, `a2i` é, assim, aceito como um identificador válido. Formalmente, esta seqüência de passos é escrita da seguinte forma:

$$
\begin{aligned}
(S, a2i\$) &\vdash (I, 2i\$) \\
&\vdash (I, i\$) \\
&\vdash (I, \$) \\
&\vdash (F,)
\end{aligned}
$$

em que $\vdash$ denota um *movimento*. Assim, mostramos:

$$(S, a2i\$) \vdash^* (F,)$$

ou seja, o DFSA aceita a *string* *a2i*$ em zero ou mais movimentos, terminando em um estado final. Observe que $\vdash^*$ é outro uso da notação de estrela de Kleene.

Para facilitar o projeto de um DFSA a reconhecer a sintaxe léxica de Clite, as seguintes convenções serão usadas:

- Um símbolo terminador explícito aparece apenas para o programa-fonte de Clite como um todo, em vez de para cada categoria léxica.
- O símbolo *l* representa uma letra arbitrária e *d*, um dígito arbitrário. Os símbolos *eoln* e *eof* representam o final da linha e o final do arquivo, respectivamente. Todos os outros símbolos representam a si próprios.
- Um arco direcionado sem rótulo representa qualquer outro símbolo de entrada válido, ou seja, o conjunto de todos os símbolos de entrada válidos menos os rótulos de arco para os outros arcos saindo do mesmo estado.

- O reconhecimento de *tokens* termina em um estado final (um sem arcos saindo).
- O reconhecimento de não *tokens* possui arcos voltando para o estado inicial.
- O reconhecimento do final de arquivo, representando o final do programa-fonte.

Finalmente, o autômato deve ser determinístico. Para assegurar isso, retiramos o reconhecimento de palavras-chave do DFSA, já que elas formam um subconjunto dos identificadores. Também analisamos todas as classes de *tokens* juntas que tenham um prefixo comum. Por exemplo, tanto os literais `int` quanto `float` começam com uma seqüência de dígitos. Como outro exemplo, o símbolo de divisão e o de comentário começam ambos com uma `/`, de modo que são analisados juntos.

O DFSA necessário para Clite está esboçado na Figura 3.3. Para que o diagrama coubesse em uma página, muitos dos casos para pontuação simples e operadores foram omitidos (é apresentado um de cada tipo). Por exemplo, o caso do caractere + aparece, mas os casos para os outros *tokens* de um caractere (`-`, `*` e assim por diante) são omitidos. De forma semelhante, o caso da distinção do operador de atribuição (=) do caso de igualdade (==) é dado, mas não o menor que (<) *versus* menor ou igual (<=) e assim por diante. Observe que as ocorrências de espaços em branco *ws*, incluindo *eoln* e comentários volta para o estado inicial *S*. Os subdiagramas para literais de ponto flutuante e inteiros e identificadores são semelhantes à Figura 3.2.

Também deve ser observado que este projeto ignora palavras reservadas, o que simplifica bastante o DFSA. Em vez disso, assim que um identificador tiver sido reconhecido, o código que implementa o DFSA faz uma simples busca em uma tabela para distinguir palavras reservadas de identificadores.

O analisador léxico (*lexer*) implementa este DFSA. Nessa implementação, o *lexer* supõe que o analisador o chama cada vez que outro *token* é necessário. Assim, o *lexer* precisa lembrar onde parou no consumo de caracteres-fontes de entrada cada vez que for chamado.

Além disso, a presença de laços no DFSA fará com que o *lexer* avance um caractere a mais na leitura da entrada. Por exemplo, analise a seguinte entrada-fonte:

```
se (a<b)
```

Na ausência de espaço em branco após o `a`, o DFSA reconhecerá o `a` como um *Identificador* somente após avançar para o símbolo <. A próxima chamada no *lexer* avançará imediatamente para o próximo caractere, a saber, `b`. Há diversas soluções possíveis para este problema:

1. Usar uma função de leitura à frente (*look-ahead*) para permitir o exame do próximo caractere sem consumi-lo. Em termos de tempo de execução, este é o projeto menos eficiente.
2. Usar uma função de envio de volta (*push-back*) de modo que um caractere possa ser retornado para a entrada quando necessário. Esta função seria usada no reconhecimento de *strings* de dígitos e identificadores, por exemplo, mas continua pouco eficiente.
3. Adote a convenção de que sair do estado inicial **não** consome um caractere. Em termos de tempo de execução, este é o projeto mais eficiente, já que cada caractere é examinado exatamente uma vez.

Nosso projeto DFSA para Clite usa a terceira alternativa, que necessita da introdução de uma transição extra para avançar a entrada em diversos lugares. Por exemplo, no caso em que um caractere de soma (+) é visto saindo do estado inicial, uma transição sem rótulo para um novo estado é adicionada de modo a manter um caractere à frente.

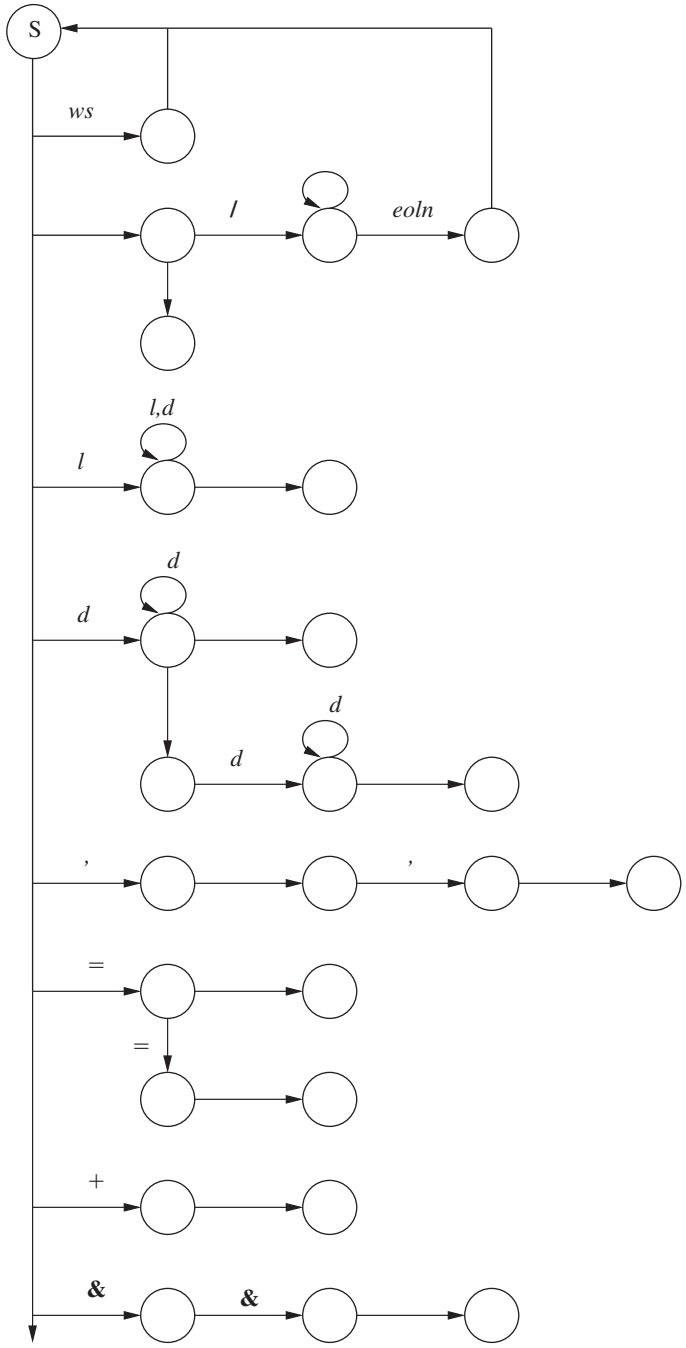


Figura 3.3 Projeto do DFSA do Analisador Léxico

Aqui, *ws* representa caracteres de espaço em branco; *d*, os dígitos; *l*, letras; *eoln*, final de linha; *eof*, final de arquivo; e todos os outros caracteres representam a si próprios. Um arco sem rótulo representa todos os outros caracteres.

Observação

Variante da Função de *Pushback*

Uma variante interessante do uso de uma função de *pushback* para executar uma procura à frente ocorreu em diversos compiladores Pascal de ETH. Pascal permitia subfaixas na forma `x..y`, na qual tanto x quanto y tinham de ser constantes inteiras ou literais inteiras. Analise a subfaixa `5..12` na ausência de espaços em branco. Para distingui-la do literal de ponto flutuante 5.12, o compilador tinha de avançar para um segundo ponto. Na falta de um recurso de *pushback*, o lexer substituía o segundo ponto com dois pontos (:). Testes confirmaram que o compilador aceitava `5:12` como uma faixa válida, embora fosse sintaxe léxica Pascal não permitida.

3.2.3 Do Projeto ao Código

Tendo estas considerações em mente, agora podemos implementar um DFSA para Clite[3]. Já que os casos tanto para espaços em branco quanto para comentários voltam para o estado inicial, em um nível mais externo, a função para obter o próximo *token* deveria ser um laço `do-while` contendo um grande comando `switch`, conforme mostrado na Figura 3.4.

O laço é repetido até que algo diferente do espaço em branco ou de um comentário seja encontrado, quando então o laço pode ser terminado por meio de um comando `return`. A variável `ch` contendo o caractere de entrada corrente deve ser global, já que seu valor deve ser lembrado de uma chamada do método `next` até a próxima; a variável `ch` é inicializada com um caractere de espaço.

Para completar o processo de tradução do DFSA para o código do *lexer*, as seguintes regras devem ser seguidas:

1 O código correspondente a atravessar um arco do nó *A* para o *B* deve obter o próximo caractere, a menos que *A* seja o estado inicial.

(a) Se o rótulo *x* do arco for um único caractere, deve haver um teste `ch == x`.

(b) Se o rótulo `x` do arco for um conjunto, deve haver um teste que implemente efetivamente $ch \in x$, ou seja, um teste para ver se *ch* faz parte do conjunto desejado.

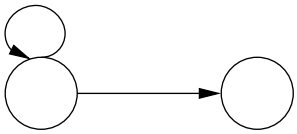
```
private chair ch = ' ';
public Token next ( ) {
 do {
     switch (ch) {
           ...
     }
 } while (true);
}
```

| Figura 3.4 Esboço da Rotina para Obter o Próximo Token

3. Estritamente falando, este passo poderia ser executado mecanicamente, como tem sido feito em muitas gerações de analisadores, ou seja, poderíamos gerar tabelas representando o DFSA e usar um algoritmo orientado a tabelas como sendo o *lexer*; versões iniciais de Lex usavam essa abordagem. O problema com ela é que o *lexer* resultante é executado até 10 vezes mais lentamente do que quando o DFSA é traduzido à mão para o código.

(c) Se o arco não tiver rótulo, deve haver um teste para garantir que `ch` não seja um dos rótulos em algum dos outros arcos. (Se este for o único arco saindo do estado, nenhum teste é necessário.)

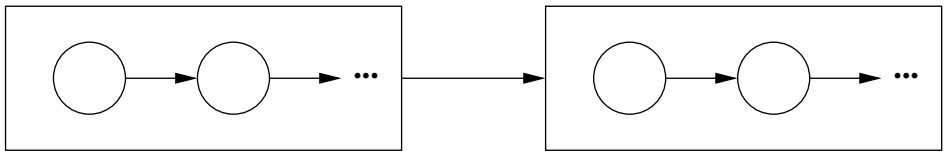
2 Um nó com um arco para si próprio corresponde a um laço `do-while`, no qual a condição corresponde ao rótulo do arco.



3 Caso contrário, o movimento é traduzido para um comando `switch` (ou `if`), com cada arco sendo um *case* separado. Se um dos arcos não estiver rotulado, ele se torna o *case default*. Caso contrário, o *case default* é um *case* de erro.

4 Uma seqüência de transições se torna uma seqüência de comandos traduzidos.

5 Um subdiagrama complexo é traduzido colocando seus componentes dentro de uma caixa, tornando-a efetivamente um nó e então traduzindo-se cada caixa usando a estratégia *outside-in*.



Aplicar essas regras ao projeto do DFSA na Figura 3.3 resulta no código dado na Figura 3.5, novamente com os *cases* semelhantes omitidos. Para simplificar esse código, a rotina auxiliar `nextChar` traduz um caractere de final de linha para o de nova linha e um de final de arquivo para um Control-D, usando convenções comuns Linux/Unix. Essas suposições funcionam para Clite porque o programa-fonte é restrito a caracteres imprimíveis (não de controle, exceto apenas pelo caractere de tabulação).

Este esqueleto de código usa diversos métodos para executar sua tarefa. Algumas são implementações das definições de expressões regulares auxiliares, como `isLetter`:

```
private boolean isLetter (char c) {
    return (c>='a' && c<='z' || c>='A' && c<='Z');
}
```

Outros são métodos que representam um código comum. Por exemplo, a lógica básica para processar os caracteres de um identificador é repetida para processar um número inteiro e uma fração decimal:

```
private String concat(String set) {
    StringBuffer r = new StringBuffer("");
    do {
        r.append(ch);
        ch = nextChar();
    } while (set.indexOf(ch) >= 0);
    return r.toString( );
}
```

```
public Token next( ) { // Retorna o próximo token
    do {
        if (isLetter(ch)) { // identificador ou palavra-chave
            String spelling = concat(letters + digits);
            return Token.keyword(spelling);
        } else if (isDigit(ch)) { // literal inteiro ou de ponto flutuante
            String number = concat(digits);
            if (ch != '.') // literal inteiro
                return Token.mkIntLiteral(number);
            number += concat(digits);
            return Token.mkFloatLiteral(number);
        } else switch (ch) {
        case ' ': case '\t': case '\r': case eolnCh:
            ch = nextChar();
            break;
        case '/' ': // divisão ou comentário
            ch = nextChar();
            if (ch != '/') return Token.divideTok;
            // comentário
            do {
                ch = nextChar();
            } while (ch != eolnCh);
            ch = nextChar();
            break;
        case ' \ ': // caractere literal
            char ch1 = nextChar();
            nextChar(); // ler '
            ch = nextChar();
            return Token.mkCharLiteral("" + ch1);
        case eofCh: return Token.eofTok;
        case '+': ch = nextChar();
            return Token.plusTok;
        ...
        case '&': check('&'); return Token.andTok;
        case '|': check('|'); return Token.orTok;
        case '=':
            return chkOpt('=', Token.assignTok,
                              Token.eqeqTok);
        ...
        } // switch
    } while (true);
}
```

| Figura 3.5 Método Tokenizador do Lexer de Clite

Dois outros métodos auxiliares são `check` e `chkOpt`. O primeiro deles, `check`, é usado para processar dois *tokens* de caracteres nos quais o primeiro caractere por si só não é um *token* válido; um exemplo é o operador de conjunção `&&`. O segundo, `chkOpt`, processa dois *tokens* de caracteres nos quais o primeiro caractere, por si só, é um *token* válido. Se o seu argumento caractere não estiver presente, ele retorna o primeiro argumento *token*; se estiver presente, retorna o segundo argumento *token*. Ele também avança pelo fluxo de caracteres-fonte conforme necessário. Neste caso, um exemplo é o operador de atribuição = *versus* o operador de igualdade ==.

Uma implementação completa das classes `Lexer` e `Token` para a sintaxe léxica apresentada na Figura 2.7 pode ser baixada do site do livro. Embora esta implementação seja feita em Java, pode ser facilmente traduzida para C, C++ ou Ada.

Para ilustrar a operação do `Lexer`, analise o seguinte programa Clite do exemplo:

```
// Um primeiro programa com
// duas linhas de comentário
int main() {
    char c;
    int i;
    c = 'h';
    i = c + 3;
} // main
```

Para esse programa, o *lexer* produz a lista de *tokens* mostrada na Figura 3.6, excluindo o *token* de final de arquivo. Na lista, a maioria dos *tokens* aparece como eles próprios. Quatro tipos de *tokens* (*Identificador, litInteiro, litPontoFlutuante* e *litCaractere*) aparecem com seu tipo de *token* e seu valor associado.

3.3 ANÁLISE SINTÁTICA

> **Definição:** O propósito do *analisador sintático*, ou *analisador*, é construir uma árvore de análise que use como entrada o fluxo de *tokens* fornecido pelo *lexer*.

A saída do analisador geralmente é uma árvore de sintaxe abstrata. A motivação para o uso de uma árvore de sintaxe abstrata, em vez de uma árvore de análise, foi discutida na Seção 2.5.1.

```
int
main
(
)
{
char
Identificador c
;
int
Identificador i
;
```

```
Identificador c
=
litCaractere h
;
Identificador i
=
Identificador c
+
litInteiro 3
;
}
```

| Figura 3.6 Lista de Tokens de um Programa de Exemplo

Definição: Um analisador *descendente recursivo* é um analisador no qual cada não-terminal da gramática é convertido em uma função que reconhece as entradas deriváveis desse não-terminal.

Para uma linguagem com uma gramática grande como C/C++, Java ou Ada, um gerador de analisador seria usado para gerar um analisador orientado a tabela (veja a Observação na página 81). Entretanto, para linguagens com gramáticas pequenas, é preferível um analisador descendente recursivo. Exemplos de tais linguagens incluem arquivos de configuração, fórmulas em planilhas eletrônicas e "pequenas linguagens" como a do Exercício 3.21.

Como vimos na Seção 2.1, a gramática de uma linguagem de programação define a estrutura das diferentes partes de um programa como expressões aritméticas (por exemplo, `x+2*y`), declarações de atribuição (por exemplo, `z=2*x+y;`), comandos de laço (por exemplo, *for* `(i=0; i<n; i++) a[i]=a[i] + 1;`), definições de funções, declarações de variáveis (por exemplo, `int n;`) e até mesmo programas completos. A sintaxe EBNF de uma linguagem fornece uma definição precisa na qual a geração de uma árvore de análise a partir de um fluxo de *tokens* pode se basear.

Analise as categorias sintáticas *Atribuição* e *Expressão*, que incluem todas as seqüências de *Tokens* que descrevem cálculos aritméticos e atribuições do resultado a uma variável. Uma gramática EBNF simples para essas categorias abreviadas da gramática Clite na Seção 2.3 é mostrada na Figura 3.7. Esta gramática servirá como a gramática de exemplo para construir um analisador. Na gramática, o símbolo terminal *Literal* denota apenas um literal inteiro, ou seja, um número completo sem sinal.

3.3.1 Definições Preliminares

No preparo da construção de um analisador para esta gramática de expressão, precisamos definir quatro conceitos preliminares: *gramáticas expandidas, Primeiro, anulável* e *grafo de dependência à esquerda*.

Algoritmos que construam analisadores às vezes requerem que a gramática esteja em uma *forma expandida*. Nesta forma, um novo símbolo inicial S' é introduzido com uma única regra, a saber:

$$S' \rightarrow S\ \$$$

em que S é o símbolo original e \$ é um novo símbolo terminal que representa o final da informação. Por exemplo, na Seção 3.2, este terminal era o *token* de final de arquivo.

Outro conceito-chave usado na construção de analisadores é o de conjunto *Primeiro*, que contém todos os terminais que sejam símbolos mais à esquerda deriváveis de um determinado símbolo, ou seja:

$$Primeiro(X) = \{a \in T \mid X \Rightarrow^* aw, w \in (N \cup T)^*\}$$

Figura 3.7
Sintaxe Concreta para Atribuições e Expressões

$$
\begin{aligned}
\textit{Atribuição} &\rightarrow \textit{Identificador} = \textit{Expressão}\ \mathbf{;}\\
\textit{Expressão} &\rightarrow \textit{Termo}\ \{\ \textit{opAdição Termo}\ \}\\
\textit{opAdição} &\rightarrow + \mid -\\
\textit{Termo} &\rightarrow \textit{Fator}\ \{\textit{opMultiplicação Fator}\ \}\\
\textit{opMultiplicação} &\rightarrow * \mid /\\
\textit{Fator} &\rightarrow [\textit{opUnário}]\ \textit{Primário}\\
\textit{opUnário} &\rightarrow - \mid !\\
\textit{Primário} &\rightarrow \textit{Identificador} \mid \textit{Literal} \mid (\ \textit{Expressão}\ )
\end{aligned}
$$

Primeiro(X) pode ser lido como o conjunto de todos os símbolos terminais que podem ocorrer como símbolos mais à esquerda em uma derivação começando em *X*. Quando *X* for um símbolo terminal, segue-se que:

$$Primeiro(a) = \{a\}$$

Ou seja, o conjunto *Primeiro*(*a*) contém apenas o próprio *a*.

Quando ω for uma *string* arbitrária de terminais e não-terminais $X_1 \ldots X_n U \ldots$, o conjunto *Primeiro(*ω*)* é definido da seguinte forma:

$$Primeiro(X_1 \ldots X_n U \ldots) = Primeiro(X_1) \cup \ldots Primeiro(X_n) \cup Primeiro(U)$$

em que $X_1, \ldots, X_n$ são anuláveis

e U não é anulável.

Um não-terminal é *anulável* se derivar a *string* vazia, ou seja:

$$A \Rightarrow *$$

Uma vez que terminais não derivam algo além de si próprios, não são anuláveis.

Uma primeira observação na gramática EBNF da Figura 3.7 sugere que nenhum dos não-terminais seja anulável, já que nenhuma das alternativas para qualquer não-terminal é vazia. Entretanto, este não é o caso. Para encontrar o conjunto de não-terminais anuláveis, podemos usar o seguinte algoritmo [Knuth, 1971]:

```
int oldSize;
Set nullable = new Set();
do {
    oldSize = nullable.size();
    for (Production p : grammar.productions()) {
          boolean allNull = true;
          for (Symbol t : p.rule())
              if (! nullable.contains(t))
                allNull = false;
          if (allNull)
              nullable.add(p.nonterminal());
    }
} while (nullable.size() > oldSize); // conjunto nullable cresceu
```

Basicamente, o algoritmo itera por todas as regras de produção da gramática. Para cada uma dessas produções `p`, se `p` for uma regra vazia ou se todos os símbolos da regra forem anuláveis, então o não-terminal no lado esquerdo da produção é adicionado ao conjunto de símbolos anuláveis. Este algoritmo seguramente parará, caso contrário, cada iteração de `do while` faria o conjunto `nullable` crescer em um símbolo não-terminal. Entretanto, o conjunto de não-terminais em uma gramática é sempre finito.

Antes de aplicar o algoritmo à gramática da Figura 3.7, reescreveremos a gramática, aplicando as seguintes transformações:

1. Coloque a gramática em uma forma expandida usando $ para designar o final da entrada.
2. Renomeie todos os não-terminais usando abreviações de um ou dois caracteres para facilitar o desenho dos grafos necessários.

3 Abrevie o *token Identificador* como *i* e *o Literal* como *l*.

4 Substitua cada metaconstrução por um não-terminal conforme esboçado no final da Seção 2.2. Lembre-se de que, se *A* for um não-terminal e x, y e z forem seqüências arbitrárias de terminais e não-terminais, então qualquer regra EBNF que tiver metachaves:

$$A \rightarrow x \{ y \} z$$

pode ser reescrita de forma equivalente sem metachaves da seguinte forma:

$$\begin{aligned} A &\rightarrow x A' z \\ A' &\rightarrow \mid y A' \end{aligned}$$

em que A' é um novo não-terminal único.

Por exemplo, aplicando essas transformações à gramática na Figura 3.7 obtemos a gramática da Figura 3.8. Nesta nova gramática, os não-terminais E' e T' representam a iteração nas produções originais *Expressão* e *Termo*, enquanto F' representa o opcional *opUnário* na produção original *Fator*.

Agora, se aplicarmos o algoritmo anulável a esta nova gramática na Figura 3.8, obteremos o seguinte conjunto de não-terminais anuláveis:

Passo	Anulável
1	$E'\ T'\ F'$
2	$E'\ T'\ F'$

O algoritmo descobre todos os não-terminais anuláveis no primeiro passo, mas requer um segundo passo para descobrir que não existem adicionais.

A seguir, definimos o grafo de *dependência à esquerda* de uma gramática G. Cada símbolo terminal ou não-terminal é um nó nesse grafo. Para cada produção da forma:

$$A \rightarrow U_1 \ldots U_n X w$$

Figura 3.8 Gramática de Expressão Reescrita

$$\begin{aligned} S &\rightarrow A\ \$ \\ A &\rightarrow i = E\ ; \\ E &\rightarrow T\ E' \\ E' &\rightarrow \mid AO\ T\ E' \\ AO &\rightarrow + \mid - \\ T &\rightarrow F\ T' \\ T' &\rightarrow \mid MO\ F\ T' \\ MO &\rightarrow * \mid / \\ F &\rightarrow F'\ P \\ F' &\rightarrow \mid UO \\ UO &\rightarrow - \mid ! \\ P &\rightarrow i \mid l \mid (E) \end{aligned}$$

Tabela 3.1

Conjunto Primeiro da Gramática de Expressão

Não-terminal	Primeiro
A	i
E	! – i l (
E′	+ –
AO	+ –
T	– i l (
T′	* /
MO	* /
F	! – i l (
F′	! –
UO	! –
P	i l (

desenhamos um arco de *A* a *X* se $n = 0$ ou $U_1, ..., U_n$ forem anuláveis, ou seja, se não houver nada à esquerda de *X* ou se todos os símbolos à esquerda de *X* forem anuláveis. Aplicar esta regra à Figura 3.8 resulta no grafo de dependência à esquerda da Figura 3.9.

Calcular o conjunto *Primeiro(A)* a partir do grafo de dependência à esquerda agora é direto. *Primeiro(A)* consiste de todos os símbolos terminais no grafo de dependência à esquerda que sejam atingíveis a partir de *A*. Aplicar esse método visualmente ao grafo resulta na coluna rotulada **Primeiro** na Tabela 3.1.

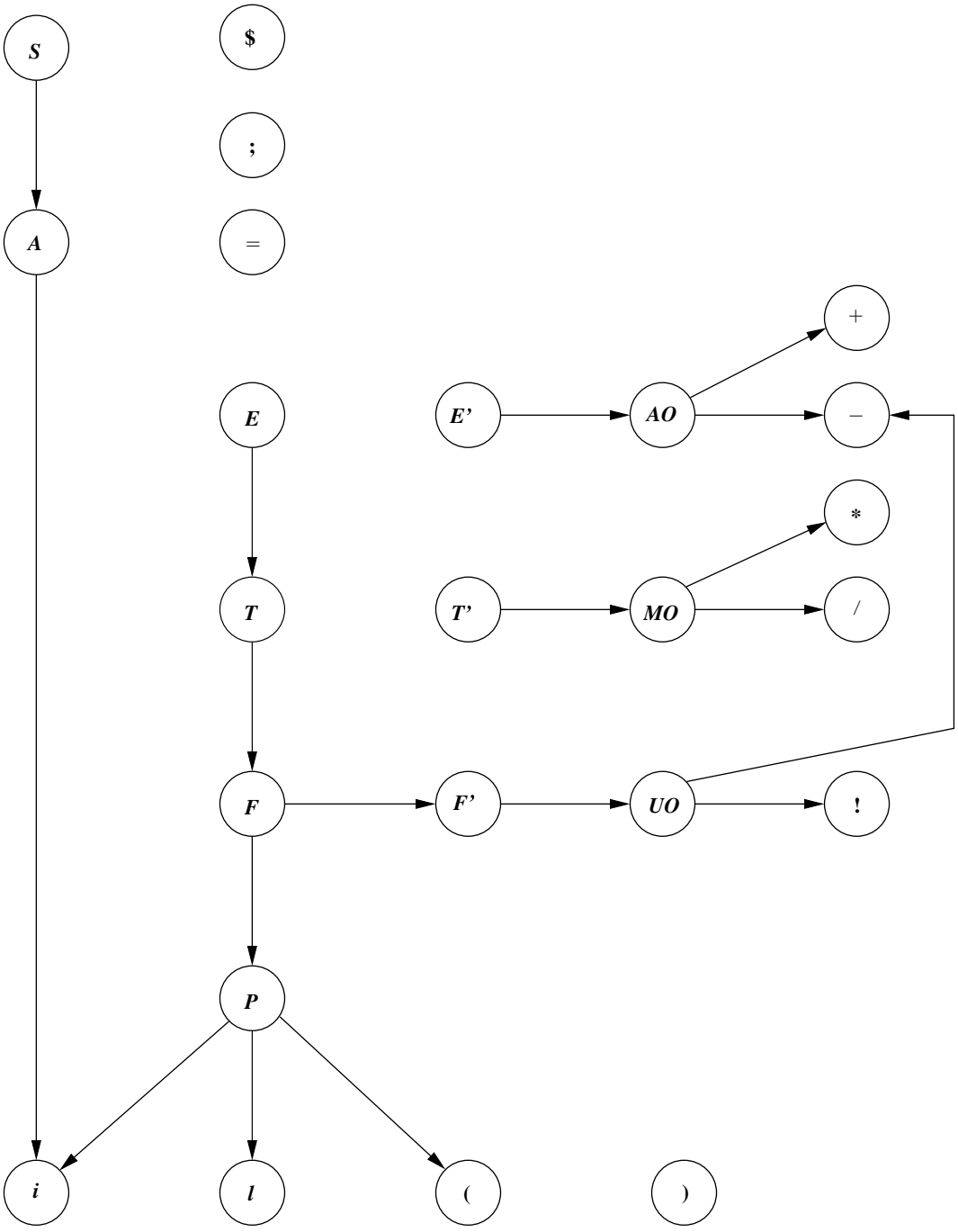
Um algoritmo para calcular todos os nós que sejam atingíveis a partir de um determinado nó pode ser esboçado da seguinte maneira:

```
Set reachable(Node start) {
   Set set = new Set( );
   set.add(start);
   int oldSize;
   do {
      oldSize = set.size();
      for (Node a : set)
         set.add(a.to( ));
   } while (set.size() > oldSize);
   return set;
}
```

3.3.2 Análise Descendente Recursiva

Com esses conceitos iniciais definidos, agora estamos prontos para construir um analisador a partir de uma gramática EBNF. Existem diversos algoritmos diferentes para análise, e cada um possui suas vantagens e desvantagens (veja a Observação na página 81). O algoritmo de análise que apresentamos aqui é chamado de *analisador descendente recursivo*[4].

4. Formalmente, a análise descendente recursiva é uma variante da análise LL(1), que se move da esquerda para a direita no fluxo de entrada e precisa observar à frente apenas um *token* para tomar qualquer decisão de análise. Essa característica é garantida pela gramática em consideração. A maioria das linguagens modernas, como Java, possui gramáticas que permitem análise eficiente.



| Figura 3.9 **Grafo de Dependência à Esquerda para a Gramática de Expressão**

O objetivo de qualquer analisador é construir uma árvore de análise a partir do fluxo de *tokens* produzido pelo *lexer*. Analisadores descendentes recursivos possuem um método correspondente a cada não-terminal da gramática. O trabalho de cada um desses métodos tem duas partes:

1. Reconhecer a seqüência mais longa de *tokens* derivável dela no fluxo de entrada e
2. Construir uma subárvore de sintaxe abstrata e retornar um objeto da classe abstrata que é a raiz dessa subárvore.

Assim, precisamos de um algoritmo para converter regras de produção em código. Se o analisador não conseguir reconhecer a entrada, ou seja, construir implicitamente uma árvore de análise, ele irá parar com uma mensagem de erro que identifica o primeiro erro encontrado.

Cada método do analisador descendente recursivo corresponde a um símbolo não-terminal A da gramática. Esse método é construído diretamente a partir de regra(s) de gramática EBNF da forma $A \rightarrow \omega$, e o nome do método é A. Assim, usando a gramática da Figura 3.7 como exemplo, haverá métodos chamados atribuição, expressão, opAdição, termo, opMult, fator, opUnário e primário. Supondo que A possua uma categoria sintática abstrata correspondente, o método terá um tipo de retorno.

Como exemplo, construiremos um método Java que analisa a entrada da classe sintática concreta *Atribuição* e gera uma árvore sintática abstrata definida pela gramática da Figura 2.11. A regra de sintaxe concreta que governa o projeto desse método (veja a Figura 3.7) é:

Atribuição → *Identificador* = *Expressão*;

Primeiro, construímos um método esqueleto apropriado:

```
private Assignment assignment ( ) {
    ...
    return new Assignment(...);
}
```

Guiado pelo lado direito dessa regra de produção, este método deve conter código que reconheça um *Identificador*, o operador =, uma *Expressão* e o separador ;, exatamente nessa ordem.

Dois métodos auxiliares geralmente são úteis nesse processo: `match` e `error`. A chamada a `match(t)` recupera o próximo *token* ou exibe uma mensagem de erro de sintaxe, dependendo de o *token* corrente corresponder ou não ao *token* `t` esperado:

```
private String match (TokenType t) {
    String value = token.value();
    if (token.type().equals(t))
        token = lexer.next();
    else
        error(t);
    return value;
}
```

A chamada a `error(t)` exibe uma mensagem de erro na tela e encerra o processo de análise:

```
private void error(TokenType tok) {
    System.err.println("Erro de sintaxe: esperando:" + tok
                       + "; encontrado: " + token);
    System.exit(1);
}
```

Na nossa implementação, um *token* consiste de duas partes: um tipo e um valor. O primeiro determina a classe léxica do *token* (por exemplo, *Identificador, Literal*) e o valor é a *string* em questão (por exemplo, `ch`, `1`). Para a maioria dos *tokens*, conhecer seu tipo é suficiente.[5]

Usando esses métodos auxiliares, a tradução da regra para *Atribuição* em um método continua da seguinte maneira:

1. Cada símbolo no lado direito da regra é traduzido em seqüência, resultando em uma seqüência de mais quatro comandos.
2. O primeiro símbolo a ser traduzido é um terminal, de modo que a tradução deve chamar o método `match`, passando para ele um *Identificador*. Além disso, a categoria concreta *Identificador* corresponde à categoria abstrata *Variável* neste contexto, de forma que uma nova variável local `target` deve ser declarada e inicializada com o valor retornado por `match`.
3. Uma tradução semelhante é aplicada ao símbolo terminal =, mas, já que este terminal não possui papel na sintaxe abstrata, nenhuma nova variável local é necessária.
4. O não-terminal *Expressão* é traduzido para uma chamada ao método `expression`, e o resultado é atribuído a uma nova variável local `source` correspondente à categoria abstrata *Expressão.*
5. Finalmente, o símbolo terminal ; é pontuação, logo é tratado como no passo 3 acima.

O método resultante tem a seguinte forma:

```
private Atribuição atribuição ( ) {
    // Atribuição --> Identificador = Expressão;
    Variável target = new Variável(match(Token.Identifier));
    match(Token.Assign);
    Expressão source = expressão();
    match(Token.Semicolon);
    return new Atribuição(target, source);
}
```

Criticamente, este método se baseia em um método `expression`, que recebe o *token* corrente (o que segue o operador =) e analisa o fluxo seguinte de *tokens* para derivar e retornar uma árvore de análise abstrata para *Expressão*. Dada a regra

Expressão → *Termo* { *opAdição Termo* }

5. Entretanto, para o *token Identificador* e os literais inteiro, ponto flutuante e caractere, um valor é associado ao *token*, dando o modo de escrever do identificador, os dígitos que constituem um literal inteiro etc. No *lexer*, há um *token* constante para todos, menos os *tokens* de literal e identificador; o último deve ser criado dinamicamente para cada ocorrência. Este projeto pode ser facilmente traduzido para Ada, C ou C++.

podemos construir o método necessário da seguinte forma:

1. O tipo de retorno é estabelecido como *Expressão* e um comando *return* é inserido.
2. O não-terminal *Termo* é traduzido para uma chamada de método cujo valor de retorno é do tipo *Expressão.*
3. A iteração é traduzida para um laço `while` com o teste assegurando que o *token* corrente pertence ao conjunto *Primeiro(opAdição)*, podendo ser um sinal de mais (+) ou de menos (-). Usamos um método auxiliar *verificaOpAdição* para executar o teste.
4. O corpo do laço é a tradução dos símbolos *opAdição Termo*. Estritamente falando, a tradução do símbolo *opAdição* deve resultar em uma chamada a um método que use um *switch* para determinar se o *token* corrente é um sinal de mais ou de menos. Já que o teste do laço `while` já assegura isso, otimizamos manualmente o código para ser uma chamada a obter o próximo *token*.
5. O não-terminal *Termo* é novamente traduzido para uma chamada de método cujo valor de retorno é do tipo *Expressão*.

Devido à iteração na produção, temos os mesmos problemas de leitura à frente que tivemos na Seção 3.2.3. As soluções, exceto por serem aplicadas a *tokens* em vez de caracteres, também são as mesmas: uma função *peek* (de leitura à frente), uma função *push back* (de envio de volta) ou manter sempre a leitura de um *token* à frente. Já que preferimos a última alternativa, nossa tradução de produções a codificar supõe que, quando o método `expression` for chamado, `token` já possui o valor do primeiro *token* a ser reconhecido, e na saída a variável `token` ficará com o valor do próximo token, seguindo a regra de produção integralmente analisada. Esta conversão é seguida para cada método de análise, exceto a regra de extensão.

O método resultante é mostrado na Figura 3.10. Este método reflete o fato de que uma *Expressão* possui pelo menos um *Termo*, e se o próximo *token* for um sinal de mais ou de menos, então deve ser seguido por outro *Termo*. Já que pode haver qualquer número de ocorrências de uma *opAdição* e um *Termo*, a iteração EBNF é traduzida para um laço. No final, o método `expression` reconhece a seqüência mais longa de *Termos* separados por sinais de mais e de menos.

Esse método também se baseia na existência de um método `term`, cuja construção é semelhante àquela de `expression`. Os métodos para as categorias sintáticas remanescentes *Termo*, *Fator* e *Primário* para essa gramática também aparecem na Figura 3.10.

Considere a análise da entrada `z=x+2*y`; como uma *Atribuição* e a geração da árvore de sintaxe abstrata mostrada na Figura 2.13. Primeiro, o método `assignment` é chamado durante a varredura do *Identificador* `z`. Isso corresponde à construção da raiz de uma árvore de análise abstrata *Atribuição*. O primeiro comando visa a reconhecer um *Identificador*, que corresponde a construir a subárvore mais à esquerda da *Atribuição* na Figura 2.13. A seguir, o operador = é reconhecido, mas não é retido na árvore abstrata porque não é mais necessário. O terceiro comando chama o método `expression`, que cria a subárvore abstrata *Expressão* para *Atribuição*. O método `expression` busca um *token Identificador*; assim ele chama `term`, que chama `factor`, que chama `primary`, que corresponde ao *token*. Tudo isso corresponde a desenhar a subárvore esquerda de *Expressão*. Terminar a pilha de chamadas de `primary`, `factor` e `term` efetivamente conclui a criação das subárvores remanescentes da Figura 2.13.

A Figura 3.11 resume um algoritmo chamado de algoritmo T, para traduzir uma produção em código. Devido a cada iteração de uma regra de produção ser traduzida para um laço *while*, o analisador ficará no laço até ter avançado um *token* a mais. Este problema é idêntico ao problema do laço no DFSA da Seção 3.2.2 e admite as mesmas soluções. O algoritmo T

```
private Expressão expressão () {
     // Expressão --> Termo { opAtrib Termo }
     Expressão e = term();
     while (isAddOp()) {
          Operator op = new Operator(match(token.type()));
          Expressão term2 = term();
          e = new Binary(op, e, term2);
     }
     return e;
}

private Expressão term () {
     // Termo --> Fator { opMult Fator }
     Expressão e = factor();
     while (isMultiplyOp()) {
          Operator op = new Operator(match(token.type()));
          Expressão term2 = factor();
          e = new Binary(op, e, term2);
     }
     return e;
}

private Expressão factor() {
     // Fator --> [ opUnário ] Primário
     if (isUnaryOp()) {
          Operator op = new Operator(match(token.type()));
          Expressão term = primary();
          return new Unary(op, term);
     }
     else return primary();
}

private Expressão primary () {
     // Primário --> Identificador | Literal | ( Expressão )
     Expressão e = null;
     if (token.type().equals(TokenType.Identifier)) {
          e = new Variable(match(TokenType.Identifier));
     } else if (isLiteral()) {
          e = literal();
     } else if (token.type().equals(TokenType.LeftParen)) {
          token = lexer.next();
          e = expression();
          match(TokenType.RightParen);
     } else error(“Identificador | Literal | (”);
     return e;
}
```

| Figura 3.10 Analisador Descendente Recursivo para Expressões

1. Nomeie o método *A*. Se *A* tiver uma categoria sintática abstrata *A'* correspondente, faça o tipo de retorno do método ser *A'* e adicione um comando `return new A'` de modo que assegure que todos os campos de *A'* sejam inicializados apropriadamente.
2. Se ω for o *y* terminal, verifique se o *token* corrente *t* é uma instância de *y* e então obtenha o próximo *token*. Caso contrário, emita uma mensagem de erro e pare (o *token* está errado). Se *y* corresponder a um campo na definição de sintaxe abstrata de *A'*, atribua seu valor a esse campo.
3. Se ω for um não-terminal *B*, chame o método *B* e atribua o valor retornado por *B* a uma variável `new` na categoria sintática abstrata *B'*.
4. Se ω for uma iteração com metachaves $\{\omega'\}$, traduza para um laço *while*. O corpo do laço *while* é T aplicado recursivamente ao conteúdo da iteração ω'. Se *t* é o *token* corrente, o teste do laço é $t \in Primeiro\ (\omega')$; como na Seção 3.2, existem várias formas para se implementar tais testes.
5. Se ω for um conjunto de alternativas $\omega_1 | \ldots | \omega_n$, com ou sem metaparênteses, traduza para um comando *switch* (ou uma seqüência de if/else-if) de acordo com o token *t* corrente. Aplique recursivamente T a cada ω_i, em que o teste do *switch* corresponde a $t \in Primeiro(\omega_i)$. Se nenhuma das alternativas for vazia, então o *case default* deve ser de que o *token* é inválido (erro); caso contrário, o *case default* é vazio.
6. Se ω for uma escolha opcional com metacolchetes $[\omega']$, pode ser reescrito como $(\omega' |\ )$ e traduzido usando a regra anterior.
7. Se ω estiver na forma de $X_1 \ldots X_n$, aplique T recursivamente a cada X_i, resultando na seqüência de comandos: $T(X_1); \ldots; T(X_n)$.

| Figura 3.11 O Algoritmo T para Implementar a Regra $A \rightarrow \omega$ no Método *A*

supõe que no reconhecimento de um não-terminal *A*, o método *A* começa encontrando o primeiro *token* que seja derivável de *A* e termina encontrando o primeiro *token* que não corresponda a uma *string* derivável de *A*. O método *A* corresponde assim a *string* mais longa derivável do não-terminal *A*.

O algoritmo T é aplicado a todas as produções, exceto à produção de extensão. Esta última é responsável por:

1. Obter o primeiro *token*.
2. Chamar o método correspondente ao símbolo inicial original da gramática.
3. Verificar se todas as informações (*tokens*) foram consumidas, mas **não** buscar um *token* após o final do arquivo (veja a Seção 3.2.3).

A análise descendente recursiva requer que a gramática não seja recursiva à esquerda, ou seja, não exista um *A* não-terminal tal que $A \Rightarrow^{+} A\ \ldots$.[6] Caso contrário, o analisador pode entrar em um laço recursivo infinito porque, sob alguma condição de entrada, o método *A* pode chamar a si próprio sem consumir alguma entrada; tendo feito isso uma vez, ele continuará a fazê-lo indefinidamente. Este é outro motivo pelo qual gramáticas EBNF são preferidas, já que a recursão direta à esquerda sempre pode ser substituída pela iteração.[7]

6. O não-terminal *A* deriva a si próprio como o símbolo mais à esquerda, usando uma ou mais derivações. Na notação Kleene-star, um sinal de mais significa um ou mais do que o símbolo à sua esquerda.

7. Felizmente, há uma verificação simples para a recursão à esquerda, a saber, a presença de um ciclo no grafo de dependência à esquerda. Isso significa que todos os não-terminais do ciclo são recursivos à esquerda. Para a nossa gramática de exemplo, nenhum ciclo está presente na Figura 3.9.

Observação

Analisadores Orientados a Tabela

Para implementar um compilador ou tradutor para uma linguagem de programação como C, C++, Java ou Ada, normalmente é usado um gerador de analisador como o yacc/bison ou o JavaCC, embora alguns compiladores C (Waite e Carter, 1993) usem um descendente recursivo. A maioria dos sistemas geradores de analisadores produzem analisadores orientados a tabela.

LL(1) é um analisador *top-down* que é basicamente uma versão orientada a tabela de um analisador descendente recursivo, em que o 1 indica a quantidade de leituras à frente, ou seja, 1 *token*. Analisadores orientados a tabela são preferidos por gramáticas grandes porque o algoritmo é fixo e apenas o tamanho da tabela cresce em função do tamanho da gramática. O primeiro "L" em *LL(1)* indica que a entrada é lida e processada da esquerda para a direita, enquanto o segundo "L" indica que a árvore de análise é construída usando a derivação mais à esquerda.

Utilizando a gramática da Figura 3.8 e numerando cada produção alternativa começando do zero, obtemos a seguinte tabela de análise LL(1):

N	id	lit	+	-	*	/	!	(	)	;
A	1									
E	2	2						2		
E'			4	4					3	3
AO			5	6						
T	7	7						7		
T'			8	8	9	9			8	8
MO					10	11				
F	12	12						12		
F'	13	13		14			14	13		
UO				15			16			
P	17	18						19		

O algoritmo de análise LL(1) inicialmente coloca o lado direito da regra estendida (por exemplo, *A*$) na pilha, de modo que *A* fique no topo. A seguir ele repete os passos abaixo, em que *X* é o topo corrente da pilha e *a*, o *token* de entrada corrente.

1. Se *X* e *a* forem ambos o símbolo final $: pare e aceite a entrada como sendo válida.
2. Se *X* for um símbolo terminal: se *X* for igual a *a*, desempilhe *X* e obtenha o próximo *token*; caso contrário, pare, porque ocorreu um erro.
3. Se *X* contiver um não-terminal: se a posição da tabela [*X, a*] estiver vazia, pare porque ocorreu um erro; caso contrário, ele deve conter um número de produção *p*; empilhe os símbolos da produção da direita para a esquerda de modo que o símbolo mais à esquerda fique no topo.

O algoritmo de análise mais amplamente usado é o LR(1), que constrói a árvore de análise de baixo para cima, em vez de construí-la de cima para baixo, como é o caso do LL(1). Por causa disso, um analisador LR(1) é capaz de lidar com gramáticas recursivas à esquerda, como a gramática G_1 da Seção 2.1.4. Além disso, é um teorema que cada gramática LL(1) também é uma LR(1). Para mais informações sobre analisadores LR(1), consulte um texto sobre compiladores como (Aho et al., 1986).

3.4 RESUMO

Este capítulo apresentou um breve exame da hierarquia de Chomsky de linguagens formais, demonstrando gramáticas regulares como base para a construção de um analisador léxico.

A seguir, mostramos um exame aprofundado do projeto e da implementação de um analisador léxico para o Clite. Primeiramente, foram discutidos o conceito de *token* e outras tarefas de um analisador léxico. Foram usadas expressões regulares como uma forma alternativa de especificação de *tokens*. Para projetar manualmente um *lexer*, foram apresentados autômatos de estados finitos determinísticos (DFSA). Finalmente, foi dada a tradução de um DFSA para código.

Na última seção deste capítulo, a construção de um analisador descendente recursivo foi demonstrada. Diversos algoritmos e algumas definições auxiliares foram apresentados, incluindo a extensão de uma gramática, o cálculo do conjunto de não-terminais anuláveis e o cálculo do conjunto de terminais mais à esquerda deriváveis de cada não-terminal A (denominado Primeiro(A)). Finalmente, foi exibido um método para traduzir cada produção da forma $A \rightarrow \omega$ para um método A para analisar ω.

EXERCÍCIOS

3.1 Reescreva as produções para cada um dos seguintes não-terminais como gramáticas regulares à direita: *Identificador, Ponto Flutuante*.

3.2 Reescreva as produções para cada um dos seguintes não-terminais como gramáticas regulares à esquerda: *Identificador, Ponto Flutuante*.

3.3 Desenhe um DFSA que contenha apenas letras e dígitos, em que o identificador deva ter pelo menos uma letra, mas que ela não precise ser o primeiro caractere. Dica: tudo à esquerda da letra mais à esquerda deve ser um dígito.

3.4 Tente definir a linguagem $\{a^n b^n\}$ usando uma gramática regular. Discuta o porquê de isso talvez não ser possível.

3.5 Tente definir a linguagem $\{a^n b^n\}$ usando um DFSA. Discuta o porquê de isso talvez não ser possível.

3.6 Determine por que linguagens mais antigas como Fortran e Cobol não diferenciam letras maiúsculas de minúsculas.

3.7 Que partes da linguagem PHP diferenciam letras maiúsculas de minúsculas? Que partes não fazem isso? Você consegue encontrar um motivo para a separação?

3.8 Desenvolva um conjunto de testes completo para `Lexer`. Explique o propósito de cada caso de teste.

3.9 Mostre as transições feitas usando o DFSA para identificadores apresentados na Seção 3.2.2 no recebimento do seguinte: (a) a, (b) a2, (c) a2i, (d) abc.

3.10 Para números de ponto flutuante em notação científica, apresente: (a) uma gramática regular à direita; (b) uma expressão regular; (c) um DFSA. Dê exemplos de números que sejam permitidos e não permitidos.

3.11 Estenda o *lexer* para incluir números de ponto flutuante em notação científica usando seu projeto do exercício anterior.

3.12 Para números em bases da forma `base # número #` (sem espaço em branco interno) apresente: (a) uma gramática regular à direita; (b) uma expressão regular; (c) um DFSA. A `base` deve ser expressa em decimais; o número deve usar letras maiúsculas ou minúsculas para representar dígitos maiores que 9. A sua definição assegura consistência do número com respeito à base, por exemplo, `8#99#` é proibido?

3.13 Para comentários no estilo da linguagem C `/* ... */`, dê: (a) uma gramática regular à direita; (b) uma expressão regular; (c) um DFSA.

3.14 Implemente no *lexer* seu projeto de comentários no estilo da linguagem C `/* ... */`.

3.15 Projete e implemente uma parte de um *lexer* que lide com seqüências arbitrárias do seguinte: (a) inteiros na forma: d^+ (em que d é um dígito); (b) números de ponto flutuante na forma: $d^+.d^+$; (c) operador de subfaixa (...). Assim, a seguinte *string* consiste de três *tokens*: 10.81, a saber, um inteiro 10, o operador de subfaixa e outro inteiro 81.

3.16 Analise a seguinte gramática:

$$S \rightarrow \mid a \mid (\, T\,)$$
$$T \rightarrow T, S \mid S$$

Após estender a gramática:
(a) Desenhe o grafo de dependência à esquerda.
(b) Calcule o conjunto *Primeiro* para cada não-terminal.

3.17 Acrescente semântica no analisador descendente recursivo que calcule o valor de cada expressão e imprima esse valor. Para implementar variáveis, restrinja-as a letras minúsculas únicas e use uma matriz com a variável usada como índice; inicialize a matriz com zero.

3.18 Melhore o analisador Clite encontrado no site do livro, adicionando um método:

```
private boolean matches (int[] tok)
```

que retorne verdadeiro se `token.type( )` corresponder a um dos `tok[]`, e falso, caso contrário. O método não deve avançar a leitura de *tokens*. Use esse método para eliminar `isAddOp( )`, `isMulOp( )` etc.

3.19 Uma linguagem pode não ter palavras reservadas? Ou seja, suponha que cada palavra reservada (como `if` e `for`) fosse meramente um identificador predefinido, que o programador estivesse livre para redefinir. Tal linguagem pode existir? Explique.

3.20 Adicione um método `display` a cada uma das classes de sintaxe abstrata da Figura 2.11 de modo que uma árvore de sintaxe abstrata para *Atribuição* possa ser exibida de maneira estruturada. Por exemplo, a árvore de sintaxe abstrata para a *Atribuição* $z = x + 2 * y$; deveria ser exibida da seguinte forma:

```
=
    z
    +
        x
        *
            2
            y
```

3.21 Projete uma sintaxe abstrata e implemente um analisador descendente recursivo para que as expressões sejam simbolicamente diferenciadas. A gramática é uma gramática polonesa prefixada:

$$
\begin{aligned}
\textit{Expr} &\rightarrow \textit{Op Expr Expr} \mid \textit{Primária} \\
\textit{Op} &\rightarrow + \mid - \mid * \mid / \\
\textit{Primário} &\rightarrow \textit{Inteiro} \mid \textit{Letra}
\end{aligned}
$$

na qual um *Inteiro* é uma seqüência arbitrária de dígitos e uma *Letra* é um único caractere minúsculo que representa uma variável matemática. Para simplificar a atribuição, você pode supor que a *Expr* é livre de erros e que a *Expr* aparece em uma linha com *tokens* separados por um ou mais espaços ou tabulações. O programa deve exibir a árvore de sintaxe abstrata.

4 Nomes

"O primeiro passo na direção da sabedoria é chamar as coisas pelos nomes certos."

Provérbio chinês

VISÃO GERAL DO CAPÍTULO

A nomenclatura eficaz de variáveis, funções, tipos, classes e outras entidades é uma importante habilidade de programação. Compreender os diversos usos e as implicações de se referenciar um nome em diferentes contextos, muitas vezes, não é uma tarefa simples.

Lembre-se do que citamos no Capítulo 1, que o termo *ligação* (*binding*) é uma associação entre uma entidade (como uma variável) e uma propriedade (como seu valor).

Definição: Uma ligação é *estática* se a associação ocorrer antes da execução.

Definição: Uma ligação é *dinâmica* se a associação ocorrer em tempo de execução.

O momento em que uma ligação ocorre com um nome desempenha um papel fundamental. Por exemplo, o escopo estático, no qual um nome é ligado a uma declaração específica antes da execução, deve ser familiar à maioria dos programadores já que é o mecanismo usado por todas as principais linguagens para resolver referências. Contudo, algumas linguagens atrasam a resolução do nome até o tempo de execução. Para entendermos tanto o escopo estático quanto o dinâmico, examinamos a noção de uma tabela de símbolos e seu uso em tempo de compilação e de execução para implementar escopo estático e dinâmico, respectivamente.

Além da resolução de nomes, também consideramos a visibilidade destes. Em algumas linguagens, a redeclaração dentro de um escopo de um nome pode esconder outras instâncias do mesmo nome. A sobrecarga permite que diferentes instâncias do mesmo nome de uma função ou de um operador sejam resolvidas com base no número ou nos tipos dos seus argumentos.

O tempo de vida do nome de uma variável se refere ao intervalo de tempo durante o qual a variável fica alocada na memória, o que pode ocorrer estática ou dinamicamente. Neste último caso, o tempo de vida de uma variável deve ser a execução dos comandos pertencentes ao seu escopo. Algumas linguagens fornecem mecanismos para estender o tempo de vida de tais variáveis.

Iniciando neste capítulo, usamos o termo linguagem *como C* para incluir todas as versões de C, C++ e Java e linguagem *como Pascal* para incluir tanto Pascal quanto Ada.

4.1 QUESTÕES SINTÁTICAS

Nomes são usados em programas para denotar muitas entidades diferentes: variáveis, tipos, funções etc. Nesta seção, analisamos a sintaxe básica dos nomes. Outro termo para nome é *Identificador*.

Regras léxicas rígidas determinam como um nome pode ser construído. Por exemplo, vimos no Capítulo 2 que um *Identificador* Clite deve ser uma série de letras e dígitos, começando com uma letra. Esta é uma regra comum para criar nomes na maioria das linguagens de programação.

As primeiras linguagens só usavam letras maiúsculas e colocavam limites estritos sobre o comprimento de um identificador (veja a Observação). Linguagens posteriores, como Pascal e Ada, continuaram a tradição de não diferenciar maiúsculas de minúsculas, de modo que qualquer um dos nomes `alpha`, `Alpha` ou `alPha` se referem à mesma entidade. Em contraste, nomes em linguagens como C diferenciam maiúsculas de minúsculas, de modo que, por exemplo, `alpha` e `Alpha` não são nomes da mesma entidade. A linguagem de scripting web PHP diferencia parcialmente maiúsculas de minúsculas, o que também é uma decisão de projeto questionável.

Muitas linguagens permitem o uso de um ou mais caracteres especiais em identificadores, mas algumas restringem seu uso; por exemplo, Cobol permite o caractere de hífen, mas um identificador não pode começar ou terminar com um hífen. Outras, como as linguagens como C, não apenas permitem um caractere especial (o sublinhado _), como não restringem o seu uso.

> **Definição:** A maioria das linguagens possui um conjunto predefinido de nomes chamado *palavras reservadas* ou *palavras-chave* que possuem um significado especial e não podem ser usados como identificadores.

Palavras reservadas incluem nomes que são úteis na análise de um programa porque identificam construções importantes. Nas linguagens como C, estas palavras-chave incluem `int`, `if` e `while`.

Observação

Nomenclatura em Fortran

Fortran, a primeira linguagem de programação de alto nível, originalmente usava apenas letras maiúsculas simplesmente porque os dispositivos de entrada primários (a máquina de teclas) e as impressoras de linha só usavam maiúsculas. Fortran foi projetada e implementada para computadores IBM 709/704 que possuíam um caractere de 6 bits (chamado BCD) e uma palavra de 36 bits. Assim, os projetistas de Fortran limitaram o tamanho de um identificador para um máximo de 6 caracteres, armazenados e justificados à esquerda com espaços de preenchimento, já que o 709/704 não tinha a habilidade de endereçar caracteres individuais.

Estas convenções foram padronizadas no Fortran 66 e 77 e mantidas inalteradas até o Fortran 90. Todavia, naquele momento, a maioria dos computadores havia migrado para ASCII (ou, no caso dos mainframes IBM, EBCDIC), que permitia caracteres em letras minúsculas. A convenção mais simples parecia ser, no caso do Fortran, não distinguir maiúsculas de minúsculas em nomes. De outra forma, todas as palavras reservadas teriam de ser digitadas em maiúsculas, para manter a compatibilidade para trás. Fortran 90 padronizou esta decisão.

Em algumas linguagens (por exemplo, Pascal e Scheme), além de palavras reservadas, existem *identificadores predefinidos* que possuem um significado especial; diferentemente das palavras reservadas, o programador está livre para redefini-los. Nomes de funções de bibliotecas geralmente caem nessa categoria. Tal característica pode ser uma bênção ou uma maldição. Por um lado, isso ajuda a minimizar o número de palavras reservadas de uma linguagem. Por outro lado, pode levar a programas confusos. Por exemplo, analise o seguinte fragmento de programa válido em Pascal, que redefine o significado do identificador predefinido `true`:

```
program confuso;
const true = false;
begin
   ...
   if (a < b) = true then
      f(a)
   else
      g(b);
   ...
end.
```

Neste caso, o procedimento `f` é chamado se `a >= b`, o oposto do que o código parece dizer. Permitir a redefinição de um *identificador* importante como `true` é, dessa forma, uma decisão ruim de projeto de linguagem.

Um projetista de linguagem deve decidir entre tornar identificadores predefinidos reservados ou permitir que o programador redefina tais identificadores. Cobol usou a primeira abordagem, com o resultado de que há centenas de palavras reservadas. A maioria das linguagens modernas usa alguma forma da segunda abordagem, minimizando assim o número de palavras reservadas. Linguagens como Pascal permitem que os nomes de tipos básicos como *integer* sejam redefinidos, enquanto as linguagens como C os tornam reservados.

4.2 VARIÁVEIS

Um uso importante da nomenclatura é dar nomes a variáveis, que são fundamentais na programação imperativa e orientada a objeto. Em tais linguagens, uma variável é um nome para um local de memória (ou bloco de locais).

> **Definição:** Uma *variável* é uma ligação de um nome com um endereço de memória. Além disso, uma variável possui um tipo, um valor e um tempo de vida.

Qualquer uma dessas ou outras ligações pode ser estática ou dinâmica, criando interessantes variações entre as linguagens. Essas características de variáveis também se relacionam com a forma pela qual uma linguagem é implementada. Por exemplo, se uma linguagem requerer que todos os nomes de variáveis sejam declarados antes de serem referenciados em um programa, então a fase de análise semântica de um compilador deve incluir uma lista de nomes declarados e seus tipos de dados associados; esta lista, muitas vezes, é denominada *tabela de símbolos*, como será mostrado na Seção 4.4.

Os programadores usam variáveis para dar nomes a locais de memória individuais e agregados; pelo restante deste capítulo, trataremos os dois como equivalentes. Uma variável de programa possui quatro ligações básicas:

- Nome.
- Endereço (que é uma questão de implementação).
- Tipo.
- Valor.

Estes itens podem ser ligados estática ou dinamicamente. Os diversos tipos de valores que podem ser atribuídos a variáveis são discutidos no Capítulo 5. O endereço de uma variável identifica de modo único o local da memória onde o valor de uma variável é armazenado. O endereçamento do valor de uma variável também é introduzido no Capítulo 5, no qual a forte associação entre um ponteiro e um endereço é estabelecida.

Algol 68 (van Wijngaarden, 1969) foi a primeira linguagem a distinguir claramente entre o uso do nome de uma variável para denotar seu endereço, o *valor-l* da variável, *versus* o uso do seu nome para denotar o próprio valor, o *valor-r* da variável. Analise o seguinte comando:

```
x = y + 1;
```

Tal comando deve ser lido: atribua ao endereço de memória denotado pela variável `x` o valor da expressão que é a soma do valor da variável `y` e um. Observe que, quando usado como um valor no lado esquerdo (valor-l), o identificador da variável `x` denota um endereço; no entanto, se usado no lado direito (valor-r), o identificador da variável `y` denota o valor armazenado no endereço.

Algol 68 permitia que o programa distinguisse explicitamente o endereço de uma variável do seu valor. Uma variável do tipo `int` foi denotada como um `ref int`, ou seja, uma referência a um inteiro. O que C++ chama de uma `const` (constante) foi denotado como um `int` em Algol 68, conceitualmente o mesmo que `int` l.

ML [Ullman, 1998] também suporta tal *desreferenciação explícita* de uma variável em um programa. Por exemplo, a expressão ML:

```
x := !y + 1;
```

usa o operador ! para converter a referência y em um valor. A maioria das linguagens não requer desreferenciação explícita para variáveis comuns em comandos de atribuição. Todavia, muitos compiladores usam uma representação de sintaxe abstrata aproximando o comando ML anterior com operadores de desreferenciação explícita inseridos.

Entretanto, para variáveis ponteiros, a desreferenciação explícita muitas vezes é útil. C/C++ usam a estrela unária * para desreferenciar uma variável ponteiro. Por exemplo, no fragmento de código:

```
int x, y;
int *p;
x = *p;
*p = y;
```

o terceiro comando atribui ao valor-l de x o valor do endereço de memória referenciado por p, ou seja, o valor-r do valor-r de p. A última linha atribui ao local de memória referenciado por p o valor-r de y. Observe que o próprio valor de p, um local de memória, permanece inalterado.

4.3 ESCOPO

Os primeiros computadores tinham memórias muito limitadas em comparação aos padrões atuais, de alguns kilobytes (milhares de bytes) a algumas centenas de kilobytes. À medida que os programas têm crescido em tamanho, a colisão de nomes tem se tornado uma questão cada vez mais importante. Para permitir que os programadores reusem o mesmo identificador dentro de um programa, é usado o conceito do *escopo* de um nome.

> **Definição:** O *escopo* de um nome é a coleção de comandos que podem acessar essa ligação de nome.

Da mesma forma que com a maioria das questões, a ligação de um nome a um escopo pode ser estática ou dinâmica, com esta última sendo analisada na Seção 4.6.

> **Definição:** No *escopo estático*, o nome é ligado a uma coleção de comandos de acordo com sua posição no programa-fonte.

Assim, o escopo estático pode ser executado em tempo de compilação e é independente da história de execução do programa. Devido à ligação do nome poder ser determinada por uma simples varredura do programa, o escopo estático melhora a legibilidade e permite um melhor nível de verificação em tempo de compilação.

A maioria das linguagens modernas, incluindo C/C++, Java, Python e Ada, usa escopo estático. Devido ao escopo estático ser baseado na estrutura gramatical de um programa, é às vezes chamado de *escopo léxico*. As estruturas gramaticais que constituem um escopo variam conforme a linguagem; antes de examinarmos esta questão, definiremos alguma terminologia.

Na maioria das linguagens, os escopos podem ser aninhados. Dois escopos diferentes são *disjuntos* ou *aninhados*. Quando dois escopos são disjuntos, o mesmo nome pode ser ligado a diferentes entidades sem qualquer interferência.

O caso mais simples do que constitui um escopo consiste das primeiras versões de Fortran, nas quais uma unidade de compilação consistia de um único escopo, que não podia ser aninhado. Uma unidade de compilação Fortran consistia do programa principal ou de uma única função ou procedimento, e assim todos os escopos não possuíam intersecção; nenhum podia ser aninhado.

Entretanto, de certo modo, essa é uma visão simplista. Todas as linguagens suportam um escopo mais externo, no qual determinados métodos ou bibliotecas existem e podem ser acessados; o que fica neste escopo e como as entidades podem ser

Tabela 4.1

O que Constitui um Escopo

	Algol	C	Java	Ada
Pacote	n/a	n/a	sim	sim
Classe	n/a	n/a	aninhado	sim
Função	aninhado	sim	sim	aninhado
Bloco	aninhado	aninhado	aninhado	aninhado
Laço For	não	não	sim	automático

acessadas varia com a linguagem. Fortran e C exportam funções e procedimentos para esse escopo mais externo.

Saindo de Fortran, uma unidade de compilação Algol 60 constituía um escopo dentro do qual funções e procedimentos introduziam novos escopos e podiam ser aninhados. Além disso, um comando composto, delimitado com as palavras reservadas `begin` e `end` constitui um escopo, e estas podiam ser aninhadas também. Um comando composto com declarações é denominado um *bloco* em Algol, e tais declarações podem incluir variáveis ou funções. Entretanto, em um escopo de Algol 60 todas as declarações devem preceder todos os outros comandos.

Linguagens mais recentes têm incorporado algumas das convenções de escopo de Algol 60. Por exemplo, linguagens como C seguem a tradição de Algol de permitir um comando composto (como um bloco, exceto que delimitado por chaves em vez de `begin` e `end`) que contenha novas declarações e, assim, introduza um novo escopo. Contudo, C/C++ não permitem funções aninhadas.

Uma tentativa de resumir a questão do que constitui um escopo para diversas linguagens importantes é apresentada na Tabela 4.1. Na tabela, a unidade de compilação é deixada de fora, já que sempre constitui um escopo. Além disso, a tabela não é para ser definitiva, no sentido que nem todas as entidades gramaticais que constituem um escopo para uma determinada linguagem aparecem na tabela. Em vez disso, ela deve ser vista como uma tentativa de apresentar o "quadro geral".

Analise a coluna da linguagem C; ela não suporta nem pacotes nem classes, de modo que a questão não é aplicável (n/a). Funções constituem um escopo, mas não podem ser aninhadas. Blocos (listas de comandos dentro de chaves) constituem um escopo e podem ser aninhados. Finalmente, uma variável de laço *for* não pode ser declarada como parte deste comando *for*; em contraste, a variável de laço *for* é declarada automaticamente em Ada e seu escopo se estende apenas ao cabeçalho e ao corpo do laço.

> **Definição:** O escopo no qual um nome é definido ou declarado é chamado de seu *escopo de definição*.

> **Definição:** Uma referência a um nome é *não-local* se ocorrer em um escopo aninhado do escopo de definição; caso contrário, a referência é *local*.

Analise a função simples de ordenação em C apresentada na Figura 4.1. O escopo da variável temporária `t` inclui as linhas 6–9, que também são o seu escopo de definição. Já que não há blocos aninhados dentro das linhas 6–9, todas as referências a `t` são locais. Assim, `t` só pode ser referenciada pelas declarações que estão dentro das chaves nas linhas 5 e 10. Este uso de um comando composto para declarar uma variável melhora a legibilidade do programa, já que o escopo da variável é limitado.

C também especifica que todas as declarações devem preceder todos os outros tipos de comandos em um bloco; assim, uma referência pode ocorrer apenas após o nome ser declarado, ou seja, o código não pode conter uma referência *forward* a um nome declarado. C++ e Java permitem que uma declaração apareça em qualquer lugar dentro

```
1  void sort (float a[ ], int size) {
2     int i, j;
3     for (i = 0; i < size; i++)
4        for (j = i + 1; j < size; j++)
5           if (a[j] < ati]) {
6              float t;
7              t = a[i];
8              a[i] = a[j];
9              a[jj = t;
10          }
11 }
```

| Figura 4.1 Exemplo de Escopos em C

de um bloco, desde que a regra de *nenhuma referência forward* seja violada. Uma *referência forward* é uma referência a um nome que ocorre antes que esse nome tenha sido declarado.

Continuando com o exemplo da Figura 4.1, o escopo para `a` e `size` são as linhas 2–10, ou seja, tudo entre as chaves nas linhas 1 e 11. De forma semelhante, o escopo para `i` e `j` é o mesmo bloco, porém referências a um ou outro nome devem ocorrer após a declaração na linha 2. As referências a `i` nas linhas 3–5 e a `j` nas linhas 4–5 também são locais. Contudo, as referências a estes nomes nas linhas 7–9 são não-locais, já que as linhas 6–10 são um escopo aninhado dentro do escopo de definição para `i` e `j`. Esta questão importante é abordada mais integralmente na Seção 4.5.

Um tipo de escopo interessante é o comando `for`, como em C++/Java, em que uma nova variável de controle pode ser declarada. Neste caso, o escopo dessa variável fica limitado ao corpo do laço:

```
for (int i = 0; i < 10; i++) {
   System.out.println(i);
   ...
{
... i ... // referência inválida a i
```

Aqui, a referência a `i` seguindo o corpo do laço `for` é inválida, já que o escopo da variável `i` é limitado ao corpo do laço. Ada também suporta essa capacidade; contudo, em Ada a variável de controle do laço `for` é implícita e automaticamente declarada.

O escopo de classe Java é interessante porque é uma exceção à regra de *referência forward*. Na declaração de uma classe, as variáveis e os métodos de instância podem ser declarados em qualquer ordem conveniente. Embora a maioria dos textos sobre Java declarem todas as variáveis de instância antes dos métodos (seguindo a tradição Algol/Pascal), alguns textos seguem a convenção oposta. O que é mais importante é que os métodos podem aparecer em qualquer ordem, independentemente de quais métodos se referem a outros métodos; isso melhora bastante a capacidade de escrita.

Até aqui, evitamos o problema de resolver referências a nomes onde existam múltiplas declarações do mesmo nome. Para abordarmos esta questão de forma apropriada, primeiro examinamos o papel e a implementação da tabela de símbolos de um compilador.

4.4 TABELA DE SÍMBOLOS

Uma das tarefas da fase de análise semântica de um tradutor (veja a Seção 2.4) é construir uma *tabela de símbolos* com todos os nomes declarados e suas associações.

> **Definição:** Uma *tabela de símbolos* é uma estrutura de dados mantida pelo tradutor que permite a ele manter registro de cada nome declarado e suas associações.

Pelo restante deste capítulo, supomos que cada nome declarado seja único dentro do seu escopo local, ou seja, as únicas outras declarações do mesmo nome ocorrem em um escopo disjunto ou em um escopo aninhado. Esta suposição importante será relatada na Seção 4.8.

A estrutura de dados usada para um escopo local pode ser qualquer implementação de um dicionário ou conjunto, em que o nome é a chave. Dado um nome, consultamos no dicionário a associação ligada a ele. Ele retorna uma associação apenas se o nome tiver sido declarado no escopo corrente; caso contrário, retorna um valor especial indicando que nenhuma declaração de nome–associação ocorreu no escopo local.[1]

Tudo o que precisamos agora é um algoritmo para lidar com referências não-locais válidas. O fato de que tais referências podem ocorrer em um escopo que esteja aninhado no escopo de definição sugere o uso de uma pilha de escopos, da seguinte forma:

1. Cada vez que se entra em um escopo, adicione um novo dicionário ao topo da pilha.
2. Cada vez que se sair de um escopo, retire um dicionário do topo da pilha.
3. Para cada nome declarado, gere uma associação apropriada e acrescente o par nome–associação ao dicionário no topo da pilha.
4. Dada uma referência a um nome, pesquise sobre este no dicionário do topo da pilha.
   (a) Se encontrado, retorne à associação.
   (b) Caso contrário, repita o processo no próximo dicionário da pilha até que o nome seja encontrado.
   (c) Se o nome não for encontrado em algum dicionário da pilha, reporte um erro.

Para o programa em C da Figura 4.1, a pilha de dicionários na linha 7 teria o seguinte conteúdo (de cima para baixo):

<t, 6>

<j, 4>, <i, 3>, <size, 1>, <a, 1>

<sort, 1>

em que o nome do par nome–associação é mostrado como um par ordenado, e a associação é mostrada apenas como o número da linha onde foi declarada, por motivo de simplicidade. Observe que o nome `sort` está no escopo externo em relação aos parâmetros e ao corpo da função. Dessa perspectiva, a referência a `t` na linha 7 é uma referência local, enquanto as referências a `a` e `i` são não-locais.

Em contraste, a tabela de símbolos na linha 4 e novamente no início da linha 11 apareceriam da seguinte forma (de cima para baixo):

<j, 4>, <i, 3>, <size, 1>, <a, 1>

<sort, 1>

O escopo contendo `t` não foi encontrado ainda na linha 4 e foi abandonado no final da linha 10.

1. O leitor observador terá notado aqui a utilização da regra de referência *forward*.

Para linguagens que usam escopo estático, a tabela de símbolos é construída durante a fase de análise semântica da compilação (veja a Seção 2.4) e dura todo o restante do processo de compilação. A Seção 4.5 analisa o caso em que possam existir múltiplas instâncias de definições do mesmo nome.

4.5 RESOLVENDO REFERÊNCIAS

O ambiente de referências é importante porque define a coleção de comandos nos quais um nome pode ser validamente referenciado.

> **Definição:** Para escopo estático, o *ambiente de referência* para um nome é o seu escopo definidor e todos os subescopos aninhados.

Usamos essa definição e o conhecimento de como uma tabela de símbolos funciona para analisar escopos disjuntos e escopos aninhados que redefinam um nome dentro do seu ambiente de referenciação.

A Figura 4.2 representa um exemplo tanto de escopos aninhados quanto de disjuntos nos quais existem três declarações do nome `i`, duas de `j` e uma de todos os outros. As tabelas de nomes individuais para cada escopo são as seguintes:

1. O escopo externo consiste de duas variáveis declaradas na linha 1 mais as três funções: <h, 1>, <i, 1>, <B, 2>, <A, 8>, <main, 14>.
2. O escopo da função `B` contém o parâmetro mais as duas variáveis da linha 3: <w, 2>, <j, 3>, <k, 3>.
3. O escopo da função `A` consiste dos dois parâmetros da linha 8 mais a declaração da linha 9: <x, 8>, <y, 8>, <i, 9>, <j, 9>.

```
1  int h, i;
2  void B(int w) {
3     int j, k;
4     i = 2*w;
5     w = w+1;
6     ...
7  }
8  void A (int x, int y) (
9     float i, j;
10    B(h);
11    i = 3;
12    ...
13 }
14 void main() {
15    int a, b;
16    h = 5; a = 3; b = 2;
17    A(a, b);
18    B(h);
19    ...
20 }
```

| Figura 4.2 Referências em Escopos Disjuntos e Aninhados

4 O escopo da função `main` consiste das duas variáveis declaradas na linha. 15: <a, 15> , <b, 15>.

Quando essas funções forem compiladas, a pilha da tabela de símbolos para cada função terá o seguinte conteúdo:

Função	Pilha de Tabelas de Símbolos
B	<w, 2>, <j, 3>, <k,3> <h, 1>, <i, 1>, <B, 2>, <A, 8>, <main, 14>
A	<x, 8>, <y, 8>, <i, 9>, <j, 9> <h, 1>, <i, 1>, <B, 2>, <A, 8>, <main, 14>
main	<a, 15>, <b, 15> <h, 1>, <i, 1>, <B, 2>, <A, 8>, <main, 14>

Com essas informações, podemos facilmente resolver as referências não-locais ou definidas multiplamente:

Linha	Referência	Declaração
4	i	1
10	h	1
11	i	9
16, 18	h	1

4.6 ESCOPO DINÂMICO

Em termos de tabelas de símbolos, o dicionário para cada escopo pode ser construído em tempo de compilação, porém o gerenciamento da pilha de escopos é feito em tempo de execução. Quando se entra em um escopo em tempo de execução, seu dicionário é empilhado no topo da tabela de símbolos e desempilhado quando se sai do escopo. A resolução de nomes é executada em tempo de execução em vez de em tempo de compilação. Fora isso, tudo ocorre da mesma forma.

> **Definição:** No *escopo dinâmico*, um nome é associado à sua declaração mais recente com base no histórico de execução do programa.

Usando novamente a Figura 4.2 como nosso exemplo e supondo escopo dinâmico, examinaremos a resolução da referência a `i` na linha 4. Suponha um histórico de chamadas: main (linha 17) → A (linha 10) → B. A tabela de símbolos em tempo de execução tem o seguinte formato:

Função	Dicionário
B	<w, 2>, <j, 3>, <k, 3>
A	<x, 8>, <y, 8>, <i, 9>, <j, 9>
main	<a, 15>, <b, 15> <h, 1>, <i, 1>, <B, 2>, <A, 8>, <main, 14>

A referência a `i` na linha 4 é resolvida com aquela declarada em `A` na linha 9, já que este é o primeiro `i` encontrado.

Se, em vez disso, analisarmos o histórico de chamadas: main (linha 18) → B, a tabela de símbolos em tempo de execução teria o seguinte formato:

Função	Dicionário
B	<w, 2>, <j, 3>, <k, 3>
main	<a, 15>, <b, 15> <h, 1>, <i, 1>, <B, 2>, <A, 8>, <main, 14>

A referência a `i` agora é resolvida com aquela declarada globalmente na linha 1.

O propósito dos exemplos é deixar claro que a resolução de nomes de referências não-locais usando escopo dinâmico requer conhecimento do histórico em tempo de execução do programa. Em especial, históricos diferentes de tempo de execução podem resultar em diferentes resoluções de nomes para a mesma referência.

Embora o escopo dinâmico seja usado em versões iniciais de Lisp (antes de Common Lisp), APL, Snobol e Perl, ele provou ser especialmente propenso a erros e tem sido freqüentemente abandonado. O escopo dinâmico não é mais uma característica comum de linguagens modernas, exceto Common Lisp e Perl, que também suportam escopo estático.

Há diversos motivos para se preferir o escopo estático. Primeiro, o escopo dinâmico compromete a capacidade de verificar-se estaticamente referências a tipos de variáveis que não sejam locais, já que nem todas as variáveis não-locais de um comando dentro de uma função chamada podem ser identificadas estaticamente. Em segundo lugar, todas as variáveis em uma função que chama outras funções são visíveis àquelas funções chamadas (e, portanto, podem ser alteradas), o que tende a reduzir a confiabilidade de um programa, de modo geral. Finalmente, o acesso a uma variável não-local, seguindo-se uma cadeia de ligações dinâmicas, tende a consumir mais tempo do que o acesso a uma não-local em um ambiente de escopo estático. Por esses motivos, o escopo dinâmico tende a não ser uma característica de linguagens modernas de programação.

A próxima seção retorna ao escopo estático para analisar a questão da visibilidade.

4.7 VISIBILIDADE

Conforme mostrado na Figura 4.2 da página 93, um nome redeclarado em um escopo aninhado efetivamente *esconde* a declaração mais externa. Isso acontece porque uma referência usa como padrão a declaração mais interna.

> **Definição:** Um nome é *visível* a uma referência se a sua referenciação incluir essa referência e o nome não tiver sido redeclarado em um escopo mais interno.

Um nome é, portanto, visível por todo o seu ambiente de referenciação desde que não seja escondido por uma redeclaração em algum subescopo aninhado. Algumas linguagens fornecem um mecanismo para se referenciar um nome escondido. Em Java, por exemplo, uma variável de instância de classe escondida, digamos `x`, pode ser referenciada como `this.x`, em que `this` se refere ao objeto corrente. Analise o seguinte fragmento de programa:

```
1  public class Student {
2     private String name;
3     public Student (String name, ...) {
4        this.name = name;
5        ...
6     }
7 }
```

```
procedure Main is
   x : Integer;
   procedure p1 is
      x : Float;
      procedure p2 is
      begin
         ... x ...
      end p2;
   begin
      ... x ...
   end p1;
   procedure p3 is
   begin
      ... x ...
   end p3;
begin
   ... x ...
end main;
```

| Figura 4.3 Programa em Ada com Múltiplos Declarantes de um Nome

O parâmetro `name` na linha 3 esconde a variável declarada na linha 2; assim, a linha 4 usa a notação `this.name` para se referir à variável escondida.

Ada permite que um nome seja qualificado explicitamente, possibilitando assim que tanto uma variável local quanto uma global com o mesmo nome seja referenciada dentro do mesmo bloco. Analise o esqueleto do programa Ada mostrado na Figura 4.3. Aqui, vemos duas variáveis declaradas com o nome `x`, uma `Integer` e outra `Float`. As referências a `x` em `p3` e em `main` acessam a variável `Integer`, enquanto as referências a `x` em `p1` e `p2` acessam a variável `Float`. Entretanto, qualquer uma das referências em `p1` ou `p2` pode ser mudada para `main.x` para acessar a variável `Integer`. Esse recurso de Ada se estende a blocos também, já que qualquer bloco Ada pode ser nomeado.

4.8 SOBRECARGA

Até agora, supomos que, em cada escopo de definição, não exista mais do que uma declaração de um determinado nome. A capacidade de usar o mesmo nome para entidades relacionadas pode melhorar tanto a legibilidade quando a facilidade de escrita de uma linguagem, reduzindo a confusão de nomes. Por exemplo, ser capaz de `print` ("imprimir") uma variável de qualquer tipo, como em Java, é muito mais simples do que se lembrar de usar `WriteInt( )` para inteiros, `WriteReal( )` para valores de ponto flutuante e assim por diante, como em Modula.

> **Definição:** A *Sobrecarga* usa o número ou tipo de argumento para distinguir entre funções ou operadores com nomes idênticos.

Um exemplo é a prática antiga de se sobrecarregar as funções e os operadores aritméticos internos de uma linguagem. A maioria das linguagens sobrecarrega os operadores +, -, * e / para denotar aritmética de ponto flutuante ou inteira, dependendo dos tipos

de operandos. Em linguagens mais modernas, esta sobrecarga foi estendida para cobrir todas as variantes de valores inteiros e de ponto flutuante (por exemplo, precisão simples e dupla). Java também sobrecarrega o operador + para denotar a concatenação de *strings* sempre que pelo menos um dos operandos for do tipo `String`. A sobrecarga também se estende a funções internas. Em Fortran, os comandos de entrada e saída `read` e `write` aceitam qualquer um dos tipos definidos pela linguagem.

Historicamente, nenhuma das principais linguagens iniciais estendia a sobrecarga para operadores e funções definidas pelo usuário. Isso levou à anomalia em Modula de que diferentes nomes para funções de entrada e saída eram usados para tipos distintos. Por exemplo, o nome `Read( )` é usado apenas para caracteres, `ReadInt( )` para inteiros, `ReadString( )` para *strings* e `ReadReal( )` para números de ponto flutuante. Isso não ocorreu em Pascal porque suas funções de entrada/saída eram internas, enquanto eram rotinas comuns de bibliotecas externas em Modula.

Ada foi a primeira linguagem amplamente usada a estender a sobrecarga de operadores a tipos e funções definidas pelo programador.[2] Parecia natural para os projetistas de Ada que, se alguém fosse definir um novo tipo para implementar vetores matemáticos, o operador + deveria ser extensível para denotar adição de vetores. Ada também permite a sobrecarga com base no tipo de retorno, diferentemente de outras linguagens que suportam sobrecarga.

C++ também permite que operadores aritméticos e métodos sejam sobrecarregados, enquanto Java restringe a sobrecarga a métodos. Java permite que nomes de métodos dentro da mesma classe sejam sobrecarregados, desde que o método realmente chamado possa ser determinado pelo número ou por tipos de parâmetros. Um exemplo ocorre na classe `PrintStream`, em que os métodos `print` e `println` são intensamente sobrecarregados:

```
public class PrintStream extends FilterOutputStream {
   ...
   public  void  print(boolean b);
   public  void  print(char c);
   public  void  print(int i);
   public  void  print(long l);
   public  void  print(float f);
   public  void  print(double d);
   public  void  print(char[ ] s);
   public  void  print(String s);
   public  void  print(Object obj);
}
```

Java permite que tanto uma variável de instância quanto um método tenham o mesmo nome, já que todas as referências ao método têm parênteses como sufixo:

```
public class Student {
   private String name;
   ...
   public String name ( ) { return name; }
```

2. Naquele tempo, isso foi um grande salto de fé. O único algoritmo conhecido para efetuar a resolução de nomes em funções e operadores sobrecarregados definidos pelo programador tinha complexidade de tempo exponencial!

Observe que o método `name( )` contém uma referência à variável de instância `name` e não há ambigüidade nele. Na verdade, uma prática comum na programação em Java é definir um método de acesso em uma classe que retorne o valor da variável de instância do mesmo nome, como este exemplo ilustra.

4.9 TEMPO DE VIDA

As primeiras linguagens imperativas, a saber, Fortran e Cobol, usavam alocação estática tanto para variáveis quanto para funções, ou seja, a memória a ser usada era atribuída em tempo de compilação. No caso de funções, havia apenas uma única área de memória tanto para argumentos quanto para o endereço de retorno; por causa disso, funções recursivas não eram suportadas. Veja nos Capítulos 9 e 10 uma discussão mais completa.

No caso de variáveis, o trabalho de gerenciar memória era totalmente de responsabilidade do programador, caso a memória de programa requerida excedesse a quantidade disponível. Já que esta era uma ocorrência comum, por conta da memória física limitada da época, muito do esforço de escrita de um programa era gasto no gerenciamento de memória.

Com o desenvolvimento de Algol veio a noção de que a memória para variáveis, incluindo parâmetros de funções e endereços de retorno, deveria ser alocada e desalocada em tempo de execução, quando se entrasse ou saísse de um escopo, respectivamente. Naquela época, esta era uma idéia radical, já que ninguém sabia como implementar tal esquema de gerenciamento de memória.

O gerenciamento básico de memória usado por todas as linguagens modernas de hoje fundamenta-se nesse conceito. O restante desta seção explora a implicação dessa noção fundamental e mostra como diversas linguagens permitem que o tempo de vida de uma variável ou um objeto seja estendido. O Capítulo 10 apresenta uma visão dessa questão mais orientada a implementação. Na discussão aqui ignoramos a questão de alocação dinâmica de memória por meio de um comando `new` ou `malloc` (e sua desalocação), um assunto explorado no Capítulo 11.

> **Definição:** O *tempo de vida* de uma variável é o intervalo de tempo durante o qual a variável tem um bloco de memória alocado para si.

A maior diferença na semântica de programas entre a alocação estática de memória de Fortran e a alocação com base em escopo de Algol é que, no primeiro caso, uma variável nunca esquece seu valor. Analise a variável `t` da Figura 4.1 na página 91. Em cada iteração do laço para `j`, qualquer memória que possa ser alocada para `t` também é desalocada. Para este exemplo em especial, a desalocação não é um problema, já que `t` não precisa lembrar seu valor.

Entretanto, analise o problema de escrever uma função que conte o número de vezes que foi chamada.

```
double Func( ) {
   int count = 0;
   ...
}
```

A partir do ponto de vista do uso, apenas a função precisa acessar o contador. Todavia, se a variável contadora for declarada dentro da função, o valor do contador é perdido cada vez que se sai da função, sob a política de alocação de memória de Algol. A alternativa é incrementar o escopo da variável contadora, com o único objetivo de aumentar seu tempo de vida.

O restante desta seção analisa mecanismos de linguagens que, de alguma forma, quebram a regra de "escopo igual a tempo de vida", ao mesmo tempo em que a mantém como caso-padrão.

Por exemplo, um programa em Pascal consiste de uma única unidade de compilação. As variáveis declaradas no nível de unidade de compilação são efetivamente alocadas estaticamente, não importa como sejam implementadas. Assim, tais variáveis mantêm seus valores pela vida do programa.

C permite múltiplas unidades de compilação. As variáveis declaradas no nível de unidade de compilação (por exemplo, `h` na Figura 4.2 na página 93) são todas efetivamente alocadas estaticamente. Tais variáveis retêm sua memória e, assim, seus valores, mesmo quando a execução de um programa entra e sai da sua unidade de compilação.

A outra característica adicionada por C é a capacidade de declarar variáveis como sendo `static`. Quando usadas com uma variável declarada dentro de uma função, a variável é alocada estaticamente. Se usada com uma variável declarada em nível de unidade de compilação, não tem efeito sobre a alocação de memória da variável, já que tais variáveis já usam alocação estática.

Java também permite que variáveis declaradas no nível de classe usem o modificador `static`. Tais variáveis ou objetos têm sua memória alocada estaticamente.

4.10 RESUMO

Este capítulo apresenta os conceitos-chave relacionados a nomes e suas ligações em linguagens de programação contemporâneas. A ligação de um nome com uma variável e seu tipo desempenha um papel-chave no desenvolvimento de um sistema de tipos para uma linguagem.

EXERCÍCIOS

4.1 Escolha uma das seguintes linguagens: Python, Modula, Ada, C# ou Perl. Após consultar uma referência autorizada, discuta cada um dos seguintes requisitos para essa linguagem:
(a) Declaração antes do uso.
(b) Sobrecarga de operadores para tipos definidos pelo programador.
(c) Tempo de associação do tamanho de uma matriz.
(d) O que constitui um escopo.
(e) A localização de uma declaração dentro de um escopo.

4.2 Após consultar uma referência autorizada sobre requisitos para uso de variáveis globais em C, responda às questões a seguir:
(a) Como elas podem ser acessadas em outras unidades de compilação?
(b) Como variáveis globais podem ser escondidas de outras unidades de compilação?
(c) Por que você esconderia variáveis globais?

4.3 C e C++ distinguem declarações e definições. Qual é a distinção? Dê um exemplo de cada uma.

4.4 Explique o uso de arquivos de cabeçalho em C e C++. Por que Java não usa arquivos de cabeçalho?

4.5 A maioria das linguagens de programação proíbe a redeclaração do mesmo nome de variável dentro do mesmo escopo.
(a) Explique por que o uso apenas de tipos não é suficiente para distinguir tal duplicação.
(b) Qual a diferença entre esta restrição e a ausência da mesma para funções sobrecarregadas?

4.6 Para a linguagem C, dê três exemplos de valores-r que não possam ser valores-l. Dê mais três exemplos de valores-l. Há valores-l que não possam ser valores-r? Explique.

5 Tipos

"Tipos são o fermento da programação computacional; eles a tornam digerível."

Robin Milner

VISÃO GERAL DO CAPÍTULO

Os tipos disponíveis para um programador são muitos e variados. Há um núcleo comum de tipos básicos que todas as linguagens suportam (inteiros, reais, boleanos e caracteres) assim como diversos tipos compostos (ponteiros, matrizes, estruturas e funções). Este capítulo resume essas idéias e seus usos em diversas linguagens contemporâneas.

> **Definição:** Um *tipo* é um conjunto de valores e um conjunto de operações sobre esses valores.

Por exemplo, o tipo inteiro familiar possui valores $\ldots, -2, -1, 0, 1, 2, \ldots$, e as operações $+, -, *, /, <, \ldots$ sobre esses valores. O tipo `boleano` possui valores `verdadeiro` e `falso` e operações $\wedge$, $\vee$ e $\neg$ (*e, ou* e *não*, respectivamente) sobre esses valores.

Geralmente, um tipo computacional possui um número finito de valores, devido ao tamanho fixo de memória alocada para armazená-los. Isso é especialmente problemático para tipos numéricos. Há exceções, entretanto. Por exemplo, Smalltalk usa números fracionários sem limites como padrão e o tipo Haskell `Integer` representa inteiros sem limites, o que permite que a aritmética simule com mais precisão a faixa completa de operações matemáticas. Java possui um recurso semelhante, que é embutido na sua classe `BigInteger`. Porém, esses tipos representam as exceções.

De modo geral, contudo, o uso de um tamanho fixo de memória para armazenar inteiros pode resultar em uma operação aritmética que gere um valor fora da faixa válida. Tente calcular 20! (fatorial) usando um inteiro com tamanho de memória fixo; o valor excede os limites de um inteiro de 64 bits.

Ainda mais problemático é o uso de números de pontos flutuantes, com tamanho fixo de memória, para aproximar números racionais matemáticos. Por exemplo, se usarmos ponto flutuante binário, o número 0,2 não é exato em binário; assim, calcular $5 \times 0{,}2$ não resulta exatamente 1. Pontos flutuantes computacionais são inconsistentes com seus correspondentes matemáticos.

Nas primeiras linguagens de alto nível, Fortran, Algol e Cobol, todos os tipos de linguagem eram internos. Com o decorrer do tempo, a inapropriação dessa situação tornou-se óbvia. Suponha, por exemplo, que o programador precisasse de um tipo `color` para representar cores que você pudesse exibir em um monitor. Geralmente, um inteiro de determinado tamanho seria usado com alguns bits que representassem as cores vermelha, verde e azul (RGB, sigla em inglês para *red*, *green* e *blue*). Com certeza, multiplicar duas cores não faz sentido, mas usar uma representação inteira permite efetuar tal operação sem significado. O propósito de tipos em linguagens de programação é fornecer aos programadores formas de definir convenientemente e efetivamente novos tipos para construir programas que modelem melhor uma solução.

5.1 ERROS DE TIPOS

Pelo padrão, a representação de máquina de dados não carrega informações explícitas relacionadas a tipos, ou seja, dados de máquina são apenas uma seqüência de bits, tornando-os essencialmente *sem tipo*. Esses bits estão sujeitos a diferentes interpretações.

Por exemplo, analise a seguinte *string* de 32 bits:

0100 0000 0101 1000 0000 0000 0000 0000

na qual adicionamos espaços por motivo de legibilidade. Esses 32 bits poderiam representar qualquer um dos seguintes:

1. O número de ponto flutuante 3,375.
2. O inteiro de 32 bits 1.079.508.992.

3 Dois inteiros de 16 bits, 16472 e 0.

4 Quatro caracteres ASCII `@ X NUL NUL`.

Sem mais informações, não há como saber qual interpretação está correta.

Uma das muitas dificuldades de programar linguagem no nível de máquina (assembly) ocorre pelo fato de a carga de trabalho para se registrar os tipos de dados das variáveis recair integralmente sobre o programador. Por exemplo, buscar incorretamente uma quantidade de 32 bits que represente um inteiro e executar uma adição de ponto flutuante no nível de uma máquina sobre ele, basicamente cria lixo como resultado.

> **Definição:** Um *erro de tipo* é qualquer erro que surja porque uma operação é tentada sobre um tipo de dado para o qual ela não está definida.

Erros de tipos são problemas comuns em programação de linguagem assembly. Programar em uma linguagem de nível mais alto geralmente reduz erros de tipos, já que compiladores e sistemas de tempo de execução são projetados para identificá-los.

Assim, a definição e a implementação apropriadas de tipos de dados no projeto de linguagens de programação é imensamente importante. O assim chamado *sistema de tipo* pode fornecer uma base para a detecção precoce (em tempo de compilação) do uso incorreto de dados por um programa.

> **Definição:** Um *sistema de tipo* é uma definição precisa das associações entre o tipo de uma variável, seus valores e as operações possíveis sobre esses valores.

A idéia de um sistema de tipo mudou dramaticamente desde a emergência da programação no nível de máquina e durante sua evolução nas últimas três décadas. Por exemplo, analise a história da linguagem de programação C, que foi desenvolvida no final da década de 1960, para permitir portar o sistema operacional Unix entre diferentes arquiteturas de máquina. C foi originalmente baseada na linguagem sem tipos BCPL. Contudo, enquanto C se desenvolvia pelas décadas de 1970 e 1980 do C de Kernighan e Ritchie para ANSI C e C++, adquiriu características de tipagem mais fortes e melhores. Entretanto, adicionar novas regras de tipagem a uma linguagem não é tão simples quanto parece (Stroustrup, 1994, p. 20):

> Durante a escrita e a depuração, adquiri um grande respeito pela expressividade do sistema de tipo de Simula e pela capacidade do seu compilador de capturar erros de tipos. Observei que erros de tipos quase invariavelmente refletiam um erro tolo de programação ou uma falha conceitual no projeto. ... Em contraste, tinha descoberto que o sistema

Observação

O custo de um erro

Um dos axiomas da prática de engenharia de software moderna é que um erro não detectado em uma fase de desenvolvimento custa, em média, uma ordem de mais magnitude para ser corrigido na próxima fase. Por exemplo, suponha que um erro de tipo capturado durante a fase de implementação custe $ 1 para ser corrigido. Se tal erro não fosse detectado até a fase de teste, custaria em média $ 10 para ser corrigido. Além disso, capturar o mesmo erro após a fase de teste custaria em torno de $ 100 para corrigi-lo. Por exemplo, um erro no algoritmo de multiplicação de ponto flutuante em modelos iniciais do chip Intel 386 custou $ 472 milhões para corrigi-lo, já que esse erro não foi detectado antes que o chip chegasse ao mercado.

de tipos de Pascal era pior do que inútil – uma camisa-de-força que causava mais problemas do que soluções, forçando-me a entortar meus projetos para adaptá-los a um artefato orientado a implementação.

5.2 TIPAGEM ESTÁTICA E DINÂMICA

Um sistema de tipo impõe restrições específicas, como o requisito de que os valores usados em uma operação aritmética devam ser numéricos. Tais restrições não podem ser expressas sintaticamente em EBNF. Para algumas linguagens (por exemplo, C), a verificação de tipos é executada em tempo de compilação, enquanto para outras (por exemplo, Perl) ela é realizada em tempo de execução. Em qualquer caso, o sistema de tipo é parte da semântica da linguagem; semântica de tempo de compilação, no primeiro caso, e semântica de tempo de execução, no último.[1] Em Java, por exemplo, a maior parte da verificação de erros ocorre em tempo de compilação, porém a conversão (*casting*) é verificada em tempo de execução.

Algumas linguagens, como C e Ada, requerem que um único tipo seja associado a uma variável quando essa for declarada, permanecendo associada por toda sua vida em tempo de execução. Isso permite que o tipo de valor de uma expressão seja determinado em tempo de compilação.

> **Definição:** Uma linguagem é *tipada estaticamente* se os tipos de todas as variáveis são fixados quando são declaradas em tempo de compilação.

Outras linguagens, como Perl, Python e Scheme, permitem que o tipo de uma variável seja redefinido cada vez que um novo valor for atribuído a ela em tempo de execução.[2] Para implementar essa característica, um indicador de tipo é armazenado em tempo de execução com cada valor.

> **Definição:** Uma linguagem é *tipada dinamicamente* se o tipo de uma variável puder variar em tempo de execução de acordo com o valor atribuído.

Lembre-se de que um *erro de tipo* é qualquer erro que ocorra quando uma operação é tentada sobre um tipo de valor para o qual não está bem definida. Por exemplo, considere a expressão em linguagem C, `x+u.p`, na qual `u` é definido como a união `{int a; float p;}`:

```
u.a = 1;
...
x = x + u.p;
```

Esse trecho de código é uma instância de um erro de tipo permitido pelo fato de que tanto valores `int` quanto `float` de `u` compartilham a mesma palavra de memória, que é inicializada com um valor `int`. Se `x` for um `float`, a expressão `x+u.p` faz com que um valor `int` seja envolvido em uma adição de ponto flutuante sem ser convertido para uma representação de ponto flutuante, gerando lixo como resultado.[3]

> **Definição:** Uma linguagem de programação é *fortemente tipada* se o seu sistema de tipo permitir que todos os erros de tipos em um programa sejam detectados em tempo de compilação ou em tempo de execução.

---

1. Alguns textos se referem a qualquer erro detectado em tempo de compilação como sendo um erro de sintaxe. Entretanto, o uso de tipos é parte do significado (ou da semântica) de um programa.

2. Alguns autores se referem a tais linguagens como *não tipadas* ou *sem tipo* no sentido de que o tipo da variável não é especificado no programa-fonte. Acreditamos que esse uso é enganoso, preferindo reservar esses termos para descrever linguagens, como linguagens assembly/de máquina, nas quais nem variáveis nem valores são tipados.

3. Em alguns compiladores C, a situação é piorada pelo fato de que um `int` ocupa 16 bits, enquanto um `float` tem 32 bits. Esse erro não pode, de modo geral, ser detectado em tempo de compilação nem em tempo de execução.

O fato de uma linguagem ser estática ou dinamicamente tipada não impede que ela seja fortemente tipada. Por exemplo, Ada e Java são linguagens fortemente tipadas, enquanto C e C++ não são. Linguagens dinamicamente tipadas como Scheme e Perl são fortemente tipadas, já que a tipagem e a verificação dos valores que é feita pelo interpretador em tempo de execução evitam erros de tipos não detectados. O exemplo dado anteriormente expõe um entre diversos problemas no sistema de tipo de C. A tipagem forte geralmente promove programas mais confiáveis e é vista como uma virtude no projeto de linguagens de programação.

As próximas duas seções examinam tipos internos. *Tipos básicos* denotam valores de dados que são atômicos ou indivisíveis do ponto de vista do programador. Em contraste, *tipos compostos* ou *não básicos* correspondem a dados que são compostos de outros dados. Por exemplo, diversos tipos de listas, matrizes e conjuntos são tipos compostos.

5.3 TIPOS BÁSICOS

Os tipos básicos de uma linguagem de programação geralmente correspondem aos tipos de dados que estão disponíveis em máquinas contemporâneas. A Tabela 5.1 resume os tipos básicos que estão disponíveis em C,[4] Ada e Java.

Em um computador moderno de 32 bits, os requisitos de memória para esses tipos são geralmente medidos com o uso das seguintes delineações de unidades de memória:

- *Nibble:* 4 bits contíguos de memória; 1/2 byte
- *Byte:* 8 bits contíguos de memória
- *Half-Word:* 16 bits contíguos de memória = 2 bytes
- *Word:* 32 bits contíguos de memória = 4 bytes
- *Double word:* 64 bits contíguos de memória = 8 bytes
- *Quad word:* 128 bits contíguos de memória = 16 bytes

Historicamente, o tamanho de uma palavra (*word*) tem variado; alguns tamanhos de palavras incluíam 8, 12, 16, 18, 24, 32, 36, 48 e 60 bits. Quando as máquinas de 64 bits substituírem as de 32 bits, uma palavra deve se tornar associada a 64 bits.

Como exemplo, em Java a noção de um byte é definida como um tipo explícito e um `byte` Java pode representar qualquer um de $2^8 = 256$ valores binários distintos. Um `short` Java ocupa meia palavra, um `int` ocupa uma palavra, e um valor `long` ocupa uma palavra dupla. Java é incomum pelo fato de a linguagem prescrever os tamanhos anteriores dos tipos

| Tabela 5.1 Tipos Básicos em C, Ada e Java

Tipo	C	Ada	Java
Byte			`byte`
Inteiro	`short, int, long`	`integer`	`short, int, long`
Número real	`float, double`	`float, decimal`	`float, double`
Caractere	`char`	`character`	`char`
Boleano		`boolean`	`boolean`

4. C++ suporta um tipo boleano, mas, nesse momento, poucos compiladores parecem suportá-lo.

inteiros. A maioria das linguagens, incluindo C, C++ e Pascal, deixam para o implementador do compilador a tarefa de associar tipos a tamanhos de memória; em muitos casos, os padrões especificam tamanhos mínimos para cada tipo. Ada permite que o programador especifique a faixa necessária para cada variável. Mas a faixa pode disparar um erro de compilação se exceder o que o compilador suporta.

Devido ao fato de cada um desses tipos numéricos ter tamanho finito, uma operação aritmética pode produzir um valor que esteja fora da sua faixa definida. Por exemplo, analise a adição 2.147.483.647 + 1 usando aritmética de complemento de dois de 32 bits; a adição gera um valor fora da faixa permitida. Algumas máquinas (por exemplo, MIPS) geram uma interrupção quando tal erro ocorre; entretanto, muitas, incluindo a Java Virtual Machine, não (Arnold e Gosling, 1998, p. 358):

> Se ocorrer *overflow*, então o sinal do resultado não é o mesmo da soma matemática dos dois valores do operando.

A falha em gerar uma interrupção ou uma exceção quando ocorre *overflow* é um erro de projeto que complica desnecessariamente a vida do programador.[5]

Em linguagens como C (usando sintaxe de estilo C), as seguintes operações `int` são funções que recebem dois argumentos `int` e retornam um valor `int` como resultado:

```
+, -, *, /, %
==, !=, <, <=, >, >=
```

O símbolo `%` denota a função de *módulo* que retorna o resto da divisão inteira. As demais funções possuem comportamentos convencionais, exceto pela divisão, que em matemática normalmente produziria um número real como resultado.

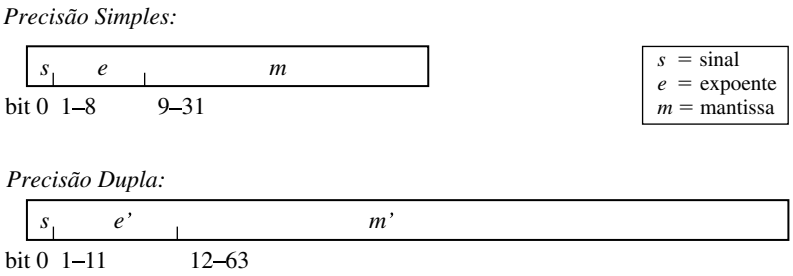
Observe que linguagens como C, exceto Java, são incomuns para que os operadores de igualdade e relacionais produzam um resultado `int`. Para permitir uma interpretação consistente, o valor zero é interpretado como `falso` e todos os outros são interpretados como `verdadeiro`. Perl também usa essa convenção.

Na maioria das linguagens de programação, um número real é um valor não integral armazenado em formato de ponto flutuante em precisão simples (por exemplo, `float` de C) ou dupla (por exemplo, `double` de C), dependendo da precisão e da faixa de valores necessárias. Para máquinas atuais de 32 bits, um valor `float` requer uma palavra de memória de 32 bits, enquanto um valor `double` requer uma palavra de 64 bits.

Valores de ponto flutuante em formato IEEE são representados pelo uso da *notação científica*, na qual um valor é representado como o produto de um número binário que contenha um único dígito diferente de zero à esquerda do ponto decimal por uma potência apropriada de 2. Por exemplo, o número decimal 123,45 é representado em notação científica decimal como $1{,}2345 \times 10^2$. Computadores usam representação binária em vez de decimal para armazenar números, mas, além disso, seguem os mesmos princípios. Computacionalmente, um valor real é armazenado como o produto de um valor real binário contendo um único 1 à esquerda do ponto binário por uma potência apropriada de 2. Máquinas mais antigas não seguem o formato IEEE, mas usam alguma forma de notação científica.

Considere representar o número real decimal 3,375 em notação científica binária. A conversão para binário é feita separando-se o número nas suas partes inteira e decimal

5. O padrão de bits 100...0 em aritmética de complemento de dois poderia ser interpretado como valor *indefinido*, em vez de um número negativo sem contraparte positiva. Tal projeto simplificaria a semântica de muitos problemas de depuração (por exemplo, a identificação de variáveis não inicializadas).



| Figura 5.1 Representação de Números de Ponto Flutuante no padrão IEEE 754

e convertendo cada parte separadamente para binário. A parte inteira de 3,375 é 3, que em binário é 11_2. A parte fracionária 0,375 é $0{,}011_2$, já que $0{,}375 = 0{,}25 + 0{,}125 = 2^{-2} + 2^{-3} = 0{,}011_2$. Assim, a notação científica para 3,375 em binário é:

$$3{,}375_{10} = 11{,}011_2 = 1{,}1011_2 \times 2^1$$

Computadores modernos inventados após 1980 usam o padrão IEEE 754 para representar valores em ponto flutuante (Goldberg, 1991). Esse padrão está resumido na Figura 5.1. Um número de ponto flutuante de precisão simples normalizado possui:

- Um bit de sinal com 0 denotando + e 1 denotando −.
- Um expoente *e* de 8 bits denotando uma potência de 2 e codificado como o expoente verdadeiro +127 (todos bits um). O número 127 é chamado de *bias*, e a codificação é chamada de *notação excedente em 127*.
- Uma mantissa *m* de 32 bits denotando uma fração binária na faixa de 0 a 1 e interpretada como o número 1.*m*. O "1" na fração é comumente chamado de *bit escondido*.

No conjunto, isso representa o número:

$$(-1)^s \times 1.m \times 2^{e-127}$$

Observe que o número 0,0 é representado pelo uso do padrão de zero bits. A versão de precisão dupla permite mais bits para a fração binária e de expoente, aumentando, assim, a faixa que pode ser representada bem como o número de dígitos significativos.

A precisão simples permite uma faixa de aproximadamente $\pm 10^{38}$ a 10^{-38}, enquanto a precisão dupla permite uma faixa de $\pm 10^{308}$ a 10^{-308}. Números de precisão simples têm em torno de sete dígitos decimais significativos na mantissa, enquanto a precisão do número de precisão dupla tem 16 dígitos decimais significativos.

O padrão IEEE 754 também fornece representações únicas para números incomuns, como *infinito* (só 1s no expoente e tudo 0s na mantissa) e *não um número* (NaN) (só 1s no expoente e não só 0s na mantissa). Números de ponto flutuante desnormalizados também são definidos para fornecer uma representação para números que sejam menores do que o menor valor normalizado (só 0s no expoente e qualquer mantissa diferente de zero).

Para continuar nosso exemplo, considere o número decimal 3,375. Ele possui um bit de sinal 0, já que o número é positivo. Uma vez que o expoente real é 1, o expoente da

máquina é 127 + 1 = 128 em decimais, ou 1000 0000 em binário, já que o expoente da máquina é armazenado em notação excedente em 127. Finalmente, a mantissa é o valor após o ponto binário: 101 1000 0000 0000 0000 0000 em binário de precisão simples. Juntando o sinal, o expoente e a mantissa, obtemos:

0 1000 0000 101 1000 0000 0000 0000 0000

A maioria dos números decimais exatos como 0,2 não tem representações exatas em ponto flutuante binário. Assim, quando executamos determinadas operações aritméticas sobre eles, como $5 \times 0{,}2$, obtemos um resultado que não é exatamente igual ao matemático ($5 \times 0{,}2 \neq 1{,}0$ nesse caso). Esse e outros problemas relacionados à aritmética de ponto flutuante são discutidos com algum detalhe em (Hennessy e Patterson, 1998, Capítulo 4) e em (Goldberg, 1991).

Em C/C++, as seguintes operações recebem dois operandos do tipo `float`:

```
+, -, *, /
==, !=, <, <=, >, >=
```

Cada operação no primeiro grupo retorna um resultado `float`, e cada operação do segundo grupo retorna um `int`, que, conforme explicado anteriormente, geralmente é interpretado como um boleano.

Duas questões são levantadas por essa capacidade. Primeiro, a maioria dos símbolos listados aqui são idênticos àqueles usados por operações inteiras. Por exemplo, a adição de dois inteiros é denotada por + e calcula um resultado `int`, enquanto a adição de dois números de ponto flutuante também usa +, porém retorna um resultado `float`. Essa é uma instância da noção mais geral de sobrecarga de operadores.

> **Definição:** Um operador ou uma função é dito *sobrecarregado* quando seu significado varia conforme os tipos dos seus operandos ou argumentos ou resultado.

A sobrecarga de operadores e funções internas (por exemplo, funções E/S) tem sido usada desde os primeiros dias de Fortran. Exemplos mais interessantes de sobrecarga são discutidos no Capítulo 4.

A segunda questão levantada aqui é a de que as definições anteriores de operador de ponto flutuante estão incompletas, ou seja, elas não incluem os casos em que um ou outro operando é `int` em vez de `float`. Isso é chamado de aritmética de "modo misturado" em Fortran, e suas instâncias para as operações anteriores são resumidas da seguinte forma:

```
float (+, -, *, /) int
int (+, -, *, /)  float
float (==. !=, <, <=, >, >=) int
int (==. !=, <, <=, >, >=) float
```

As duas primeiras produzem um resultado `float`, enquanto as duas últimas produzem um resultado `int`.

Por exemplo, considere a soma 2,5 + 1 envolvendo um `float` e um `int`. O resultado mais apropriado para essa soma é 3,5, que não pode ser representado como `int`; ele deve ser `float` ou `double`. Para obter esse resultado, o valor `int` 1 deve ser convertido para seu equivalente `float` antes de a soma ser calculada, já que não há operações de modo misturado no nível de máquina.

Tabela 5.2

O Valor 2 em Diversas Representações de Dados

Tipo	Bits
`char`	0000 0000 0011 0010
`short`	0000 0000 0000 0010
`int`	0000 0000 0000 0000 0000 0000 0000 0010
`float`	0100 0000 0000 0000 0000 0000 0000 0000

Conversão de Tipos No nível de máquina, as operações aritméticas requerem que ambos os operandos sejam do mesmo tipo (`int`, `float` ou `double`). Para simular uma operação de modo misturado, um dos operandos deve ter seu valor convertido para o tipo do outro. Tal conversão é chamada de *conversão de tipo*, já que cria um padrão de bits diferente para representar o valor em um tipo diferente.

Como exemplo, considere as diversas representações do valor 2 em Java apresentadas na Tabela 5.2. Interpretar o caractere Unicode '2' como um inteiro de 16 bits daria o valor inteiro 50 (veja a Tabela 5.3). As representações inteiras de 16 e 32 bits são idênticas, exceto pelos 16 zeros iniciais a mais na versão de 32 bits. Interpretar o número de ponto flutuante 2,0 como um inteiro daria o valor inteiro 2^{30}, um número muito grande. Assim, uma conversão de tipo muitas vezes é necessária para garantir a interpretação apropriada de um valor em preparação para executar uma operação aritmética sobre esse valor.

Uma conversão de tipo é chamada de conversão *limitante*, se o valor resultante permitir menos bits do que o original (dessa forma, potencialmente perdendo informação). Por exemplo, na conversão de um `float` para um `int`, a parte fracionária após o ponto decimal será perdida; se convertermos 2,4 para um `int` resultará no valor 2, com a perda da fração 0,4.

Uma conversão de tipo é chamada de conversão *ampliadora* se o resultado do valor não requerer menos bits do que o original (geralmente sem perda de informação). Assim, converter 2 para `float` resulta no valor 2,0, que é matematicamente equivalente. Usamos o termo *geralmente* porque há exceções. Por exemplo, converter um inteiro de 32 bits suficientemente grande em um número de ponto flutuante de 32 bits resulta na perda dos 8 bits de ordem mais baixa do valor inteiro; assim, os dois valores só são aproximadamente iguais.

As linguagens devem suportar conversões de tipo *implícitas* apenas se forem do tipo de ampliação. Assim, por exemplo, a expressão 2,5 + 1 é permitida porque é interpretada como `2,5 + float(1)`, na qual a função de conversão `float()` é considerada uma conversão ampliadora.

Historicamente, tanto os tipos de dados numéricos quanto as *strings* têm causado problemas por causa das conversões de redução. Problemas com *strings* têm resultado principalmente do uso de *strings* de tamanho fixo.

De modo geral, as definições dos diversos tipos de conversões de tipo implícitas permitidas em uma linguagem de programação devem ser feitas com cuidado. Analise o algoritmo da Figura 5.2, que define conversões de tipos aritméticos em ANSI C (Kernighan e Ritchie, 1988, p. 198). Essas são regras confusas! Muitas vezes, elas não são bem compreendidas pelos programadores e, às vezes, apresentam resultados surpreendentes. Por exemplo, o efeito de algumas dessas regras é remover o sinal de um valor com sinal.

Conversões de redução devem requerer que o programa chame explicitamente uma função de conversão de tipo. Em qualquer conjunto de conversões implícitas existe o perigo de

Se...	então converta...
algum operando for `long double`	o outro para `long double`
algum operando for `double`	o outro para `double`
algum operando for `float`	o outro para `float`
algum operando for `unsigned long int`	o outro para `unsigned long int`
os operandos forem `long int` e `unsigned int`	
e `long int` puder representar `unsigned int`	o `unsigned int` para `long int`
e `long int` não puder representar `unsigned int`	ambos os operadores para `unsigned long int`
um operando for `long int`	o outro para `long int`
um operando for `unsigned int`	o outro para `unsigned int`

Figura 5.2 Regras de Conversão Implícita de Tipos ANSI C

que o resultado seja inesperado, às vezes até para o projetista da linguagem. Um exemplo clássico da linguagem PL/I é:

```
declare (a) char (3);
...
a = '123';
a = a + 1;
```

em que o valor final de `a` são três caracteres de espaço, e não `'124'`, como esperado. O perigo aqui é que, embora as regras de conversão de tipos possam ser razoáveis isoladamente, as formas pelas quais elas interagem nem sempre são bem compreendidas.

O termo *coerção de tipo* é usado para denotar uma alteração implícita de um valor de um tipo para outro, que possa ou não envolver uma conversão de tipo. No último caso, o valor permanece inalterado, apenas seu tipo mudou. Um exemplo de Java seria:

```
String s = "abc";
...
Object obj = s;
```

Nesse caso, a *string* `s` é convertida para um *Object* (objeto), mas o valor permanece o mesmo, a saber, um objeto *string*.

Conjuntos de Caracteres Linguagens mais antigas, como C, usam o conjunto de caracteres ASCII, que possui códigos de caracteres acima de 127 não especificados. A maioria dos sistemas operacionais modernos usa a chamada *codificação Latin-1*, que especifica os códigos acima de 127.

O conjunto de caracteres ASCII e seus códigos de caracteres em base 8 (octal)[6] são apresentados na Tabela 5.3. Os códigos de caracteres 000–037 e 177 octal (0–31, 127 decimal) não são imprimíveis. Todos os códigos 001–037 podem ser gerados por meio de combinações de teclas de controle; por exemplo, `n1` pode ser gerado por meio de Ctrl-J e o caractere

6. Mostrar os códigos de caracteres usando hexadecimais poderia ter sido melhor, mas não preencheria horizontalmente a página.

Tabela 5.3 Conjunto de Caracteres ASCII

Octal	0	1	2	3	4	5	6	7
00x	null	soh	stx	etx	eot	enq	ack	bell
01x	bs	ht	nl	vt	np	cr	so	si
02x	dle	dc1	dc2	dc3	dc4	nak	syn	etb
03x	can	em	sub	esc	fs	gs	rs	us
04x	␣	!	″	#	$	%	&	‘
05x	(	)	*	+	,	–	.	/
06x	0	1	2	3	4	5	6	7
07x	8	9	:	;	<	=	>	?
10x	@	A	B	C	D	E	F	G
11x	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W
13x	X	Y	Z	[	\	]	^	_
14x	´	a	b	c	d	e	f	g
15x	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w
17x	x	y	z	{	l	}	~	del

familiar de nova linha `\n`. Muitos dos nomes para os códigos 001–037 são abreviados; por exemplo, `bs` para backspace, `ht` para tabulação horizontal, `nl` para nova linha, `cr` para retorno de carro etc. A partir dessa tabela vemos que o caractere `2` possui código octal 062 (decimal 50).

Um valor Java do tipo `char` usa o conjunto Unicode (Unicode Consortium, 2000) UTF-16, que é um padrão de 16 bits para codificação de caracteres, em vez do padrão mais convencional de 7 bits ASCII.[7] O padrão Unicode define três conjuntos de caracteres, UTF-8, UTF-16 e UTF-32, de 8, 16 e 32 bits, respectivamente. O padrão UTF-8 inclui ASCII, sendo um subconjunto próprio de UTF-16, o qual, por sua vez, é um subconjunto próprio de UTF-32. O conjunto de caracteres Unicode UTF-16 fornece uma coleção rica de caracteres que permite a programas Java embutir um vocabulário mais internacional de que seus predecessores. Uma amostra da cobertura do conjunto de caracteres Unicode UTF-16 das linguagens do mundo e outros conjuntos de caracteres é apresentada na Tabela 5.4.

Como a tabela sugere, o padrão Unicode inclui códigos para conjuntos de caracteres que se espalham pelas principais línguas do mundo. Ele também inclui pontuação, sinais de acentuação, símbolos matemáticos e uma ampla gama de outros símbolos técnicos. Fornece códigos para caracteres modificadores, como o til (~) e códigos que são usados para codificar letras acentuadas (ñ). O padrão Unicode está constantemente sendo atualizado (veja informações mais detalhadas em **www.unicode.org**). Sua versão corrente define códigos para 49.194 caracteres dos alfabetos do mundo e outros conjuntos de símbolos.

7. ASCII é o *American Standard Code for Information Interchange*, que é um subconjunto do padrão ISO 8859-1 de código de 8 bits, conhecido como o conjunto de caracteres Latin-1, que está embutido apropriadamente no conjunto Unicode, conforme mostrado na Tabela 5.4.

| **Tabela 5.4** Amostra do Conjunto de Caracteres Unicode UTF-16

Valor Unicode (hex)	Caracteres	Valor Unicode (hex)	Caracteres
\u0000-\u007F	Conjunto ASCII	\u0F00-\u0FBF	Tibetano
\u0100-\u024F	Latim estendido	\u10A0-\u10FF	Georgiano
\u0300-\u036F	acentuação	\u2000-\u206FF	pontuação
\u0300-\u036F	Grego	\u20A0-\u20CF	símbolos monetários
\u0400-\u04FF	Cirílico	\u2190-\u21FF	setas
\u0530-\u058F	Armênio	\u2200-\u22FF	símbolos matemáticos
\u0590-\u05FF	Hebreu	\u2300-\u23FF	símbolos técnicos
\u0600-\u06FF	Árabe	\u2440-\u245F	OCR
\u0B80-\u0BFF	Tâmil	\u25A0-\u25FF	símbolos geométricos
\u0C00-\u0C7F	Telugu	\u2700-\u27BF	Dingbats
\u0C80-\u0CFF	Kannada	\u30A0-\u30FF	Katakana
\u0E00-\u0E7F	Tailandês	\u3200-\u32FF	letras CJK
\u0E80-\u0EFF	Laos	\uAC00-\uD7A3	Hangul

5.4 TIPOS COMPOSTOS

Nas primeiras seções, discutimos tipos de dados que são construídos a partir dos básicos. Os tipos de dados construídos primários incluem enumerações, ponteiros, *strings* de caracteres, matrizes, registros e uniões (ou registros de invólucro variante).

5.4.1 Enumerações

Outro tipo que é bastante relacionado ao tipo inteiro é a *enumeração*. Suportadas por muitas linguagens, enumerações fornecem um meio de atribuir nomes a uma série de valores inteiros. Isso permite que os nomes sejam usados nos programas no lugar dos seus valores integrais associados, melhorando a legibilidade do texto do próprio programa.

C, C++, Java e Ada suportam enumerações de maneiras semelhantes. Por exemplo, a seguinte declaração C/C++:

```
enum dia {segunda, terça, quarta, quinta,
   sexta, sábado, domingo};
enum dia meuDia = quarta;
```

define um tipo `dia` integral e uma variável `meuDia` inicializada com o valor `quarta`. Em C/C++ os valores desse tipo são tratados como se fossem os inteiros 0, . . . , 6 respectivamente, de modo que a variável `meuDia` pode ser tratada como se fosse uma variável `int` comum.

Um tipo de enumeração em Java 1.5 é mais poderoso do que a sua contrapartida em C, C++ e Ada. Analise o fragmento de código em Java da Figura 5.3. Esse código define `Dia` como um tipo `enum`, que é uma subclasse da classe Java `Enum`. Os valores `Segunda`, `Terça`,. . . , `Domingo` são ordenados implicitamente pela ordem de sua aparição na definição, como também acontece em C/C++ e Ada. Cada tipo `enum` de Java é `Comparable` e `Serializable` e herda os métodos da classe Java `Object`, como `to`*`String`*, que é necessário na última linha da Figura 5.3. (Veja no Capítulo 13 uma discussão mais completa sobre classes.)

```
import java.io.*;
...
enum Dia {Segunda, Terça, Quarta, Quinta, Sexta, Sábado, Domingo};
...
for (Dia d : Dia.values()) {
   System.out.println(d);
}
```

| Figura 5.3 Exemplo de `enum` em Java 1.5

A vantagem dos tipos de enumeração, de modo geral, é que eles permitem que o código fique mais legível. Se as enumerações não estiverem disponíveis (ou não usadas), um tipo como `dia` deve ser simulado associando-se seus valores com os inteiros 0–6 e usando atribuições e operadores inteiros. Na versão em Java, o uso de `enum` torna o código ainda mais confiável, já que toda a verificação de tipos extensiva de Java é aplicada naturalmente a todos os objetos `enum` em tempo de compilação.

5.4.2 Ponteiros

Um *ponteiro* é um valor que representa uma referência ou um endereço de memória. Ele fornece um nível de indireção na referenciação da memória que nenhum outro tipo de dados possui. Ponteiros são usados comumente em C, C++, Ada e Perl.

C fornece duas operações com ponteiros: o operador *endereço de* (unário `&`) recebe uma variável como argumento e retorna o endereço dessa variável, e o operador de *desreferenciação* (unário`*`) recebe uma referência e produz o valor dessa referência.

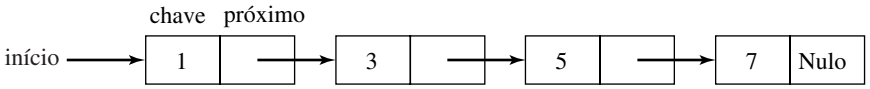
Para ilustrar o uso de ponteiros, analise a lista encadeada definida por um `Nó` em C/C++:

```
struct Nó {
  int chave;
  struct Nó* próximo;
};
struct Nó* início;
```

Aqui, um `Nó` é um par de valores, um inteiro `chave` e um ponteiro `próximo`. Assim, `Nó` é uma estrutura de dados recursiva. O ponteiro `início` se refere ao primeiro `Nó` da lista. Um exemplo de uma lista encadeada é mostrado na Figura 5.4.

Ponteiros são, muitas vezes, considerados a perdição do desenvolvimento de software confiável, já que programas que usam ponteiros tendem a ter mais erros. Em especial, algumas linguagens (por exemplo, C) requerem que o programador gerencie explicitamente os ponteiros tanto para alocar quanto para desalocar estruturas dinâmicas de dados. Entretanto, esse tipo de *gerenciamento dinâmico de memória* é tão complexo que a maioria dos programadores não o executa bem. Esse tópico é mais explorado no Capítulo 11.

Os ponteiros são especialmente problemáticos em C porque referências a matrizes e ponteiros são consideradas equivalentes. Por exemplo, as duas funções apresentadas na Figura 5.5 são semanticamente idênticas. Em especial, o uso de um ponteiro em uma declaração de incremento dentro de um laço `for` da função à direita é mais crítico do que a sua contrapartida à esquerda.



| Figura 5.4 Uma Lista Encadeada Simples em C

Apesar disso, o código à direita ilustra o relacionamento entre um índice de matriz e um ponteiro, ou seja, se `a` for uma matriz, então a referência `a` é a mesma do endereço de `a[0]`. Isso leva à regra que se `E1` ou `E2` for um tipo ponteiro (por exemplo, seu valor for um endereço), então:

$$E1[E2] = *((E1) + (E2))$$

em que o unário * é o operador de desreferenciação. Por exemplo, usando a Figura 5.5, se `a` for do tipo `float[ ]` e `i` for um `int`, então:

```
a[i] = *(a + 1) = i[a]
```

Assim, C não faz distinção efetiva entre um índice `i` de matriz e um ponteiro referenciando o elemento de índice `i` de uma matriz. Por motivo de clareza do programa, a indexação de matrizes deve ser preferida, já que é menos propensa a erros do que a aritmética de ponteiros.

Um exemplo clássico em C do uso de ponteiros para a indexação de matrizes é a implementação da função `strcpy` apresentada na Figura 5.6. A decifração desse código se baseia no conhecimento de que as *strings* `p` e `q` são matrizes de caracteres que terminam em um caractere NUL, que é interpretado como `false`. Nesse processo, o código não evita que a operação de cópia avance o limite da matriz `p`, o que ocorrerá se o bloco de armazenamento alocado para `p` for menor do que o comprimento da *string* `q`.

Embora C/C++ e Ada não tenham encontrado uma forma de remover ponteiros do seu vocabulário, outras linguagens, incluindo as linguagens funcionais e lógicas e algumas orientadas a objetos, como Java, o fizeram. Java, linguagens funcionais como Scheme e Haskell e a linguagem lógica Prolog fazem uso intenso de ponteiros, embora eles não façam parte do vocabulário dessas linguagens. Remover ponteiros explícitos de uma linguagem não elimina a capacidade de realizar alocação e desalocação dinâmica de memória de estruturas complexas, como veremos no Capítulo 14. Estudaremos o gerenciamento dinâmico de memória no Capítulo 11.

```
float sum(float a[ ], int n)
{
  int i;
  float s = 0.0;
  for (i = 0; i < n;i++)
    s += a[i];
}
```

```
float sum(float *a, int n)
{
  int i;
  float s = 0.0;
  for (i = 0; i < n; i++)
    s += *a++;
}
```

| Figura 5.5 Equivalência de Matrizes e Ponteiros em C/C++

```
void strcpy (char* p, char* q) {
   while (*p++ = *q++) ;
}
```

| Figura 5.6 A Função `strcpy`

5.4.3 Matrizes e Listas

Matrizes são seqüências indexadas de valores; em uma linguagem estaticamente tipada, todos os valores de uma matriz devem ser do mesmo tipo. Aqui está a sintaxe de algumas declarações exemplo de matrizes:

```
int A[10];
float C[4][3];
char S[40];
```

A variável `A` é uma matriz unidimensional de valores do tipo `int`, enquanto a variável `C` é uma matriz bidimensional de valores do tipo `float`, composta de quatro linhas de três entradas cada. Embora algumas linguagens mais antigas (por exemplo, Fortran) re*string*issem o número máximo de dimensões de uma matriz, a maioria das linguagens atuais não o faz.

Tecnicamente, linguagens como C re*string*em as matrizes a uma dimensão, o que é sugerido pela declaração anterior de `C`. Todavia, qualquer declaração de matriz unidimensional pode ser estendida para produzir uma "matriz de matrizes". Por exemplo, a matriz `C` declarada anteriormente é tecnicamente uma matriz de quatro elementos. Observe a sintaxe estranha para se declarar o número de linhas e colunas em `C`; essa estranheza é mantida em C++ e Java também. Uma vantagem desse projeto é que é possível criar uma matriz não retangular bidimensional, ou seja, uma matriz bidimensional na qual cada linha tenha um número diferente de colunas.

Todas as matrizes anteriores possuem limites estáticos em cada dimensão. Muitas linguagens permitem a criação de matrizes cujos tamanhos em cada dimensão podem ser determinados em tempo de execução. Por exemplo, em Java, todas as matrizes têm seu tamanho estabelecido em tempo de execução, mas, uma vez que foram criadas, seu tamanho fica fixo.

O operador básico de matrizes é o de indexação, que referencia uma única entrada na matriz por intermédio de seu índice. Exemplos incluem `a[3]` e `c[i+1][j-1]`, usando as matrizes declaradas anteriormente. Em linguagens de estilo C, o índice estabelecido para qualquer dimensão é $\{0, \ldots, n - 1\}$, no qual n é o tamanho dessa dimensão. Em contraste, Fortran usa $\{1, \ldots, n\}$, enquanto Pascal e Ada permitem a declaração explícita dos limites inferior e superior de cada dimensão. Por exemplo, as declarações anteriores em Ada poderiam aparecer como:

```
a : array(-4..5) of INTEGER;
c : array(1..4, 0..2) of FLOAT;
s : array(1..40) of CHARACTER;
```

As matrizes de Pascal têm tamanho estático, com limites estáticos para cada dimensão. Em tais linguagens, os limites da matriz estática são parte do tipo dessa matriz, o que (Stroustrup, 1994) chamou de "camisa-de-força".

Matrizes em Ada também possuem tamanho estático, mas com um pouco mais de flexibilidade na declaração do tamanho. Esse recurso é chamado de "sem restrição", o que significa

que a faixa de valores do índice para uma matriz declarada é determinada implicitamente pelo número de elementos no seu valor inicial. Por exemplo:

```
a : array(INTEGER range <>) of INTEGER :=
      (1 | 10 => 12, 2 .. 8 => 0, 9 => 33);
```

inicializa a matriz `a` com a faixa de índices 1 . . . 10 e os valores iniciais 12, 0, 0, . . . , 0, 33, 12.

Linguagens como Pascal permitem que os índices sejam de qualquer um dos tipos integrais, incluindo `int` (de todos os tamanhos), `char` e `enum`. O tipo `char`, quando visto através da sua codificação, pode ser tratado da mesma forma que outro tipo integral.

As linguagens Ada e Java verificam o valor de cada índice da matriz para assegurar que está na faixa válida, embora em compiladores Ada a verificação possa requerer uma opção de compilação. Nem C nem C++ obrigam a verificação em tempo de execução, e quase nenhum compilador C/C++ implementa a verificação de índices, nem mesmo como opção. Devido à equivalência de matrizes e ponteiros, não fica claro como a verificação de índices poderia ser implementada nessas linguagens (veja exemplos na Seção 5.4.2).

Questões de Implementação A fase semântica de um compilador constrói um bloco de informações para cada matriz, incluindo (dependendo da linguagem): tipo do elemento da matriz, tamanho do elemento da matriz, tipo do índice para cada dimensão, faixa ou tamanho dos índices para cada dimensão. Em especial, essas informações são necessárias para se calcularem os endereços de memória para expressões indexadas como `A[i]` e para executar verificações de faixa de índices em tempo de execução, caso seja necessário. Essas informações são chamadas de *descritor de vetor* e podem existir apenas em tempo de compilação ou parcialmente em tempo de execução, dependendo da linguagem.

Como exemplo, analise a declaração de matriz em Java:

```
int [ ] A = new int(n);
```

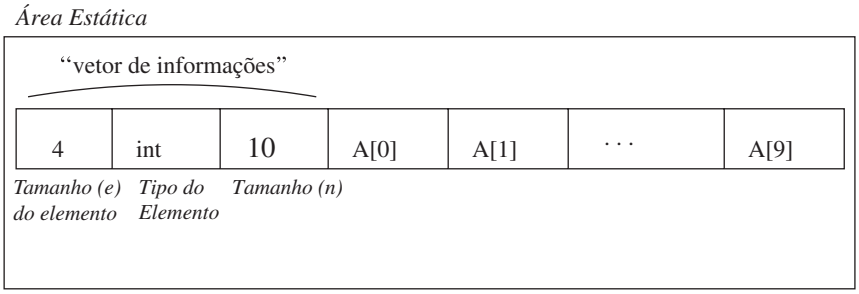
Em tempo de execução, cada referência na forma `a[i]` deve ter um valor de índice `i` entre 0 e $n - 1$, e cada valor atribuído a `a[i]` deve ser do tipo `int`. Já que o valor de `i` ou de `n` não pode ser determinado antes do tempo de execução, a parte de tempo de execução do descritor de vetor deve conter o tamanho da matriz. Na verdade, Java disponibiliza essas informações como `a.length`.

Em contrapartida, C e C++ não requerem verificação de índices de matrizes em tempo de execução. Transbordamentos de índices em matrizes em C/C++ são uma fonte de erros em tempo de execução em grandes sistemas de software; muitas vezes, esses erros aparecem com "falhas de proteção de memória". *Overflows de buffers* são um tipo de transbordamento de indexação que é explorado por hackers para entrar em sistemas. Assim, linguagens fortemente tipadas como Java e Ada, que forneçam verificação de faixas de índices, têm vantagens sobre C/C++ quanto à confiabilidade de software.

Em Java, um descritor de vetor poderia conter o tamanho *e* do elemento, o tipo do elemento, o número *n* de elementos e o endereço do primeiro elemento da matriz para facilitar cálculos de indexação. Por exemplo, o descritor de vetor e o *layout* da matriz `A` quando `n = 10` são mostrados na Figura 5.7, supondo que um `int` ocupe 4 bytes.

O endereço de memória do primeiro elemento, *addr(a[0]),* pode ser usado para calcular o endereço de qualquer outro elemento *addr(a[i])* com tamanho *e* de elemento, por intermédio da seguinte fórmula:

$$addr(a[\mathrm{i}]) = addr(a[0]) + e \cdot i$$



| Figura 5.7 Alocação de Memória para a Matriz Unidimensional `A`

Por exemplo, o endereço de memória de `a[2]`, supondo que o tamanho de um `int` seja de 4 bytes, é calculado como:

$$addr(a[2]) = addr(a[0]) + 4 \cdot 2 = addr(a[0]) + 8$$

Esse cálculo é generalizado para matrizes de duas ou mais dimensões, estendendo o descritor de vetor com informações suficientes para permitir o cálculo do endereço de um elemento individual. Por exemplo, uma alocação estática de memória para a matriz bidimensional `C` anterior é ilustrada na Figura 5.8.

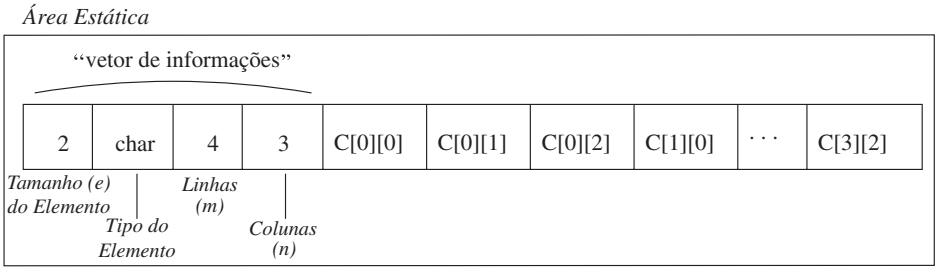
Aqui, observamos que `C` possui quatro linhas e três colunas, elementos individuais são armazenados em locações contíguas de memória, uma linha por vez, e a matriz bidimensional é retangular (garantidamente pelo tamanho da alocação estática). Isso é chamado de *ordenação linha-coluna*. O cálculo do endereço de `C[i][j]`, dados os valores para os índices `i` e `j`, pode ser definido da seguinte forma geral, considerando novamente o número de unidades endereçáveis (bytes) que um elemento requer:

$$addr(C[i][j]) = addr(C[0][0]) + e \cdot (ni + j)$$

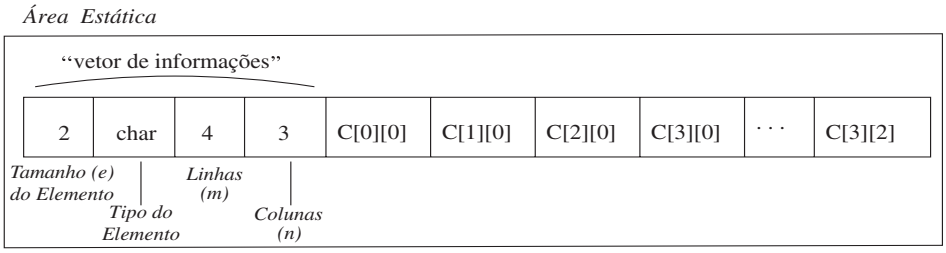
Por exemplo, o endereço de `C[2][1]` é calculado por:

$$\begin{aligned} addr(C[2][1]) &= addr(C[0][0]) + 2 \cdot (3 \times 2 + 1) \\ &= addr(C[0][0]) + 14 \end{aligned}$$

Por motivos históricos, Fortran armazena os elementos de suas matrizes bidimensionais na ordem *coluna-linha*, ou seja, todos os elementos da primeira coluna são armazenados em



| Figura 5.8 Alocação de Memória para a Matriz `C` **na Ordem Linha-Coluna**



| Figura 5.9 Alocação de Memória para a Matriz `C` na Ordem Coluna-Linha

locações contíguas de memória, depois, todos os elementos da segunda coluna, e assim por diante. Por exemplo, se nossa matriz de exemplo `C` fosse declarada em um programa em Fortran, seus elementos seriam alocados na memória conforme mostrado na Figura 5.9.

Com essa mudança, o cálculo do endereço de `C[i][j]` agora se torna:

$$addr(C[i][j]) = addr(C[0][0]) + e \cdot (mj + i)$$

Por exemplo, o endereço de `C[2][1]` muda para:

$$\begin{aligned} addr(C[2][1]) &= addr(C[0][0]) + 2 \cdot (4 \times 1 + 2) \\ &= addr(C[0][0]) + 12 \end{aligned}$$

Em linguagens fortemente tipadas, a verificação de faixas para referências de matrizes não pode, de modo geral, ocorrer até o tempo de execução, já que os valores de índices `i` e `j` variam durante a execução do programa. Para linguagens com indexação de origem em 0, o seguinte código de verificação de faixa deve ser executado para as matrizes `a` e `c` declaradas anteriormente:

- Para cada referência `A[i]`, verifique se $0 \leq i < n$.
- Para cada referência `C[i][j]`, verifique se $0 \leq i < m$ e $0 \leq j < n$.

Em algumas situações, é possível garantir em tempo de compilação que um índice inválido não possa ocorrer. Analise o seguinte laço em Java:

```
double sum = 0.0;
for (int i = 0; i < A.lenght; i++)
    sum += A[i];
```

A faixa de valores do índice `i` é 0, . . . , *A.length* − 1, que é precisamente a faixa de subscritos válidos para a matriz `A`.

Em linguagens como Ada, que permitem indexação que começa com valores diferentes de zero, o descritor de vetor deve conter informações adicionais para permitir o cálculo efetivo de endereços e a verificação de faixa em tempo de execução para referências como `A[i]` e `C[i][j]`. Essas informações incluem o limite inferior (não zero) do índice em cada dimensão da matriz, assim como o tamanho dessa dimensão ou o limite superior do seu índice. Da mesma forma que Java, Ada disponibiliza essas informações para verificação em tempo de execução.

Em algumas linguagens, todos os elementos de uma matriz são automaticamente pré-inicializados quando são declarados; em outras, não. Por exemplo, Java segue a convenção de que matrizes de números (`int`, `float`, `short`, `long` e `double`) têm todos os seus elementos

pré-inicializados com 0, matrizes de char são pré-inicializadas com `'\u0000'`, matrizes de referências a objetos são pré-inicializadas com `null`, e matrizes de valores boleanos são pré-inicializadas com `false`.

Em algumas linguagens, o programador pode inicializar os elementos de uma matriz quando essa matriz é declarada. A seguinte declaração Java inicializa todos os seis números inteiros da matriz 2 x 3 `T` com o valor inteiro 1:

```
int[][] T = {{1, 1, 1},{1, 1, 1}};
```

Listas e Fatias Muitas vezes, é desejável em programação examinar uma seção inteira de uma matriz de uma só vez, como uma porção contígua de uma linha ou uma coluna inteira de uma matriz bidimensional. Por exemplo, considere a seguinte declaração Python:

```
b = [33, 55, 'hello', 'R2D2']
```

Aqui a matriz `b` (conhecida como uma *lista* em Python) é *heterogênea*, já que os valores de diferentes entradas podem ter tipos diferentes. Listas em Python são indexadas começando em 0, de modo que a referência a `b[0]` retorna o valor 33.

Uma *fatia* de uma lista em Python é uma série contígua de entradas, especificada pelo seu índice inicial e comprimento. Por exemplo, a fatia `b[0:3]` se refere à sublista `[33, 55, 'hello']`. Ada também suporta o fatiamento de matrizes.

Uma coleção rica de operações que usam fatias de listas está disponível em Python. Por exemplo, os operadores + e = designam a concatenação e a atribuição de listas, respectivamente. Assim, o seguinte diálogo[8] em Python apresenta resultados interessantes:

```
>>> b
[33, 55, 'hello', 'R2D2']
>>> b[2:2] + 2*['C3PO', 'Luke']
['hello', 'R2D2', 'C3PO', 'Luke', 'C3PO', 'Luke']
>>> b[:0] = b
[33, 55, 'hello', 'R2D2', 33, 55, 'hello', 'R2D2']
```

A primeira operação constrói uma lista concatenando duas, a segunda, contendo duas cópias da lista `['C3PO', 'Luke']`. A segunda operação insere uma nova cópia da lista `b` no início da lista `b`, efetivamente duplicando seu comprimento.

Muitas oportunidades para usar listas dinâmicas e suas operações ocorrem na programação funcional. Revisitaremos esse tópico com mais detalhes no Capítulo 14, sobre programação funcional, em que as listas são usadas extensivamente para resolver diversos problemas de programação.

5.4.4 Strings

As *strings* são tão fundamentais em programação que agora são suportadas diretamente pela maioria das linguagens de programação. Historicamente, nem Fortran nem Algol 60 tinham algum suporte a *strings*; Cobol tinha *strings* de tamanho estático, com apenas um conjunto limitado de operações. C, sendo uma linguagem relativamente moderna, é incomum, já que não suporta explicitamente *strings* de caracteres como um tipo de dados distinto.

8. Aqui, os sinais >>> indicam um *prompt* do interpretador Python.

Em vez disso, *strings* em C são codificadas como matrizes de caracteres ASCII, um caractere por byte, com um caractere NUL adicional (todos bits zero) no final. Por exemplo, para declarar a variável *string* `greeting` e atribuir o valor "hello", o programador C escreveria:

```
char greeting[] = "hello";
```

mas a *string* "hello" requer 6 bytes, um para cada um dos cinco caracteres mais um para o caractere NUL, que marca o final da *string*. Além do comprimento declarado da matriz de *strings* de caracteres, todos os caracteres que seguem o caractere ASCII NUL na memória serão valores desconhecidos ou indefinidos. Transformações sobre esse valor são efetuadas em um caractere por vez.

Assim, o projeto de *strings* em C é propenso a erros porque:

1. O programador deve lembrar-se de alocar pelo menos 1 byte extra para o caractere NUL.
2. A atribuição `a = b` em que `a` e `b` são *strings* é errônea; em vez disso, você deve usar a função de biblioteca `strcpy`. É responsabilidade do programador assegurar que o armazenamento alocado para `a` seja capaz de guardar todos os caracteres em `b`, até, e incluindo, o caractere NUL.

Ada fornece *strings* de tamanho fixo, que, como C, são matrizes de caracteres. Entretanto, diferentemente de C, uma *string* Ada de tamanho 6, por exemplo, sempre armazena seis caracteres. Assim, a atribuição da *string* "hello" só pode ser feita em uma variável *string* de tamanho declarado 5, sem incorrer em erro em tempo de execução.

Em contraste, uma *string* se comporta como um tipo interno em Java, Perl e Python, capaz de armazenar um número ilimitado de caracteres. Nessas linguagens, uma variável *string* não é declarada com um tamanho, como em C e Ada, nem o programador sabe (ou se importa) como as *strings* são armazenadas em tempo de execução. Aqui está a declaração anterior reescrita em Java:

```
String greeting = "hello";
```

Java, Perl e Python fornecem amplas bibliotecas de funções e operações de *strings*, incluindo concatenação, conversão de maiúsculas e minúsculas, busca de padrões e "sub*strings*". Java e Python fornecem muitas operações por intermédio de bibliotecas de classes.

5.4.5 Estruturas

Uma *estrutura* ou um *registro* é um conjunto de elementos de tipos potencialmente diferentes. Assim, estruturas são como listas em Python no sentido de que os tipos dos seus elementos podem variar. Entretanto, elas são diferentes de matrizes e listas porque seus elementos possuem nomes distintos e são acessados por nome, em vez de serem acessados por índice. Os elementos individuais de uma estrutura são, muitas vezes, denominados *campos*.

As estruturas apareceram primeiro em Cobol e PL/I, após serem omitidas tanto em Fortran quanto em Algol 60. A forma de abstração corrente de uma estrutura se originou tanto em linguagens como Pascal quanto em linguagens de estilo C, embora sua sintaxe concreta divirja. Java omite as estruturas completamente, tendo-as efetivamente agrupado na noção mais geral de uma classe. C++ possui tanto estruturas quanto classes, conservando estruturas para manter compatibilidade com C.

Como exemplo, analise a seguinte estrutura e a declaração de uma variável em C que modela um funcionário individual em uma organização:

```
struct employeeType {
    int id;
    char name[26];
    int age;
    float salary;
    char dept;
};
struct employeeType employee;
```

Cada campo dentro de uma estrutura pode ter um tipo de dado diferente. Nesse exemplo, vemos que os tipos são `int`, `char[]`, `int`, `float` e `char`. Entretanto, cada nome de campo dentro de uma estrutura deve ser único.

Campos individuais dentro de uma estrutura são geralmente alocados em um bloco contíguo. O número de bytes realmente alocados e a ordem dos campos alocados dependem da máquina e do compilador. Por exemplo, alguns compiladores alocam os campos na ordem reversa da declaração. Além disso, algumas máquinas requerem, por exemplo, que um `float` ou `int` de 32 bits seja alocado em um endereço que seja múltiplo de 4. Assim, embora a alocação mínima para `employeeType` seja 39 bytes, sob as suposições recém-mencionadas, a alocação requer 44 bytes. Programas que fazem suposições sobre ordem de armazenamento e tamanho encontrarão problemas de portabilidade.

A única operação significativa sobre estruturas, além da atribuição, é o operador de referência à estrutura (seleção de campos), denotado por um ponto. A referência do operador de estrutura associa da esquerda para a direita; seu primeiro operando é uma expressão que identifica a estrutura, e seu segundo operando é um identificador *id* que identifica um campo da estrutura. Como exemplo, a referência `employee.age` seleciona o campo `age` da variável estrutura `employee`. Assim, uma declaração de atribuição como:

```
employee.age = 45;
```

é válida porque `employee` é uma `struct` que contém o campo `age` do tipo `int`.

Em linguagens orientadas a objetos, uma classe pode ser vista como uma extensão de uma estrutura, porque encapsula funções com variáveis de instâncias (veja o Capítulo 13). As próprias variáveis de instância de uma classe são comparáveis com os campos de uma estrutura, ignorando, por enquanto, que algumas linguagens orientadas a objetos podem limitar sua acessibilidade.

5.4.6 Registros Variantes e Uniões

Anos atrás, a memória era escassa. Os programadores tinham de ser muito econômicos com a forma pela qual definiam estruturas de matrizes e registros de modo que uma quantidade mínima de armazenamento fosse desperdiçada. Parte dessa economia poderia ser realizada por intermédio de *registros variantes*, nos quais dois ou mais campos diferentes compartilham o mesmo bloco de memória.

Fortran foi a primeira linguagem a suportar o conceito de registros variantes, na forma de seu comando **`EQUIVALENCE`**. Esse comando acabou sendo um recurso muito inseguro, já que os programadores tinham de manter o processamento das duas variáveis separadas

por todo o tempo de execução do programa. Por exemplo, se A e B forem definidas para compartilhar o mesmo bloco de memória apenas por motivo de economia, todo o código que acessa A não pode sobrepor em tempo todo o código que acessa B.

Linguagens posteriores, incluindo Pascal e Ada, definiram registros variantes de uma forma mais restrita, o que levou a um estilo mais disciplinado de programação, embora ainda suportasse a economia de memória permitida pelo comando **EQUIVALENCE**.

Apesar de os registros variantes fornecerem certo nível de clareza nas definições de tipos, conforme ilustrado anteriormente, eles parecem ser menos amplamente usados em aplicações modernas. Um motivo pode ser que a memória não é claramente escassa como o era na década de 1960, quando Fortran e Cobol eram as linguagens imperativas dominantes.

Em linguagens como C, um tipo *união* realiza o objetivo de um registro variante, já que permite duas ou mais formas de visualização do mesmo local de armazenamento. Analise a seguinte declaração de tipo em C/C++:

```
union myUnion {
        int i;
        float r;
};
union myUnion u;
```

Qualquer variável do tipo `myUnion`, como `u`, permite duas formas de acesso aos mesmos locais de armazenamento. Sob a variante `u.i`, o local é visto como um inteiro, enquanto sob a variante `u.r` o local é visto como um valor de ponto flutuante. Na programação de sistemas de software básico (incluindo escrita de compiladores, partes de sistemas operacionais, utilitários como o *grep*), muitas vezes, tarefas como o acesso em nível de bit a números de ponto flutuante são desejáveis.

Todavia, o tipo de registro variante ou união cria uma fraqueza no sistema de tipo de uma linguagem. Usando o exemplo anterior, o mesmo padrão de bits pode ser tratado como um valor `int` ou `float`, o que cria precisamente o problema discutido na Seção 5.1. Linguagens que têm essa falha nos seus sistemas de tipos incluem C, Pascal, C++ (por causa da sua compatibilidade com C) e Modula.

Linguagens como Pascal também têm uma união *discriminada com rótulos*, na qual um campo rótulo é declarado. O rótulo em Ada é usado pelo compilador para registrar o tipo da variante que está ativa no momento, resolvendo, assim, a falha citada anteriormente no sistema de tipo. Analise a seguinte versão em Ada do exemplo anterior, no qual um campo rótulo `b` foi adicionado ao tipo de registro `union`:

```
type union(b : boolean) is
     record
          case b is
             when true =>
                  i : integer;
             when false =>
                  r : float;
          end case;
     end record;
```

```
tagged : union;
begin
     tagged := (b => false, r => 3.375);
     put(tagged.i);
```

A sintaxe de Ada requer que tanto a variante quanto o rótulo sejam atribuídos em uma declaração; qualquer tentativa de atribuir a variante ou o rótulo separadamente resulta em um erro em tempo de compilação. A primeira declaração após a declaração `type` configura o rótulo e atribui à variante `r` um valor consistente com essa definição do rótulo.

Assim, Ada pode verificar cada referência a uma variante em tempo de execução para assegurar que a referência seja consistente com a configuração atual do rótulo. Observe que a última linha acima viola o sistema de tipo acessando o campo variante `i` do registro sem verificar que seu uso é consistente com a definição corrente do rótulo. Assim, a referência `tagged.i` gera um erro em tempo de execução, já que o rótulo indica que o valor corrente é um número de ponto flutuante, não um inteiro. Substituir a referência a `i` na declaração `put` por `r` elimina o erro de tipo.

Os programadores, hoje, parecem não estar cientes do tipo união e raramente usam essa construção em programas. Entretanto, simular um tipo união usando uma estrutura comum como base para diversas realizações concretas é uma prática popular. Analise o seguinte exemplo em C++ que define uma estrutura `Value`, que armazena um único valor do tipo `int`, `float`, `char` ou `bool`:

```
struct Value {
// Value = int intValue | boolean boolValue |
//          float floatValue | char charValue
   enum Type type;
   int intValue; boolean boolValue
   float floatValue | char charValue
}
```

Essa `struct` imita a classe abstrata `Value` de Java que aparece na sintaxe abstrata de Clite (veja a Seção 2.5.3). O `enum type` pode receber um de cinco valores indicando que tipo de valor é armazenado, ou *indefinido*, indicando que nenhum valor válido é armazenado.

5.5 TIPOS DE DADOS RECURSIVOS

Na Seção 5.4.2, vimos como os elementos de uma lista encadeada poderiam ser definidos recursivamente em C, usando a estrutura `Node` como base. A recursão é uma ferramenta natural para situações como essa, porque o números de `Node`s em uma lista encadeada varia em tempo de execução, mas a natureza de cada `Node` não.

Além das estruturas, algumas linguagens de programação suportam a definição recursiva de tipos de dados de uma forma mais genérica.[9] Para ilustrar, analise as seguintes definições de tipo em Haskell relacionadas à sintaxe abstrata de Clite que foi introduzida no Capítulo 2.

9. Nesta seção, introduzimos tipos de dados recursivos usando Haskell; exploraremos sua utilidade com mais detalhes no Capítulo 14.

```
data Value = IntValue Integer | FloatValue Float |
             BoolValue Bool | CharValue Char
             deriving (Eq, Ord, Show)
data Expression = Var Variable | Lit Value |
             Binary Op Expression Expression |
             Unary Op Expression
             deriving (Eq, Ord, Show)
type Variable = String
type Op = String
type State = [(Variable, Value)]
```

A primeira linha nessa definição é um exemplo de definição de *tipo algébrico* em Haskell, no qual o novo tipo `Value` é definido como um valor `Integer`, um `Float`, um `Bool` ou um `Char` (esses são os tipos básicos em Haskell, assim como na maioria das outras linguagens).[10]

A segunda linha nessa definição é semelhante; ela define recursivamente o tipo de dado `Expression` como tendo uma entre quatro formas alternativas. `Expression` é definida recursivamente, já que qualquer número de `Expression`s Clite pode ser aninhado.

As próximas duas linhas definem os tipos Variable e Op como tipos particulares do tipo básico *`String`*. A última linha define o tipo `State` como uma lista de pares, cada um tendo uma `Variable` e um `Value`.

Com essas definições, diversas funções que analisam ou interpretam `Expressions` abstratas em Clite podem ser escritas como transformações sobre esse tipo de dado definido recursivamente. Esse exemplo particular será mais explorado no Capítulo 8, no qual é examinada a interpretação semântica de linguagens de programação, e na Seção 14.3.8, na qual é examinada a implementação de Clite em Haskell.

5.6 FUNÇÕES COMO TIPOS

Em linguagens como Fortran, C/C++, Ada e Java, há muitos tipos de entidades que não são "cidadãs de primeira classe" da linguagem. Um exemplo óbvio é uma estrutura de controle como um comando *if* ou *while*, pelo fato de que uma instância de tal comando não pode ser atribuída como um valor a uma variável e nem passada como um argumento. Em muitas linguagens, as funções não são "cidadãs de primeira classe".

A necessidade de se passar funções como argumentos surgiu no início da história de Fortran. Um exemplo incluía rotinas de desenho que geravam gráficos de funções como $y = f(x)$. Outro exemplo comum inclui resolvedores de raiz para $y = f(x)$. Fortran permitia funções como parâmetros, mas não permitia nenhum tipo definido pelo programador, incluindo funções. Assim, os compiladores Fortran não podiam verificar em tempo de compilação se uma função apropriada estava sendo passada.

Pascal, por outro lado, permitia tipos função. Assim, um resolvedor de raiz geral (como o método de Newton) poderia declarar o tipo de argumento requerido:

```
function Newton(a, b: real; function f: real): real;
```

10. A cláusula `deriving` (`Eq`, `Ord`, `Show`) declara que esse novo tipo herda a igualdade, a ordenação e as características de exibição dos seus tipos componentes, permitindo-nos, assim, usar as funções de igualdade (==), ordenação (<) e exibição (`show`) em qualquer um de seus valores.

Os argumentos a e b são valores de x em ambos os lados da raiz. Também nos é mostrado que a função f retorna um número real, mas o número de argumentos de f não é especificado. Esse recurso foi amplamente omitido da maioria dos compiladores Pascal.

Um uso de uma classe de interface em Java resolve precisamente esse problema (uma discussão mais completa sobre classes de interface ocorre na Seção 13.2.8). Tal definição de interface é a seguinte:

```
public interface RootSolvable {
       double valueAt(double x);
}
```

Qualquer classe que implemente essa interface poderia então ser passada como um argumento para o nosso descobridor de raiz de Newton, cuja assinatura seria:

```
public double Newton(double a, double b, RootSolvable f);
```

Apesar dessa capacidade, as funções ainda não são cidadãs de primeira classe em Java. Por exemplo, uma função em Java não pode construir uma nova função e retorná-la como um valor. Em vez disso, ela está limitada a retornar apenas funções que já existem.

A mensagem-chave aqui é a de que uma função deve ser um tipo. A generalidade que esse recurso traz à programação é significativa, como veremos no Capítulo 14.

5.7 EQUIVALÊNCIA DE TIPOS

Às vezes, precisamos saber quando dois tipos se equivalem. Por exemplo, na descrição de Pascal (Jensen e Wirth, 1975, p. 149), lemos:

> O comando de atribuição serve para substituir o valor corrente de uma variável por um novo valor especificado como uma expressão. ... A variável (ou a função) e a expressão devem ser de tipo idêntico ...

Entretanto, o Pascal Report falha ao definir o termo *tipo idêntico*. Na verdade, dois compiladores antigos ETH Pascal implementavam essa verificação usando diferentes definições. É importante definir esse termo de forma clara, para a implementação apropriada tanto da atribuição quanto da passagem de parâmetros para tipos compostos.[11]

Por exemplo, analise o seguinte exemplo em C:

```
struct complex {
      float re, im;
};
struct polar {
      float x, y;
};
```

11. Felizmente, o padrão ANSI/ISO Pascal mais tarde corrigiu esse descuido e definiu a noção do "mesmo tipo".

```
struct {
      float re, im;
} a, b;
struct complex c, d;
struct polar e;
int f[5], g[10];
```

A questão é: quais das variáveis anteriores possuem o mesmo tipo?

Sob *equivalência de nome,* dois tipos são o mesmo se tiverem o mesmo nome. No exemplo anterior, as variáveis `c` e `d` são, dessa forma, do mesmo tipo, porque compartilham o mesmo nome de tipo `struct complex`. Entretanto, `a` e `c` não são do mesmo tipo, já que têm diferentes nomes de tipo; na verdade, `a` é uma `struct` anônima. Pelo mesmo motivo, `d` e `e` também não são do mesmo tipo.

A outra forma de equivalência de tipo é a *equivalência estrutural* na qual dois tipos são o mesmo se tiverem a mesma estrutura. Para tipos registro, ter a mesma estrutura inclui o número e a ordem dos campos, assim como o nome e o tipo de cada campo. Sob equivalência estrutural, todos os `a`, `b`, `c` e `d` teriam o mesmo tipo. As variáveis `d` e `e` teriam tipos diferentes, já que os dois nomes de campos são diferentes (embora ambos tenham dois campos de ponto flutuante).[12]

Analise a questão da equivalência de tipos a partir da perspectiva de um compilador, no qual os tipos são implementados como referências a um registro variante rotulado. Se o tipo tiver um nome, o nome e o tipo (ponteiro) são colocados em uma tabela de símbolos de modo que o tipo pode ser recuperado pelo nome. De forma semelhante, uma variável e um ponteiro para seu tipo (mas não um nome de tipo) são inseridos em uma tabela de símbolos. Assim, sob a equivalência de nomes, dois tipos são o mesmo se os ponteiros referenciarem a mesma entrada na tabela de símbolos. No exemplo anterior, `a` e `b` são o mesmo tipo sob a equivalência de nomes.

Ada usa a equivalência de nomes para tipos, o que estende para matrizes e ponteiros. Na verdade, Ada proíbe sintaticamente a maioria das instâncias de tipos anônimos. C usa a equivalência de nomes para `struct`s e `union`s e a equivalência estrutural para outros tipos construídos, incluindo matrizes e ponteiros. Assim, no exemplo anterior em C, `f` e `g` são do mesmo tipo, já que ambas são uma matriz unidimensional de `int`; o tamanho não importa.

Da mesma forma que Ada, Java requer que tanto classes quanto interfaces tenham nomes. Assim, Java usa a equivalência de nomes para determinar se duas classes são a mesma. Ao usar um objeto no qual uma interface específica é requerida, Java requer que a classe declarada do objeto implemente a interface; esse é um requisito sintático e semântico.

5.8 SUBTIPOS

Um *subtipo* pode ser visto como um tipo que possui determinadas restrições colocadas sobre seus valores ou suas operações.

12. No mundo real, os tipos número complexo e coordenadas polares são, na verdade, distintos nos seus comportamentos, o que dá crédito a essa conclusão.

Em Ada, subtipos podem ser especificados diretamente para variáveis, matrizes e outras estruturas de dados. Por exemplo, analise as seguintes declarações em Ada:

```
subtype um_a_dez is Integer range 1 .. 10;
type Dia is (Segunda, Terça, Quarta, Quinta,
             Sexta, Sábado, Domingo);
subtype Fim_de_Semana is Dia range Sábado .. Domingo;
type Salário is delta 0.01 digits 9
             range 0.00 .. 9_999_999.99;
subtype Salário_Autor is Salário digits 5
             faixa 0.00 .. 999.99;
```

Aqui, o subtipo `um_a_dez` re*string*e o tipo `Integer`, o subtipo `Fim_de_Semana` re*string*e a faixa de valores do seu supertipo `Dia`, e o subtipo `Salário_Autor` re*string*e a faixa de valores do seu supertipo `Salário`. Variáveis podem ser declaradas para qualquer um desses tipos ou subtipos, com verificação integral em tempo de compilação imposta pelas suas respectivas restrições.

Em Java, subtipos são realizados por meio da hierarquia de classes, ou seja, um objeto *s* da classe *S* pode ser atribuído a um objeto *t* da clase *T*, $t = s$, desde que *S* e *T* sejam da mesma classe ou se *S* for uma subclasse de *T*. Executar tal atribuição não altera o tipo do objeto atribuído a *t* em tempo de execução. Entretanto, o compilador trata *t* como se fosse da classe *T*, no sentido de que apenas os métodos da classe *T* podem ser chamados. Assim, tal atribuição é um tipo de conversão ampliadora porque nenhuma informação é perdida.

Analise a classe Java `Number` e sua subclasse `Integer` definida em `java.lang`, junto ao seu uso no seguinte trecho de programa em Java:

```
1   Integer i = new Integer (3);
2   ...
3   Number v = i;
4   ...
5   Integer x = (Integer) v;
```

A atribuição na linha 3 é permitida porque um `Integer` é uma subclasse da classe `Number`.[13] A atribuição oposta na linha 5 requer uma conversão para evitar um erro de tipo em tempo de compilação. A conversão é verificada em tempo de execução para determinar se o objeto armazenado em `v` é da classe `Integer` ou de uma de suas subclasses. Se `Integer` não fosse uma subclasse de `Number`, a falha na conversão seria detectável em tempo de compilação e relatada como um erro de tipo.

5.9 POLIMORFISMO E GENÉRICOS

De acordo com o dicionário, o termo *polimorfismo* vem do grego e significa "que tem muitas formas". Em programação, polimorfismo é um pouco diferente da idéia de sobrecarga.

> **Definição:** Uma função ou uma operação é *polimórfica* se puder ser aplicada a qualquer um de diversos tipos relacionados e obtiver o mesmo resultado.

13. Tal atribuição é perigosa em C++ porque informações podem ser descartadas.

Uma vantagem do polimorfismo é que ele permite a reutilização de código. Por exemplo, a lógica de ordenação de uma matriz não deve necessariamente variar com o tipo dos elementos de dados que estiverem sendo ordenados. Analise a seguinte rotina de ordenação simples em Ada:

```
procedure sort (in out a : list) is
begin
    for i in a'first .. a'last-1 loop
       for j in i+1 .. a'last loop
          if a(i) > a(j) then
          begin t : integer := a(i);
             a(i) := a(j);
             a(j) := t;
          end if;
       end loop;
    end loop;
end sort;
```

As únicas operações que dependem do tipo são o operador de atribuição e a comparação maior que. A atribuição só é dependente do tipo porque necessita saber o tamanho de um objeto. Mesmo em tipos de dados simples (veja a Seção 5.3), as comparações são claramente dependentes dos tipos.

Reconhecendo a necessidade de evitar redundância e suportar polimorfismo, Ada introduziu o conceito de *genéricos* ou *modelos*. Uma *função* ou um *procedimento genérico* é um modelo que pode ser instanciado em tempo de compilação com operadores e tipos concretos. Em Ada, uma ou mais dessas rotinas pode ser reunida em um pacote e parametrizada tanto por tipo de elemento quanto pelo operador de comparação, conforme mostrado na Figura 5.10. Procedimentos genéricos como esse são exemplos de *polimorfismo paramétrico*.

Nesse caso, a ligação do argumento com um tipo é deferida do tempo de programação para o de compilação:

```
package integer_sort is new generic_sort( Integer, ">");
```

Tal adiamento na ligação cria flexibilidade, já que o procedimento de ordenação precisa ser escrito apenas uma vez, mas pode ser reutilizado para ordenar diferentes tipos de dados. Com esse recurso, não ocorre nenhuma perda de verificação estática de tipos; ele apenas adiciona complexidade à implementação da linguagem.

Analise um exemplo ligeiramente mais complicado, a saber, a implementação de uma pilha na forma de uma lista encadeada de registros. Neste caso, a única operação dependente de tipo é a atribuição, que só depende do tamanho do objeto que será armazenado na pilha. Aqui mostramos apenas a operação de empilhamento (`push`), novamente em Ada:

```
package stack_pck is
  type stack is private;
  procedure push (i : integer; in out s : stack);
  ...
```

```
private
    type stack is access node;
  type node is record
      val : integer;
      next : stack;
  end record;
end stack_pck;

package body stack_pck is
procedure push (i : integer; in out s : stack) is
begin
    node n = new node;
    n.val = i;
    n.next = s;
    s = n;
end push;
...
end stack_pck;
```

A pilha contém apenas dados do tipo inteiro. Todavia, da mesma forma que em qualquer conjunto de estruturas de dados, outros além do operador de atribuição, o código da pilha não depende do tipo dos dados armazenados na pilha.

Assim como em `sort`, Ada permite a parametrização do tipo do elemento em `stack`, conforme mostrado na Figura 5.11. O único parâmetro para esse modelo é o tipo dos dados sendo armazenados.

Ada, C++ e Java 1.5 fornecem genéricos e modelos para esse propósito.[14] O mecanismo básico de implementação em C++ pode ser pensado como uma macroexpansão, como o dispositivo `#define` do pré-processador C. Primeiro, os parâmetros reais são substituídos textualmente pelos parâmetros genéricos, e o texto resultante é então compilado.

Linguagens orientadas a objetos também suportam coleções genéricas (como uma pilha) por meio de *herança*, na qual uma classe pode ser uma subclasse de outra. Especialmente em Java, cada classe é uma subclasse da classe `Object`. Assim, todas as classes *collection* em Java armazenam `Object`s ou uma subclasse de `Object`. Simplifica mais ainda essa abordagem o fato de que cada variável de objeto armazena uma referência ou um ponteiro para esse, em vez dele próprio. Já que todos os ponteiros são do mesmo tamanho, independentemente da classe de um objeto, a implementação de uma classe *collection* não requer o uso de genéricos. Uma implementação simples de uma classe *stack* é:

14. Modelos em Java 1.5 são implementados de forma diferente de genéricos de Ada ou C++. Uma comparação adequada requer um conhecimento mais aprofundado de programação orientada a objetos, então a adiamos para o Capítulo 13.

```
public class Stack {
    private class Node {
         Object val;
         Node next;
         Node(Object v, Node n) {
            val = v; next = n;
         }
    }
    private Node stack = null;

...

    public void push(Object v) { stack = new Node(v, stack); }
}
```

```
generic
   type element is private;
   type list is array(natural range <>) of element;
   with function ">"(a, b : element) return boolean;

package sort_pck is
   procedure sort (in out a : list);
end sort_pck;

package sort_pck is
procedure sort (in out a : list) is
begin
   for i in a'first .. a'last - 1 loop
      for j in i+1 .. a'last loop
         if a(i) > a(j) then
            declare t : element;
            begin
               t := a(i);
               a(i) := a(j);
               a(j) := t;
            end;
         end if;
      end loop;
   end loop;
end sort;
end sort_pck;
```

| Figura 5.10 Ordenação Genérica em Ada

```
generic
   type element is private;

package stack_pck is
   type stack is private;
   procedure push (i : element; in out s : stack);
...

private
   type stack is access node;
   type stack is record
       val : element;
       next : stack;
   end record;
end stack_pck;

package body stack_pck is
procedure push (i : element; in out s : stack) is
begin
   node n = new node;
   n.val = i;
   n.next = s;
   s = n;
end push;
...
end stack_pck;
```

| Figura 5.11 Uma Pilha Genérica em Ada

Observe o uso da classe interna `Node`, que é, na verdade, apenas um registro com um construtor. Uma discussão mais completa sobre herança é apresentada no Capítulo 13.

Já vimos na Seção 5.4.6 um exemplo do uso de herança em uma subclasse para anular um método de uma classe mãe e fornecer assim a funcionalidade apropriada para a subclasse. Tais exemplos de *polimorfismo puro* são comuns em Java; um exemplo comum é criar um método `toString( )` em uma classe, anulando assim o da classe `Object`.

Outra abordagem, novamente de Java, usa uma `interface` para criar uma função genérica. Embora uma `interface` possa ter subclasses, podemos por enquanto vê-la apenas como um conjunto de assinaturas de métodos. Uma classe que implemente uma interface é obrigada a implementar todas as funções definidas pela interface.

Analise uma função de ordenação genérica em Java:

```
public static void sort (Comparable[ ] a) {
  for (int i = 0; i < a.lenght; i++)
         for (int j = i+1; j < a.lenght; j++)
                if (a[i].compareTo(a[j]) > 0) {
                       Comparable t = a[i];
                       a[i] = a[j];
                       a[j] = t;
                }
}
```

A lógica imita aquela do programa em Ada. Há duas diferenças importantes. Primeiro, o tipo do parâmetro formal `a` é um `Comparable`, que é a interface:

```
public interface Comparable {
   public abstract int compareTo(Object o);
}
```

Segundo, o operador maior que é substituído por uma chamada ao método `compareTo( )`.

Assim, uma matriz de objetos de qualquer classe que implemente o método `compareTo( )` pode ser usada na nossa ordenação genérica. O método `compareTo( )` deve retornar um número negativo, se o receptor da mensagem for menor que o argumento, zero, se for igual, e um número positivo, se for maior.

5.10 TIPOS DEFINIDOS PELO PROGRAMADOR

Lembre-se da definição de um tipo apresentada no início do capítulo: um tipo é um conjunto de valores e um conjunto de operações sobre esses valores. Estruturas (discutidas na Seção 5.4.5) nos permitem definir uma representação para um novo tipo. Todavia, historicamente, elas tinham dois problemas:

1. A representação não ficava escondida.
2. Nenhuma operação sobre o novo tipo poderia ser definida.

Uma tentativa de resolver esse problema foi o desenvolvimento de tipos abstratos de dados, que permitiram que a representação fosse escondida. Também é permitido que o fornecedor do tipo defina operações sobre o novo tipo. Tipos abstratos de dados são discutidos na Seção 13.1.

A programação orientada a objetos (POO) fornece um meio poderoso para os programadores definirem novos tipos. Devido ao fato de a POO ser um paradigma distinto, discutiremos seus recursos de definição de tipos extensivamente no Capítulo 13.

5.11 RESUMO

Este capítulo apresenta os tipos que são suportados em diversas linguagens de programação. Esses tipos incluem números, valores lógicos, ponteiros, *strings* de caracteres, matrizes, estruturas e funções. O relacionamento entre um tipo e sua representação de máquina também foi discutido.

Diversas características incomuns de tipos foram apresentadas, incluindo o fatiamento de matrizes, o uso de ponteiros como índices de matrizes, os conceitos de registros variantes, a diferença entre alocação estática e dinâmica de matrizes e estruturas. Tipos de dados recursivos, que são especialmente importantes na programação funcional, também são introduzidos aqui.

Este capítulo fornece, assim, fundamentos para os próximos dois capítulos, que examinam mais cuidadosamente o relacionamento entre variáveis, seus tipos e seu aparecimento em expressões e funções que transformam seus valores. Esses relacionamentos são cruciais para o entendimento dos recursos de verificação de tipos e o comportamento em tempo de execução das linguagens de programação.

EXERCÍCIOS

5.1 Para a sua linguagem e seu compilador favoritos (outra que não seja Java), liste todos os tipos básicos suportados e seus tamanhos em bytes.

5.2 Para a sua linguagem favorita que suporte um tipo boleano, responda às questões a seguir. Ele é ordenado? Ele pode ser convertido em um inteiro? Argumente com base em princípios o porquê de isso ser útil ou não.

5.3 Java não permite valores não boleanos em testes lógicos, diferentemente de outras linguagens do como C. Argumente com base nos princípios o porquê de isso ser útil ou não.

5.4 Escreva um programa para ler um valor de ponto flutuante de 32 bits, imprimir o valor e sua representação em bits. Dica: use um tipo `union`.

5.5 Qual a representação em ponto flutuante de 32 bits para 0,2? 0,5? 0,3? 1,0?

5.6 Qual a representação em ponto flutuante de 32 bits para 0,1?

5.7 Escreva um programa para ler um valor inteiro de 32 bits, imprimir o valor e sua representação em bits.

5.8 Qual a representação em inteiro de 32 bits para 0? 1? –1? 2? −2? 3? −3?

5.9 Qual a diferença entre uma máquina *big-endian* e uma *little-endian*? Mencione pelo menos uma arquitetura computacional de cada tipo. Como essas arquiteturas afetam os valores apresentados na Tabela 5.2?

5.10 Ada fornece dois operadores de resto. Qual a diferença entre eles? Para sua linguagem de máquina favorita, que operador de resto elas implementam?

5.11 Escolha uma das seguintes linguagens: Pascal, Modula, Ada, C, C++, C#, Java ou outra linguagem aprovada pelo professor. Consulte uma fonte oficial. Escreva um relatório que pesquise a definição da linguagem quanto a conversões de tipos consideradas inseguras, tanto em declarações quanto em expressões.

5.12 Na sua linguagem favorita, use inteiros de 32 bits para calcular o fatorial N de 2, 3, ... Para qual valor de N você obtém um erro? Como o erro se manifesta?

5.13 Compare `enum`s em Java 1.5 com aquelas em C/C++. Escreva um programa exemplo para cada linguagem. Qual a diferença lingüística fundamental entre as duas?

5.14 Analise o trecho em C:

```
char a[20];
char b[20];
```

O comando de atribuição `a = b` gera um erro de compilação. Apresente pelo menos duas interpretações possíveis para a atribuição e, para cada uma delas, corrija a declaração ou a atribuição conforme for necessário para que satisfaça à interpretação.

5.15 Assim como Python, Ada também suporta a operação de fatiamento de matrizes.
(a) Compare essa operação com sua contraparte em Python discutida neste capítulo.
(b) Dê três exemplos realistas do uso de fatiamento de matrizes, em Ada ou Python.

5.16 Perl e Python suportam matrizes dinâmicas. Discuta a implementação de matrizes dinâmicas na sua linguagem favorita.

5.17 Perl e Python suportam *matrizes hash* ou *matrizes associativas*. O que são elas? Dê um exemplo realista de seus usos. Como você simularia uma matriz associativa na sua linguagem favorita?

5.18 Analise as linguagens ML, Scheme e Python. Escreva um relatório breve que compare e ilustre como a noção de equivalência de tipos é usada em cada uma dessas linguagens.

5.19 Defina uma EBNF e uma sintaxe abstrata para adicionar estruturas de registros a Clite. A EBNF para referenciar um elemento de uma estrutura deve usar a notação de "ponto" discutida neste capítulo. As sintaxes concreta e abstrata de *Declaração*, *Expressão* e *Atribuição* devem ser modificadas.

5.20 Usando suas definições do exercício anterior, projete um algoritmo que determine se dois registros são estruturalmente equivalentes.

5.21 Explore a definição de tipos de dados recursivos em ML. Reescreva o exemplo em Haskell mostrado neste capítulo usando ML.

5.22 Após consultar uma fonte oficial, escreva um relatório breve que discuta o tratamento de *equivalência de tipos* em C, Perl ou Python.

5.23 Estenda a EBNF e a sintaxe abstrata de Clite para incluir estruturas ou registros. Implemente um algoritmo que determine se dois registros são estruturalmente equivalentes.

Sistemas de Tipos

6

"Eu acabei convencido da necessidade de projetar notações de programação de modo a maximizar o número de erros que não podem ser cometidos ou, caso o sejam, possam ser confiavelmente detectados em tempo de compilação."

C. A. R. Hoare

VISÃO GERAL DO CAPÍTULO

A preocupação com a melhoria na confiabilidade de programas e na detecção inicial de erros forçou os projetistas a melhorar a capacidade de verificação de erros de linguagens. Um bom sistema de tipos permite que os tipos de valores que ocorrem em cada expressão de um programa sejam determinados antes que esta seja executada (em tempo de compilação ou dinamicamente, durante a interpretação do programa), reduzindo assim enormemente a chance da ocorrência de erros relacionados a tipos. Este capítulo explora os elementos de sistemas de tipos e ilustra como um tipo pode ser projetado para a pequena linguagem Clite.

Definição: A detecção de erros de tipos, em tempo de compilação ou de execução, é denominada *verificação de tipos*.

O *sistema de tipos* de uma linguagem de programação fornece um meio para se definir novos tipos e determinar quando um tipo é usado apropriadamente. Um sistema de tipos pode ser definido formal ou informalmente. Neste capítulo, desenvolvemos um sistema de tipos informal para Clite e, a seguir, mostramos como um verificador estático de tipos pode ser implementado com o uso da sintaxe abstrata de Clite introduzida no Capítulo 2.

Também discutimos um transformador de árvore de sintaxe abstrata para Clite, que converte operadores sobrecarregados como + em operadores específicos de tipos. Isso insere conversão de tipos explícita onde for necessário para assegurar que ambos os operandos estejam em conformidade de tipos com relação àqueles requeridos pelo operador.

Lembre-se do que falamos no Capítulo 2, que Clite possui quatro tipos básicos: `int`, `float`, `bool` e `char`. Possui referências e declarações de matrizes, mas não tem construtores de tipos, `struct`, `union` e tipo ponteiro. Com essa simplificação, podemos definir um sistema de tipos rigoroso para Clite que mostra como sua sintaxe abstrata permite uma implementação direta do mesmo.

Clite é uma linguagem simples. Um programa de Clite possui uma única função chamada `main`, cujo corpo possui um único escopo; não há escopos aninhados e nem declarações globais. Os limites das variáveis em Clite são determinados estaticamente, incluindo referências e declarações de matrizes,[1] de modo que todos os tipos de erros podem ser detectados em tempo de compilação. Além disso, todas as conversões implícitas de tipos também são detectadas em tempo de compilação.

A fase semântica do compilador (veja a Figura 2.8) também deve verificar que:

1. Cada *Identificador* de variável declarado deve ser único, e não uma palavra-chave.
2. Cada variável referenciada dentro de qualquer expressão do programa deve ter sido declarada.

Formalmente, nenhuma dessas verificações faz parte do sistema de tipos. Entretanto, as incluiremos aqui como se fizessem.

A propósito, poderíamos perguntar por que essas duas restrições adicionadas não estão de alguma forma definidas na sintaxe EBNF de Clite. Infelizmente, a natureza limitada da EBNF impede que tais restrições sensíveis ao contexto sejam expressas, ou seja, EBNF não é suficientemente poderosa para expressar idéias sintáticas que dependem do contexto no qual uma categoria sintática (como *Identificador*) ocorre.[2] Em vez disso, devemos examinar a sintaxe abstrata de um programa e perguntar que tipo de algoritmo pode ser projetado para assegurar que estas restrições não sejam violadas.

Uma forma de implementar o sistema de tipos de uma linguagem é escrever um conjunto de funções boleanas que definam matematicamente regras como as anteriores. Essa abordagem caracteriza a modelagem formal de um sistema de tipos, e tal modelo para Clite é apresentado na Seção 6.3.

Outra forma de implementar o sistema de tipos de uma linguagem é declarar as regras em uma linguagem altamente estilizada, junto a um algoritmo que implemente essas declarações. Para ilustrar essa abordagem, analise o programa simples em Clite na Figura 6.1 que calcula o fatorial de 8.

Neste programa, as variáveis chamadas `n`, `i` e `result` declaradas no programa (linha 3) devem ser únicas. A variável `n` referenciada nas linhas 4 e 7 deve ser declarada; de forma semelhante, a variável `i` nas linhas 5, 7, 8 e 9 e a variável `result` nas linhas 6 e 9 também

1. Veja no Capítulo 11 outra discussão sobre alocação dinâmica de matrizes.

2. Esta limitação de gramáticas BNF é geralmente exposta em um estudo de linguagens formais (do tipo Chomsky) e seria adequada em um curso sobre teoria da computação ou compiladores.

```
1   // calcular o fatorial do inteiro n
2   void main ( ) {
3       int n, i, result;
4       n = 8;
5       i = 1;
6       result = 1;
7       while (i < n) {
8           i = i + 1;
9           result = result * i;
10      }
11  }
```

| Figura 6.1 Um Pequeno Programa em Clite

devem ser declaradas. Assim, informalmente, podemos ver que o programa em Clite da Figura 6.1 é válido quanto a essas duas regras acrescentadas.[3]

6.1 O SISTEMA DE TIPOS DE CLITE

Como modelamos e implementamos verificações de validade para programas em Clite? Tal modelo requer um conjunto de funções de validação que represente regras como "todas as variáveis declaradas possuem nomes únicos". Cada função de validação *V* pode retornar um valor boleano indicando se essa parte da árvore de sintaxe abstrata do programa é válida quanto ao tipo e é chamada sempre que sua respectiva regra precise ser verificada, ou seja:

$$V : \textit{ClassedeSintaxeAbstrata} \rightarrow \mathbf{B}$$

Um sistema de verificação de tipos também se baseia em um *mapa de tipos*, que é uma função que liga cada nome de variável declarada com seu tipo. O mapa de tipos pode ser extraído das *Declarações* abstratas para o programa em Clite da seguinte maneira:

```
public static TypeMap typing (Declarations d) {
    TypeMap map = new TypeMap();
    for (Declaration di : d)
        map.put (di.v, di.t);
    return map;
}
```

Lembre-se de que a sintaxe abstrata para uma *Declaração* Clite é:

$$\begin{aligned} \textit{Declaração} &= \textit{DeclVariável} \mid \textit{DeclMatriz} \\ \textit{DeclVariável} &= \textit{Variável}\ \texttt{v}\ ;\ \textit{Tipo}\ \texttt{t} \\ \textit{DeclMatriz} &= \textit{Variável}\ \texttt{v}\ ;\ \textit{Tipo}\ \texttt{t};\ \textit{Inteiro}\ \texttt{tamanho} \end{aligned}$$

3. Novamente observamos que as restrições adicionadas em identificadores não fazem, estritamente, parte de um sistema de tipos.

Observação

Segurança em C/C++ /Java

Clite é mais seguro do que C/C++, já que esta última permite conversões implícitas que reduzem o espaço de armazenamento de uma variável em um comando de atribuição (Kernighan e Ritchie, 1988, p. 208):

> O tipo de um comando de atribuição é o tipo de seu operando à esquerda... Na atribuição simples com =, o valor da expressão substitui o valor do objeto referenciado pelo valor-l... Ambos os operandos têm tipo aritmético, em cujo caso o operando à direita é convertido para o tipo do operando à esquerda pela atribuição.

Não há restrições sobre essa atribuição, mesmo se houver conversão limitante. Essa fraqueza no sistema de tipos permite que informações sejam perdidas sem aviso.

A conversão de atribuição em Java para operandos numéricos ou `char` segue um conjunto de regras um pouco mais restritivo.

- Se a expressão à direita for do tipo byte, short, char, int, long ou float, seu valor pode ser ampliado para um valor `double` equivalente.
- Se for byte, short, char, int ou long, pode ser ampliado para um valor `float`.
- Se for byte, short, char ou int, pode ser ampliado para um valor `long`.
- Se for byte, short ou char, pode ser ampliado para um valor `int`.
- Se for byte ou short, pode ser ampliado para um valor `char`.
- Se for byte, pode ser ampliado para um valor `short`.

Uma conversão com redução pode ser usada em uma atribuição sob condições muito limitadas; por exemplo, se a expressão à direita for uma expressão constante e o valor da expressão for representável no tipo da variável à esquerda.

Tais problemas necessitaram do desenvolvimento da aplicação `lint` que "examina potenciais problemas, problemas de portabilidade e construções dúbias em programas em C" (Kernighan e Pike, 1984).

Assim, dada uma lista `d` de *Declarações* `d[i]`, o método `typing` retorna um `TypeMap` Java cujas chaves são as variáveis declaradas `d[i].v` e cujos valores são seus respectivos tipos `d[i].t`.[4], [5] Por exemplo, o programa da Figura 6.1 possui o seguinte `TypeMap`:

```
{ <n, int>, <i, int>, <result, int> }
```

Ainda nesta seção, definiremos cada uma das regras restantes no sistema de tipos de Clite (mais as duas restrições a identificadores acrescentadas), junto a uma discussão sobre sua implementação.

Regra de Tipo 6.1 *Todas as variáveis referenciadas devem ser declaradas.*

4. A lista `d` pode ser implementada como uma `ArrayList` Java, por exemplo, e um `TypeMap` pode ser implementado como uma extensão de um `HashMap` Java.

5. O `size` de uma *DeclMatriz* é ignorado aqui, já que pode ser um valor determinado dinamicamente.

A imposição dessa regra é compartilhada entre as funções de verificação de tipo para diferentes tipos de declarações e expressões Clite, como veremos a seguir:

Regra de Tipo 6.2 *Todas as variáveis declaradas devem ter nomes únicos.*

Esta regra pode ser imposta se verificarmos se cada par distinto de variáveis em uma lista de declarações tem identificadores diferentes entre si. Em especial, ela aborda apenas os requisitos de unicidade mútua para variáveis. De modo geral, essa função também deve especificar que o tipo de cada variável pertence ao conjunto de tipos disponíveis na linguagem (como {`int`, `float`, `char bool`} para Clite). Omitimos este requisito aqui porque ele já foi imposto pela sintaxe de Clite.

Supondo que a categoria de sintaxe abstrata `Declarações` seja implementada como uma `ArrayList` em Java, o método `V` para `Declarações` pode ser especificado da seguinte maneira:[6]

```
public static void V (Declarations d) {
     for (int i = 0; i<d.size() - 1; i++)
          for (int j=i+1; j<d.size(); j++) {
               Declaration di = d.get(i);
               Declaration dj = d.get(j);
               check( ! (di.v.equals(dj.v)),
                       "duplicate declaration: " + dj.v);
          }
}
```

Por exemplo, as declarações do programa na Figura 6.1 são nomeadas como `n`, `i` e `result`, que são nomes únicos entre si. Assim, o método *V* para este conjunto específico de declarações deve retornar normalmente (a condição `check` é `verdadeira` para todos os pares). A implementação da função `V` em Java retorna `void`, em vez de parar a execução na primeira instância de um par duplicado. Essa prática se baseia na avaliação de curto circuito do operador *e*, na qual:

a e b ≡ se não a então falso senão b

Isso significa que, assim que uma verificação de validade falhar, o programa que estiver sendo verificado continua a ser inválido apesar da verdade ou falsidade de verificações posteriores de validade.

Finalmente, observamos que esse método não espelha exatamente a definição dada para a unicidade entre nomes de variáveis declaradas, ou seja, a regra sugere que *todos* os pares distintos devem ser verificados (isto é, `d[i]` *versus* `d[j]` *e* `d[j]` *versus* `d[i]`). Omitimos essa redundância computacional executando uma diagonalização sobre o código do laço.

Para completar o sistema de tipos de Clite, um conjunto de funções *V* é definido para cada classe sintática abstrata, incluindo a classe mais geral *Programa*. Um programa completo,

6. Na nossa implementação da função *V*, o resultado `boleano` é substituído por uma chamada a uma função *check*, que gera uma exceção. Assim, se o verificador de tipos terminar de forma normal, o compilador pode estar seguro de que a árvore de sintaxe abstrata é válida no que diz respeito às regras de tipos.

do ponto de vista da sintaxe abstrata, possui duas partes, uma série de *Declarações* e um *Bloco*:

```
class Program {
     // Programa = Declarações partedec ; Bloco corpo
     Declarações partedec;
     Bloco corpo;
}
```

Regra de Tipo 6.3 *Um Programa é válido em termos de tipos se suas Declarações* `partedec` *forem válidas e seu Bloco* `corpo` *for válido considerando-se o mapa de tipos para essas Declarações específicas.*

Já que essas duas partes são implementadas como classes Java, esta especificação é fácil de ser implementada.

```
public static void V (Program p) {
   V (p.decpart);
   V (p.body, typing (p.decpart));
}
```

Para o programa de exemplo da Figura 6.1, isso significa que *Declarações* (`n`, `i` e `result`) são válidas e que o bloco que começa com a *Atribuição* `n = 8`; é válido. Observe que a validade de tipos do `corpo` do programa pode ser verificada apenas em relação ao mapa de tipos específico que representa sua `partedec`.

Para garantir a validade de tipos de um *Bloco* devemos, é claro, assegurar a validade de tipos de cada um de seus comandos individuais. As regras que definem a validade de um *Comando* são expressas na Regra de Tipo 6.4.

Regra de Tipo 6.4 *Um Comando é válido em relação ao mapa de tipos do programa se satisfizer às seguintes restrições:*

1 *Um Salto é sempre válido.*

2 *Uma Atribuição é válida, se todas as seguintes forem verdadeiras:*

(a) *Sua Variável* `alvo` *está declarada.*

(b) *Sua Expressão* `origem` *é válida.*

(c) *Se o tipo da sua Variável* `alvo` *for* `float`*, então o tipo de sua Expressão* `origem` *deve ser* `float` *ou* `int`.[7]

(d) *Caso contrário, se o tipo de sua Variável* `alvo` *for* `int`*, então o tipo da sua Expressão* `origem` *deve ser* `int` *ou* `char`.

(e) *Caso contrário, o tipo da sua Variável* `alvo` *deve ser o mesmo da sua Expressão* `origem`.

3 *Um comando Condicional é válido se sua Expressão* `teste` *for válida e tiver tipo* `bool` *e tanto seus Blocos de Comandos* `ramificaçãoEntão` *quanto os de* `ramificaçãoSenão` *forem válidos.*

4 *Um Laço é válido se sua Expressão* `teste` *for válida e tiver tipo* `bool` *e seu Comando* `corpo` *for válido.*

5 *Um Bloco é válido se todos os seus Comandos forem válidos.*

7. Estamos excluindo *RefMatriz* da definição, já que são deixadas como um exercício.

Referindo-se novamente ao nosso programa exemplo da Figura 6.1, observe que seu *Bloco* abstrato contém quatro *Comandos*: três *Atribuições* e um *Laço.* Assim, de acordo com a Regra de Tipo 6.4, a validade desse programa depende da validade de cada um desses comandos individuais e, recursivamente, da validade do outro *Bloco* que seja o `corpo` do próprio *Laço*. Além disso, para cada *Atribuição* como `n = 8;`, a Regra de Tipo 6.4 requer que:

- A *Variável* `alvo n` seja declarada;
- A *Expressão* `origem` 8 seja válida; e
- O tipo de 8 seja `int` ou `char`.

Para completar essa atividade de verificação de tipos, devemos então ter uma definição clara do que queremos dizer por uma *Expressão* ser válida.

Implementar as regras anteriores com base na sintaxe abstrata de Clite é direto, conforme sugerido no esboço da Figura 6.2. Lá, o primeiro comando `if` espelha a Regra de Tipo 6.4.1 sobre a validade de comandos *Salto* e a lógica do comando *Atribuição* segue a Regra de Tipo 6.4.2.

O leitor observador notará o uso de diversas técnicas de programação defensiva na Figura 6.2. O primeiro comando verifica se *Statement* não é o objeto Java `null`. Outra verificação no final do método garante que nenhum caso foi deixado de lado e, portanto, o controle não deve *chegar aqui.*[8]

A seguir, especificamos a regra de validade para uma *Expressão*:

> **Regra de Tipo 6.5** *A validade de uma Expressão é definida com o uso do mapa de tipos do programa e cada uma das subclasses da Expressão:*
>
> **1** *Um Valor é válido.*
>
> **2** *Uma Variável é válida se seu* `id` *aparecer no mapa de tipos.*[9]
>
> **3** *Um Binário é válido se todas as afirmações seguintes forem verdadeiras:*
>
> (a) *Suas Expressões* `termo1` *e* `termo2` *são válidas.*
>
> (b) *Se seu OpBinário* `op` *for aritmético (+, -, *, /), então ambas as suas Expressões devem ser* `int` *ou* `float`.
>
> (c) *Se* `op` *for relacional* (==, !=, <, <=, >, >=)*, então ambas as suas Expressões devem ter o mesmo tipo.*
>
> (d) *Se* `op` *for boleano* (&&, ||), *então ambas as suas Expressões devem ser* `bool`.
>
> **4** *Um Unário é válido se todas as afirmações seguintes forem verdadeiras:*
>
> (a) *Sua Expressão* `termo` *for válida.*
>
> (b) *Se o seu OpUnário* `op` *for* `!`, *então* `termo` *deve ser* `bool`.
>
> (c) *Se* `op` *for –, então* `termo` *deve ser* `int` *ou* `float`.
>
> (d) *Se* `op` *for a conversão de tipo* `float( )` *ou* `char( )`, *então* `termo` *deve ser* `int`.
>
> (e) *Se* `op` *for a conversão de tipo* `int( )`, *então* `termo` *deve ser* `float` *ou* `char`.

A implementação dessa função *V* é esboçada na Figura 6.3. O trecho dado espelha as regras 1–3 anteriores, com a implementação de expressões *Unário* omitidas.

Finalmente, o tipo do resultado de uma *Expressão* deve ser definido para todas as variações.

---

8. Um programador orientado a objeto deve observar que os comandos `if` poderiam ter sido evitados se tivéssemos usado uma abordagem mais orientada a objeto. Entretanto, neste ponto, estamos tentando implementar precisamente o modelo formal da Seção 6.3, que, sendo matemático, é de natureza funcional.

9. Novamente, estamos excluindo as *RefMatrizes* desta definição; elas são deixadas como um exercício.

```
public static void V (Statement s, TypeMap tm) {
    if ( s == null )
        throw new IllegalArgumentException(
            "Erro AST: declaração nula");
    if (s instanceof Skip) return;
    if (s instanceof Assignment) {
        Assignment a = (Assignment)s;
        check( tm.containsKey(a.target)
                , " alvo não definido na atribuição: " +
                a.target);
        V(a.source, tm);
        Type ttype = (Type)tm.get(a.target);
        Type srctype = typeOf(a.source, tm);
        if (ttype != srctype) {
            if (ttype == Type.FLOAT)
                check( srctype == Type.INT
                        , "modo de atribuição misturado " +
                        a.target);
            else if (ttype == Type.INT)
                check( srctype == Type.CHAR
                        , "modo de atribuição misturado " +
                        a.target);
            else
                check( false
                        , "modo de atribuição misturado " +
                        a.target);
        }
        return;
    }
    ...

    throw new IllegalArgumentException(
            "nunca deve chegar aqui");
}
```

| Figura 6.2 Função de Validação *V* para *Comando*

Regra de Tipo 6.6 *O tipo de resultado de toda Expressão é determinado da seguinte maneira:*

1. *Se a Expressão for um Valor, então o tipo do seu resultado é o tipo desse Valor.*
2. *Se a Expressão for uma Variável, então o tipo do seu resultado é o tipo dessa Variável.*
3. *Se a Expressão for um Binário, então:*
   (a) *Se o Operador for aritmético* (+, −, *, /) *então o tipo do seu resultado é o tipo dos seus operandos. Por exemplo, a Expressão* `x+1` *requer que* `x` *seja* `int` *(já que* `1` *é* `int`*), de modo que o tipo do seu resultado é* `int`.

```
public static void V (Expression e, TypeMap tm) {
    if (e instanceof Value)
        return;
    if (e instanceof Variable) {
        Variable v = (Variable)e;
        check( tm.containsKey(v)
                , "variável não declarada: " + v);
        return;
    }
    if (e instanceof Binary) {
        Binary b = (Binary) e;
        Type typ1 = typeOf(b.term1, tm);
        Type typ2 = typeOf(b.term2, tm);
        V (b.term1, tm);
        V (b.term2, tm);
        if (b.op.ArithmeticOp( ))
            check( typ1 == typ2 &&
                (typ1 == Type.INT || typ1 == Type.FLOAT)
                , "erro de tipo para " + b.op);
        else if (b.op.RelationalOp( ))
            check( typ1 == typ2 , "erro de tipo para " + b.op);
        else if (b.op.BooleanOp( ))
            check( typ1 == Type.BOOL && typ2 == Type.BOOL,
                b.op + ": operando não boleano");
        else
            throw new IllegalArgumentException(
                "nunca deve chegar aqui");
        return;
    }
    ...
    throw new IllegalArgumentException("nunca deve chegar aqui");
}
```

| Figura 6.3 Função de Validação *V* para Expressão

(b) *Se o Operador for relacional* (`<`, `<=`, `>`, `>=`, `==`, `!=`) *ou boleano* (`&&`, `||`)*, então o tipo do seu resultado é* `bool`.

4 *Se a Expressão for um Unário, então:*

(a) *Se o operador for* `!`*, então o tipo do seu resultado é* `bool`.

(b) *Se o operador for* `-`*, então o tipo do seu resultado é o tipo do seu operando.*

(c) *Se o operador for uma conversão de tipo, então o tipo do seu resultado é dado pela conversão.*

Como exemplo, os leitores devem ver que a *Expressão* `result*i` que é a quarta *Atribuição* do programa de exemplo em Clite da Figura 6.1 é válida, ou seja, pela Regra 6.5, `result*i` é um *Binário*, seu *Operador* é aritmético e ambos os seus operandos são do tipo `int` (usando a Regra 6.6).

```
public static Type typeOf (Expression e, TypeMap tm) {
    if (e instanceof Value) return ((Value)e).type;
    if (e instanceof Variable) {
        Variable v = (Variable)e;
        check (tm.containsKey(v), "variável indefinida: " + v);
        return (Type) tm.get(v);
    }
    if (e instanceof Binary) {
        Binary b = (Binary)e;
        if (b.op.ArithmeticOp( ))
            if (typeOf(b.term1,tm) == Type.FLOAT)
                return (Type.FLOAT);
            else return (Type.INT);
        if (b.op.RelationalOp( ) || b.op.BoleanOp( ))
            return (Type.BOOL);
    }
    if (e instanceof Unary) {
        Unary u = (Unary)e;
        if (u.op.NotOp( )) return (Type.BOOL);
        else if (u.op.NegateOp( )) return typeOf(u.term,tm);
        else if (u.op.intOp( )) return (Type.INT);
        else if (u.op.floatOp( )) return (Type.FLOAT);
        else if (u.op.charOp( )) return (Type.CHAR);
    }
    throw new IllegalArgumentException(
        "nunca deve chegar aqui");
    }
```

| Figura 6.4 A Função `typeof`

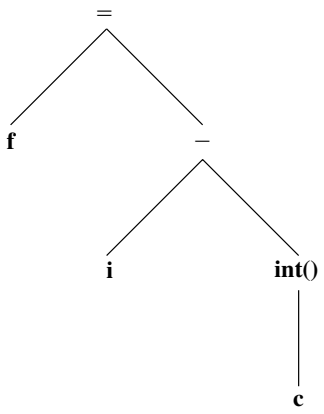
Uma implementação parcial da função `typeof`, que implementa esta regra, é dada na Figura 6.4; novamente o código para o tipo de uma expressão *Unária* está omitido. O código tenta espelhar fielmente a Regra de Tipo 6.6.

6.2 CONVERSÃO IMPLÍCITA DE TIPOS

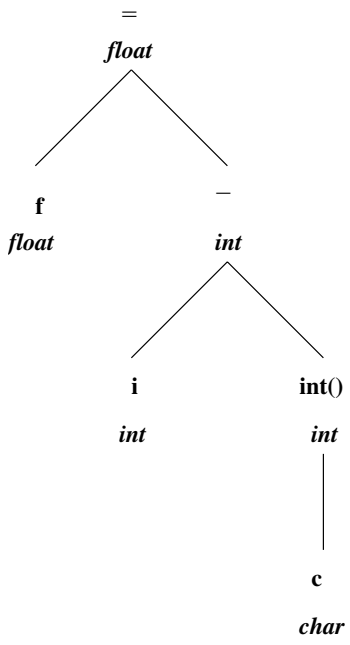
Conforme sugerido na Seção 6.1, Clite suporta algumas conversões implícitas de tipos em uma *Atribuição*, sendo que todas são conversões ampliadoras. Todas as outras conversões requerem que o programa inclua uma função de conversão de tipo explícita, que é uma expressão *Unária*, já que ambos os operandos na implementação de uma expressão *Binária* devem ter o mesmo tipo.

Na semântica de uma *Atribuição*, o valor da *Expressão* `origem` substitui o da *Variável* `alvo`. Se for necessário, o tipo da `origem` deve ser convertido explicitamente, novamente usando as conversões ampliadoras especificadas na Regra 6.4.2 e governadas pelo tipo do `alvo`. De outra maneira, os tipos do `alvo` e da `origem` devem ser o mesmo.

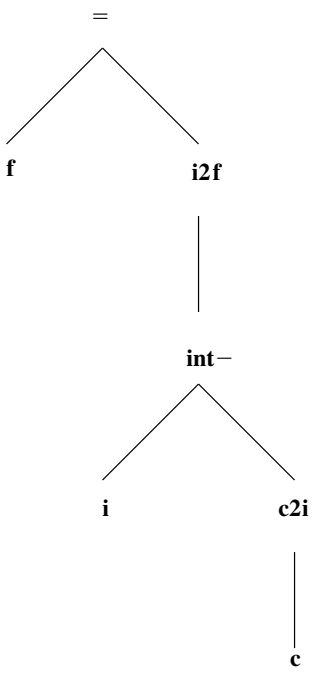
Como exemplo de conversão implícita de tipo, analise a seguinte árvore de sintaxe abstrata:



Nesta árvore, estamos usando tipagem implícita com a primeira letra de cada variável indicando seu tipo; por exemplo, a variável `f` é do tipo `float`. A árvore é considerada de tipo válido porque anotando-a com as seguintes informações de tipo:



força uma conversão ampliadora implícita na declaração de atribuição, o que é permitido pela Regra de Tipo 6.4.2. Finalmente, transformamos a árvore de modo que todos os operadores sejam tipados explicitamente e as conversões implícitas se tornem explícitas:



Aqui, `c2i` denota a conversão de caractere para inteiro, `int-` denota a subtração de inteiro e `i2f`, a conversão de inteiro para ponto flutuante. Em um sistema real, o operador de atribuição = seria substituído por uma atribuição específica de tipo, como `float=`.

Assim, a árvore de sintaxe abstrata transformada substitui todos os operadores sobrecarregados por operadores específicos de tipo e insere conversões de tipo explícitas no lugar das implícitas onde quer que elas sejam necessárias.

As transformações são implementadas como um conjunto de funções *T* (para *transformação*) que convertem uma árvore de sintaxe abstrata e um mapa de tipos em outra árvore de sintaxe abstrata com todas as conversões de tipo implícitas removidas:

$$T : \textit{ClassedeSintaxeAbstrata} \times \textit{MapadeTipos} \rightarrow \textit{ClassedeSintaxeAbstrata}$$

A função de transformação *T* para *Programa* constrói um novo *Programa* que consiste das *Declarações* originais[10] e um corpo transformado (um *Bloco*):

```
public static Program T (Program p, Typemap tm) {
    Block body = (Block)T(p.body, tm);
    return new Program(p.decpart, body);
}
```

As transformações-chave são as transformações para os comandos *Atribuição* e *Expressão*. A primeira deve inserir conversões explícitas de `int` para `float` e de `char` para `int`, conforme

10. Nenhuma dessas transformações é necessária para a categoria *Declarações*, já que ela não contém operadores.

```
if (s instanceof Assignment) {
  Assignment a = (Assignment)s;
     Variable target = a.target;
     Expression src = T (a.source, tm);
     Type ttype = (Type)tm.get(a.target);
     Type srctype = StaticTypeCheck.typeOf(a.source, tm);
     if (ttype == Type.FLOAT) {
     if (srctype == Type.INT) {
          src = new Unary(new Operator(Operator.I2F), src);
          srctype = Type.FLOAT;
          }
     }
     else if (ttype == Type.INT) {
          if (srctype == Type.CHAR) {
               src = new Unary(new Operator(Operator,C2I), src);
               sctype = Type.INT;
          }
     }
     StaticTypeCheck.check( ttype == srctype,
                  "erro de tipo na atribuição a " + target);
     return new Assignment(target, src);
}
```

| Figura 6.5 Transformação de Atribuição

mostrado nas Figuras 6.5 e 6.6. Caso contrário, as regras de verificação de tipo asseguram que os tipos sejam os mesmos e, assim, nenhuma transformação adicional é necessária. Observe também na Figura 6.6 que não transformamos o operador de atribuição por tipo porque não é necessário para o modelo semântico de tempo de execução. Para a maioria das máquinas, o operador de atribuição precisaria ser distinguido por tipo (pelo menos pelo tamanho da memória necessária para o tipo), conforme discutido na Seção 5.3.

A maior parte do trabalho dos operadores de transformação por tipo é realizada pela função *T* para uma *Expressão,* o que é esboçado na Figura 6.6. Para operadores binários, um mapa específico de tipos é usado para traduzir o operador. Por exemplo, em `i - int(c)`, o tipo do operando é *int*, de modo que o operador - é traduzido para `int -` usando o mapa de operadores inteiros `intMap( )`.

Para expressões *Unária* (não mostradas), o operador é o principal regulador na determinação da transformação do operador. Entretanto, para o - unário e a conversão de tipo `int( )`, são usados mapas de tipos dependentes de tipo, ou seja, o mapa de tipo inteiro `intMap( )` é usado para transformar uma conversão `int( )` sobre um caractere em um operador `c2i`.

A árvore de sintaxe abstrata transformada serve como base para a especificação e a implementação de uma função semântica de tempo de execução no Capítulo 8. Conforme descrito aqui, essa função assume que a árvore de sintaxe abstrata traduzida é segura no que diz respeito a tipos.

6.3 FORMALIZANDO O SISTEMA DE TIPOS DE CLITE

A seguir estão as regras formais para verificação estática de tipos em Clite; elas são escritas naturalmente como predicados. A discussão e as implementações dessas regras como

```
public static Expression T (Expression e, TypeMap tm) {
    if (e instanceof Value)
        return e;
    if (e instanceof Variable)
        return e;
    if (e instanceof Binary) {
        Binary b = (Binary)e;
        Type typ1 = StaticTypeCheck.tyeof(b.term1, tm);
        Type typ2 = StaticTypeCheck.tyeof(b.term2, tm);
        Expression t1 = T (b.term1, tm);
        Expression t2 = T (b.term2, tm);
        if (typ1 == Type.INT)
            return new Binary(b.op.intMap(b.op.val), t1,t2);
        else if (typ1 == Type.FLOAT)
            return new Binary(b.op.floatMap(b.op.val), t1,t2);
        else if (typ1 == Type.CHAR)
            return new Binary(b.op.charMap(b.op.val), t1,t2);
        else if (typ1 == Type.BOOL)
            return new Binary(b.op.boolMap(b.op.val), t1,t2);
        throw new IllegalArgumentException(
                "nunca deve chegar aqui");
    }
    ...
    throw new IllegalArgumentException(
                "nunca deve chegar aqui");
}
```

| Figura 6.6 Transformação de Expressão

uma coleção de métodos Java já foram mostradas na Seção 6.1. Os leitores que não estiverem interessados na matemática relacionada à especificação do sistema de tipos podem pular esta seção.[11]

O *mapa de tipos* de um *Programa* é um conjunto de pares ordenados, com cada par tendo uma *Variável* v e seu *Tipo t* declarado.

$$mt = \{\langle v_1, t_1\rangle, \langle v_2, t_2\rangle, \ldots, \langle v_n, t_n\rangle\}$$

A função *tipagem* cria um *MapadeTipo* a partir de uma série de *Declarações* e corresponde à Regra de Tipo 6.1. Já que identificadores individuais são únicos, a união do conjunto assegura que todas as variáveis e seus tipos declarados estejam representados.

$$tipagem: Declarações \rightarrow MapadeTipo$$

$$tipagem(Declarações\ d) = \bigcup_{i \in \{1,\ldots,n\}} \langle d_i.v, d_i.t\rangle$$

11. A validação de tipos de *DeclMatrizes* e *RefMatrizes* é ignorada aqui, já que é ignorada na Seção 6.2.

Uma série de *Declarações* é válida se seus nomes de variáveis forem mutuamente exclusivos (Regra de Tipo 6.2). Observe que a definição funcional verifica todos os pares de identificadores.

V : *Declarações* $\rightarrow$ **B**

V(Declarações d*)* $= \forall i, j \in \{1, \ldots, n\}\ (i \neq j \Rightarrow d_i.v \neq d_j.v)$

Um programa é válido se suas *Declarações* forem válidas (*V*(`p.partedec`)) e seu *Bloco* for válido (*V*(`p.corpo, tipagem (p.partedec)`) para o *MapadeTipo* definido para aquelas *Declarações* (Regra de Tipo 6.3).

V : *Programa* $\rightarrow$ **B**

V(Programa p*)* $= V$(p.partedec) $\wedge$ V(p.corpo, *tipagem*(p.partedec))

A verificação de validade para um *Comando* depende do tipo deste comando; por exemplo, um comando *Salte* é sempre válido, enquanto um comando *Atribuição* requer a verificação tanto da variável alvo quanto da expressão fonte. A função a seguir é equivalente à Regra de Tipo 6.4.

V : *Asserção* $\times$ *MapadeTipos* $\rightarrow$ **B**

V(Asserção s, MapadeTipos mt*)*

$= verdadeiro$ se s for um *Salto*

$= s$.alvo $\in$ mt $\wedge$ $V(s$.fonte, mt$)$
$\wedge((mt(s$.alvo$)$ = float $\wedge$ *tipoDe*$(s$.fonte, *mt*$) \in \{$float, int$\})$
$\vee(mt(s$.alvo$)$ = int $\wedge$ *tipoDe*$(s$.fonte, *mt*$) \in \{$int, char$\})$
$\vee(mt(s$.alvo$)$ = *tipoDe*$(s$.fonte, *mt*$)$
se s for uma *Atribuição*

$= V(s$.teste, mt$) \wedge$ *tipoDe*$(s$.teste, mt$)$ = boleano
$\wedge V(s$.ramificaçãoEntão, mt$) \wedge V(s$.ramificaçãoSenão, mt$)$
se s for uma *Condicional*

$= V(s$.teste, mt$) \wedge$ *tipoDe*$(s$.teste, mt$)$ = boleano
$\wedge V(s$.corpo, mt$)$
se s for um *Laço*

$= V(b_1$, mt$) \wedge V(b_2$, mt$) \wedge \ldots \wedge (b_n$, mt$)$
se s for um *Bloco* $= b_1 b_2 \ldots b_n$
$\wedge\ n \geq 0$

Uma *Expressão* é válida se for um *Valor,* uma *Variável* no mapa de tipos do programa ou uma expressão *Binária* ou *Unária* cujos operandos satisfaçam a restrições adicionais. A função abaixo é equivalente à Regra de Tipo 6.5.

V : *Expressão* $\times$ *MapadeTipo* $\rightarrow$ **B**

V(Expressão e, MapadeTipo mt*)*

$= verdadeiro$ se e for um *Valor*

$= e \in$ mt se e for uma *Variável*

$= V\,(e$.termo1, *mt*$) \wedge V(e$.termo2, *mt*$)$
$\wedge$ *tipoDe*$(e$.termo1, mt$) \in \{$float, int$\} \wedge$ *tipoDe*$(e$.termo2, mt$) \in \{$float, int$\}$

$\wedge$ *tipoDe*(*e*.termo1, mt) = *tipoDe*(*e*.termo2, mt)
se *e* for um *Binário* $\wedge$ *e.op* $\in$ {*OpAritmético*}
= *V*(*e*.termo1, *mt*) $\wedge$ *V*(*e*.termo2, mt)
$\wedge$ *tipoDe*(*e*.termo1, mt) = *tipoDe*(*e*.termo2, mt)
se *e* for um Binário $\wedge$ *e.op* $\in$ {*OpRelacional*}
= *V*(*e*.termo*1, mt*) $\wedge$ *V*(*e*.termo2, mt)
$\wedge$ *tipoDe*(*e*.termo1, mt) = boleano $\wedge$ *tipoDe*(*e*.termo2, mt) = boleano
se *e* for um Binário $\wedge$ *e.op* $\in$ {*OpBoleano*}
= *V*(*e*.termo, *mt*)
$\wedge$ ((*e*.op = ! $\wedge$ *tipoDe*(*e*.termo, *mt*) = boleano)
$\vee$ (*e*.op = $-$ $\wedge$ *tipoDe*(*e*.termo, *mt*) = $\in$ {float, int})
$\vee$ (*e*.op = float() $\wedge$ *tipoDe*(*e*.termo, *mt*) = $\in$ {int})
$\vee$ (*e*.op = char() $\wedge$ *tipoDe*(*e*.termo, *mt*) = $\in$ {int})
$\vee$ (*e*.op = int() $\wedge$ *tipoDe*(*e*.termo, *mt*) = $\in$ {float, char}))
se *e* for um *Unário*

A função auxiliar *tipoDe* define o tipo de uma *Expressão*, conforme determinado pela Regra 6.6. Aqui, a expressão *e*.tipo denota o tipo de um valor ou de uma variável *e* no mapa de tipos.

tipoDe: *Expressão* $\times$ *MapadeTipos* $\rightarrow$ *Tipo*
tipoDe(*expressão e, MapadeTipos* mt)

= *e*.tipo	se *e* for um *Valor*
= *e*.tipo	se *e* for uma *Variável* $\wedge$ *e* $\in$ mt
= *tipoDe*(*e*.termo1, *mt*)	se *e* for um *Binário* $\wedge$ *e.op* $\in$ {*OpAritmético*}
= boleano	se *e* for um *Binário* $\wedge$ *e.op* $\in$ {*OpBoleano*, *OpRelacional*
= boleano	se *e* for um *Unário* $\wedge$ *e.op* = !
= *tipoDe*(*e*.termo, *mt*)	se *e* for um *Unário* $\wedge$ *e.op* = $-$
= float	se *e* for um *Unário* $\wedge$ *e.op* = float()
= char	se *e* for um *Unário* $\wedge$ *e.op* = char()
= int	se *e* for um *Unário* $\wedge$ *e.op* = int()

Isso completa a definição formal do sistema de tipos de Clite. Tanto as regras formais em português quanto a implementação em Java mostrada na Seção 6.1 se baseiam nessa definição. Ela ilustra como o sistema de tipos de uma linguagem pode ser definido de forma rigorosa com o uso de ferramentas matemáticas convencionais. Tal rigor assegura que o entendimento informal e a implementação do sistema de tipos da linguagem sejam válidas e consistentes.

6.4 RESUMO

Com um sistema forte de tipos, muitos erros em tempo de execução podem ser evitados em tempo de compilação, ou pelo menos mais claramente compreendidos em tempo de execução. A pequena linguagem Clite fornece um bom meio para o desenvolvimento e a implementação de um sistema de tipos completo para uma linguagem. O tratamento formal de sistemas de tipos, usando predicados e funções matemáticas, também é ilustrado.

EXERCÍCIOS

6.1 Expanda a Regra de Tipo 6.2 para *Declarações* de modo que defina o requisito de que o tipo de cada variável pertença a um pequeno conjunto de tipos disponíveis {float, int, bool, char}. Use o mesmo estilo de regras e sintaxe abstrata para *Declarações* usado neste capítulo.

6.2 Expanda o método Java que implementa a função *V* para *Declarações* de modo que ele implemente o requisito adicional declarado no Exercício 6.1.

6.3 Argumente que o método Java que implemente a função *V* para *Declarações* está correto, no sentido de que ele cobre todos os casos que a própria Regra de Tipo cobre.

6.4 Para cada um dos seguintes comandos usando as regras de tipos implícitas da Seção 6.1, primeiro desenhe a árvore sintática abstrata não tipada e a seguir adicione informações sobre tipos, conforme feito na Seção 6.2. Se alguma árvore de sintaxe abstrata tipada for inválida quanto ao tipo, indique claramente onde e por quê.

(a) f = −3;
(b) f = −3,5;
(c) i = −2;
(d) i = −2,5;
(e) f = c;
(f) i = c;
(g) i = i + 1;
(h) i = i + c;
(i) f = f + i;
(j) if (b) i = 1;
(k) if (f) i = 1;
(l) if (f1 > f2) f3 = f4 + f5;

6.5 Adicione informações sobre o número da linha e coluna àquelas exibidas pelos `checks` nas implementações da função *V*. Dica: para fazer isso, você precisará modificar as classes `Lexer`, `Token` e `AbstractSintax`.

6.6 Modifique a Regra de Tipo 6.5 para permitir conversões implícitas de `char` para `int` e de `int` para `float`. Modifique a função `V` para expressões de modo a implementar as regras de tipo.

6.7 Adicione um comando `put()` a Clite, no qual `put()` recebe uma expressão como argumento e grava o valor desta expressão em `stdout`. Para este novo comando:
(a) Defina seu EBNF e sua sintaxe abstrata;
(b) Modifique a Regra de Tipo 6.4 para definir as verificações de validade necessárias; e
(c) Modifique a função `V` para comandos de modo a implementar as regras de tipo.

6.8 Adicione as funções `getInt()` e `getFloat()` a Clite, onde `getInt/GetFloat` recebe o próximo valor de `stdin` e retorna o valor lido como um `int/float`, respectivamente. Para esta nova expressão:
(a) Defina seu EBNF e sua sintaxe abstrata.
(b) Modifique a Regra de Tipo 6.5 para definir as verificações de tipo necessárias.
(c) Modifique a função *V* para expressões de modo a implementar as regras de tipo.

6.9 Modifique a classe transformadora de tipos de modo que as conversões implícitas de tipos do Exercício 6.6 sejam convertidas em explícitas.

6.10 Modifique as funções *V* matemáticas da Seção 6.3 de modo que as conversões implícitas do Exercício 6.6 sejam incluídas.

6.11 Modifique a Regra de Tipo 6.5 de modo que ela verifique a validade de uma *RefMatriz* (veja a Figura 2.14).

6.12 Modifique a Regra de Tipo 6.6 de modo que ela calcule o tipo de uma *RefMatriz* (veja a Figura 2.14).

6.13 Implemente a verificação de tipos para referências de matrizes modificando os métodos da função *V* para *Comando* e *Expressão* e o método *Tipo De*.

7 Semântica

"Ishmael: 'Com certeza isso deve ter algum significado.'"

Herman Melville, Moby Dick

VISÃO GERAL DO CAPÍTULO

Nos primeiros tempos do projeto de linguagens, era possível escrever um programa sintaticamente correto em uma determinada linguagem que se comportaria de forma diferente ao ser executado com a mesma entrada em plataformas diferentes. Essa situação surgiu porque o comportamento em tempo de execução da linguagem (semântica) não era definido com precisão suficiente. Compiladores diferentes traduziam um programa para versões de linguagem de máquina não equivalentes.

Desde aquela época, os projetistas de linguagens perceberam gradualmente que um tratamento rigoroso da semântica é tão importante quanto um tratamento formal da sintaxe. Uma semântica rigorosa deve assegurar que determinado programa "signifique" a mesma coisa, independentemente da plataforma em tempo de execução.

Um meio-termo, ao se lidar com casos em que diferentes implementações possam produzir diferentes resultados, identificaria aqueles casos na definição e na especificação da linguagem cujos significados são deixados para a implementação. Isso ocorre em C, por exemplo, para a ordem na avaliação dos argumentos de funções.

7.1 MOTIVAÇÃO

As principais motivações para se definir com precisão uma linguagem de programação são:

1. Fornecer aos programadores uma definição oficial do significado de todas as construções da linguagem.[1]
2. Fornecer aos escritores de compiladores uma definição oficial do significado de todas as construções, evitando, assim, diferenças nas implementações.
3. Fornecer uma base para a padronização da linguagem.[2]

Uma linguagem de programação é bem definida apenas quando sua semântica, assim como sua sintaxe e seu sistema de tipos, estiver integralmente definida. Lembre-se do Capítulo 2, que diz que a *semântica* de uma linguagem de programação é uma definição precisa do *significado* de qualquer programa que seja correto sintaticamente e também quanto aos seus tipos. Três abordagens diferentes têm sido usadas amplamente para definir a semântica de uma linguagem de programação.

A idéia mais direta do significado de um programa é "o que acontece quando o programa é compilado pelo compilador C e executado na máquina M". No final da década de 1950, Fortran foi definido originalmente dessa forma, usando o compilador Fortran em um computador IBM 709. Uma caracterização precisa dessa idéia é chamada de *semântica operacional.*[3] Uma segunda forma de ver o significado de um programa é axiomatizar o significado de cada comando da linguagem. Assim, dada essa especificação formal do que um programa deve fazer, pode-se provar com rigor que o programa executa o que foi especificado usando um argumento lógico sistemático. Essa abordagem é chamada *semântica axiomática* e é usada no Capítulo 18.

Uma terceira forma de visualizar a semântica é definir o significado de cada tipo de comando que ocorre na sintaxe (abstrata) como uma função matemática de transformação de estados. Assim, o significado de um programa pode ser expresso como um conjunto de funções que atuam sobre o estado do programa. Essa abordagem é chamada *semântica denotacional* e é usada no Capítulo 8.

Cada uma dessas abordagens à semântica tem vantagens e desvantagens. A semântica operacional possui a vantagem de representar o significado de um programa diretamente no código de uma máquina real (ou simulada). Contudo, essa também é uma potencial fraqueza, já que definir a semântica de uma linguagem de programação com base em alguma determinada arquitetura, real ou abstrata, limita a utilidade dessa definição para escritores de compiladores e programadores que trabalham com arquiteturas

1. Pessoas bastante versadas nas nuances de uma linguagem são, muitas vezes, chamadas de *advogados da linguagem*.

2. PL/I foi a primeira linguagem na qual um modelo preciso serviu como base para seu padrão ANSI/ISO.

3. Tecnicamente, há dois tipos de semânticas operacionais, chamadas de "tradicional" e "estruturada" (às vezes, chamadas de "semântica natural").

diferentes. Além disso, a máquina virtual na qual as instruções são executadas também precisa de uma descrição semântica, o que adiciona complexidade e pode levar a definições circulares.

A semântica axiomática é especialmente útil na exploração de propriedades formais de programas. Programadores que precisarem escrever programas que sejam verificavelmente corretos a partir de um conjunto preciso de especificações estão especialmente bem servidos por esse estilo de semântica. A utilidade da semântica axiomática é ilustrada no Capítulo 18.

A semântica denotacional é valiosa porque seu estilo funcional traz a definição da semântica de uma linguagem a um nível alto de precisão matemática. Por meio dela, os projetistas de linguagens obtêm uma definição funcional do significado de cada construção da linguagem que é independente de qualquer arquitetura de máquina. O modelo denotacional também é valioso porque nos permite explorar ativamente conceitos de projeto de linguagens usando uma abordagem experimental.

Este capítulo apresenta um enfoque semântico operacional informal, de modo que pode abordar não apenas a semântica básica das atribuições, das expressões e do fluxo de controle, mas também as questões mais amplas de semântica que envolvem recursos como manipulação de exceções e entrada e saída.

7.2 SEMÂNTICA DE EXPRESSÕES

Em linguagens de programação, a avaliação de expressões é fundamentalmente importante. Nesta seção, exploramos a semântica de expressões, incluindo operadores e sua associatividade e precedência, o papel de diferentes ordens de avaliação e a importância da precisão.

Em matemática, expressões são puras no sentido de que não têm efeitos colaterais. Assim, a ordem em que subexpressões são avaliadas não é importante. Algumas linguagens de programação funcionais espelham essa pureza, já que proíbem os efeitos colaterais. O Capítulo 14 explora essa idéia mais integralmente.

7.2.1 Notação

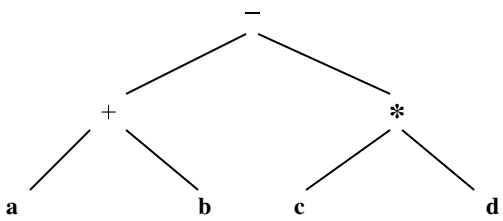
Analise a árvore de expressão da Figura 7.1. Seguindo a notação matemática convencional, a maioria das linguagens de programação usa notação *interfixada* para operações binárias, ou seja, cada operador binário é escrito entre seus operandos. Na notação interfixada, a expressão da Figura 7.1 seria escrita como uma expressão completamente parametrizada como:

```
(a + b) - (c * d)
```

Supondo a associatividade e a precedência normais dos operadores +, - e *, a expressão também poderia ser escrita como:

```
a + b - c * d
```

Figura 7.1 Exemplo de Árvore de Expressão



Isso traz à tona uma questão importante. A associatividade e a precedência de operadores, muitas vezes, variam de uma linguagem para outra, como veremos na Seção 7.2.2. Mesmo em uma única linguagem ou família de linguagens, a tabela de precedência pode ser bastante grande (veja, por exemplo, a Tabela 2.4). Na maioria dos casos, entretanto, os parênteses podem ser inseridos para se obter a estrutura semântica desejada.

Na ausência de regras de precedência e associatividade, a notação interfixada é inerentemente ambígua. Atribuir associatividade e precedência aos operadores é uma forma de eliminar a ambigüidade.

Uma forma alternativa de eliminar ambigüidade semântica é usar uma *notação polonesa*. A semântica dessa notação é inerentemente não-ambígua. Na *notação polonesa prefixada*, um operador binário é sempre escrito na frente dos seus dois operandos. Por exemplo, a expressão da Figura 7.1 em notação prefixada seria escrita:

```
- + a b * c d
```

Dada uma árvore de expressão, sua notação prefixada pode ser gerada usando-se uma "caminhada prefixada" (às vezes chamada de "travessia em pré-ordem") da árvore. Esse algoritmo pode ser expresso da seguinte forma, começando com o nó raiz:

1. Gerar o valor do nó.
2. Visitar a subárvore esquerda.
3. Visitar a subárvore direita.

Outra variante é chamada *notação polonesa pós-fixada*, na qual um operador sempre segue seus operandos. A expressão da Figura 7.1 em notação pós-fixada seria escrita:

```
a b + c d * -
```

A notação pós-fixada é usada nas linguagens Fortran e Postscript (uma linguagem de controle de impressão).

Dada uma árvore de expressão, a expressão em notação pós-fixada pode ser gerada usando-se uma caminhada pós-fixada (travessia em pós-ordem) pela árvore. Esse algoritmo pode ser expresso da seguinte maneira, começando novamente na raiz:

1. Visitar a subárvore esquerda.
2. Visitar a subárvore direita.
3. Gerar o valor do nó.

Uma das limitações da notação polonesa é que o mesmo símbolo não pode ser usado em uma operação com diferentes números de argumentos. Por exemplo, em matemática comum, o símbolo – é usado tanto para o sinal de menos unário quanto para o binário. Uma solução é escolher um símbolo diferente para o sinal de menos unário, como o til (~).

A notação de prefixo de Cambridge evita esse problema fazendo com que um operador sempre preceda seus operandos e colocando a expressão inteira entre parênteses. A expressão da Figura 7.1 em notação de prefixo de Cambridge seria escrita:

```
(- (+ a b) (* c d) )
```

Tabela 7.1 Associatividade de Operadores por Linguagem

Linguagem	+ –* /	– Unário	**	== != etc.
C (e similares)	esquerda	direita		esquerda
Ada	esquerda	nenhuma	nenhuma	nenhuma
Fortran	esquerda	direita	direita	esquerda

Uma vantagem da notação de prefixo de Cambridge é que operadores como + e - se tornam naturalmente n-ários. A expressão:

```
a + b + c + d
```

pode então ser escrita:

```
(+ a b c d)
```

Isso também permite interpretações naturais para `(+)`, que significa `0`, e `(+a)`, que significa `a`. A notação de prefixo de Cambridge é usada por Lisp e por Scheme (veja a Seção 14.2).

7.2.2 Associatividade e Precedência

A maioria das linguagens usa notação interfixada para expressões que envolvem operadores unários e binários. Elas também seguem as convenções matemáticas comuns para associatividade e precedência sempre que possível. Assim, a semântica refletida na árvore de expressão da Figura 7.1 pode geralmente ser obtida da seguinte forma:

```
a + b - c * d
```

A Tabela 7.1 mostra a associatividade de alguns operadores comuns para algumas linguagens.[4] O cabeçalho da coluna dos operadores relacionais usa a representação em C. Para os operadores de adição e multiplicação comuns, existe uma concordância nas linguagens mostradas. Entretanto, para o sinal de menos unário e a negação boleana, linguagens do tipo C e Fortran os tornam associativos pela direita, enquanto Ada os torna não-associativos.

Linguagens do tipo de C não possuem operador de exponenciação (** em Fortran e Ada). Fortran o faz associativo pela direita, consistente com a matemática. Ada torna a exponenciação não-associativa, forçando, assim, o programador a deixar clara tal expressão pelo uso de parênteses.

Já discutimos os problemas de associatividade de expressões relacionais, mas vale a pena repetir aqui. Linguagens do tipo de C tornam os operadores relacionais associativos pela esquerda. Todavia, a expressão `a < x < b` não é equivalente a:

```
a < x && x < b
```

como se poderia esperar. Em vez disso, em C e C++ é equivalente a:

```
se (a < ×) 1 < b então 0 < b
```

4. A representação dos operadores relacionais varia muito em cada linguagem, sendo que a representação de *não igual* é a que mais varia.

Tabela 7.2

Precedência de Operadores por Linguagem

Operadores	C (e similares)	Ada	Fortran
– Unário	7	3	3
**		5	5
* /	6	4	4
+ –	5	3	3
= = ! =	4	2	2
< <= > >=	3	2	2
"não" unário	7	2	2
"e" lógico	2	1	1
"ou" lógico	1	1	1

que não é o que se espera. Tornar os operadores relacionais não-associativos, como Ada faz, é uma decisão de projeto claramente superior.

A Tabela 7.2 mostra a precedência de diversos operadores por linguagem e mostra que há uma concordância geral quanto à ordenação relativa de operadores aritméticos binários, que os operadores relacionais têm menor precedência do que os aritméticos e que os operadores boleanos binários têm a menor precedência.

Entretanto, há um grande desacordo em relação à colocação dos operadores unários nessa hierarquia. Finalmente, linguagens do tipo de C dividem os operadores de igualdade (igual e não igual) dos outros quatro operadores relacionais assim como dos operadores lógicos.

Nem todas as linguagens usam convenções matemáticas comuns para definir a precedência de operadores. Por exemplo, a linguagem Smalltalk (veja a Seção 13.3) usa avaliação direta da esquerda para a direita, atribuindo um único nível de precedência e associatividade pela esquerda a todos os operadores. APL usa avaliação direta da direita para a esquerda.

7.2.3 Avaliação de Curto-Circuito

Seguindo a convenção matemática, os primeiros compiladores avaliavam uma expressão boleana como `A e B` primeiro avaliando `A` e `B` separadamente. A avaliação de curto-circuito avalia uma expressão boleana da esquerda para a direita e pára assim que a verdade da expressão puder ser determinada.

Assim, uma definição de curto-circuito de `A e B` é:

```
if (! A) false else B
```

De forma semelhante, uma definição de curto-circuito de `A ou B` é:

```
if (A) true else B
```

Diferentemente da matemática, `B` pode ser indefinido nesses casos, já que pode não ser avaliado.

Um exemplo clássico ocorre na pesquisa de uma lista encadeada, com o ponteiro inicial `head`, para encontrar determinado valor de chave. Usando a sintaxe Java para esse trecho de código, podemos escrever:

```
Node p = head;
while (p != null && p.info != key)
    p = p.next;
if (p == null) // not in list
    ...
else // found it
    ...
```

O teste `p.info != key` gerará uma exceção quando `p` for `null`. Entretanto, sob avaliação de curto-circuito, o teste `p.info != key` não será executado sempre que `p == null`.

A vantagem é que o programador pode escrever código menor e mais claro aproveitando a avaliação de curto-circuito. Sem ela, o laço anterior pode ser reescrito por meio de um comando `break`:

```
while (p != null) {
    if (p.info) == key
        break;
    p = p.next;
}
```

Evitar o comando `break` requer a introdução de uma variável boleana:

```
boolean found = false;
while (p != null && ! found) {
    if (p.info == key)
        found = true;
    else
        p = p.next;
}
```

A introdução de uma variável boleana torna o código mais difícil de ser entendido com relação à versão original com curto-circuito.

Uma desvantagem de expressões de curto-circuito é que este quebra a lei comutativa. Não é verdade que `a && b` seja a mesma coisa que `b && a`. Quando `a` é avaliada como falsa e `b` como indefinido, `a && b` é falso, enquanto `b && a` é indefinido.

7.2.4 O Significado de uma Expressão

Em uma linguagem de programação, o significado de uma expressão deve depender apenas dos valores das suas subexpressões e do significado do seu operador. Para simplificar, supomos aqui que a expressão tenha sido verificada estaticamente quanto aos tipos.

Uma complicação é que na matemática não há número maior ou menor. Os computadores e suas linguagens de programação suportam dois tipos de números (em diversas precisões): números inteiros e de ponto flutuante. Ambos possuem um valor máximo/mínimo.

Devido ao fato de qualquer representação ser finita, a lei associativa não vale para a aritmética computacional. Em especial:

$$(a + b) + c = a + (b + c)$$

não é verdadeiro para todos *a, b* e *c*. Para aritmética de números inteiros, se *a* for o maior inteiro positivo possível, $b = 3$ e $c = -5$, então o lado esquerdo da igualdade é avaliado como um erro (*overflow* de inteiros), enquanto o lado direito é avaliado como o maior número inteiro menos 2.

Outro problema ocorre se alguma subexpressão tiver um efeito colateral. Por exemplo, analise a expressão `a + b` , na qual `a` e `b` são valores de ponto flutuante e + denota uma adição de ponto flutuante. O significado da expressão, ou seja, seu valor, deve depender apenas dos valores de `a` e `b`. Em especial, se tanto `a` quanto `b` forem subexpressões que envolvem operadores e funções, a ordem da avaliação de `a` e `b` não deve afetar o valor da expressão.

Analise os operadores ++ e `--` na linguagem C, que podem ser problemáticos quando ocorrem dentro de uma expressão maior. Por exemplo, após os seguintes comandos serem executados:

```
i = 2; b = 2; c = 5;
a = b * i++ + c * i;
```

qual é o valor resultante de `a`? Na verdade, linguagens como C definem a expressão anterior como semanticamente indefinida, já que `a` pode ser 14 (incremente `i` após sua segunda referência) ou 19 (incremente `i` antes da sua segunda referência). As duas interpretações dependem de qual das duas subexpressões `b * i++` e `c * i` for avaliada primeiro.

Na maioria das linguagens, subexpressões em subárvores separadas podem ser avaliadas em qualquer ordem. Se o significado do programa depender da ordem, então esse programa é semanticamente indefinido. Um compilador não precisa, pelo projeto da linguagem, detectar tais erros de semântica.

7.3 O ESTADO DO PROGRAMA

Para que se entenda perfeitamente a semântica, o conceito de um *estado* de programa é fundamental.

> **Definição:** O estado de um programa é a associação de todos os objetos ativos com seus valores correntes.

O estado possui dois mapas: (1) a associação de objetos ativos com locais específicos na memória e (2) a associação de locais ativos na memória com seus valores correntes. O comando corrente (parte de uma árvore de sintaxe abstrata) a ser executado em um programa é interpretado de forma relativa ao seu estado corrente. As etapas individuais que ocorrem durante a execução de um programa podem ser vistas como uma série de transformações de estados.

Por exemplo, suponha que tenhamos variáveis *i* e *j* com valores 13 e -1 em algum momento durante a execução de um programa. Suponha que os locais de memória sejam numerados serialmente começando em 0 e que as variáveis *i* e *j* estejam associadas aos locais de memória 154 e 155 nesse momento. Então o estado representado por essa configuração pode ser expresso da seguinte forma:

$$\textit{ambiente} = \{\langle \text{i},154\rangle, \langle \text{j},155\rangle\}$$
$$\textit{memória} = \{\langle 0, \textit{undef}\rangle, \ldots, \langle 154,13\rangle, \langle 155, -1\rangle, \ldots\}$$

```
1  // calcula o fatorial de n
2  void main ( ) {
3    int n, i, f;
4    n = 3;
5    i = 1;
6    f = 1;
7    while (i < n) {
8     i = i + 1;
9     f = f * i;
10   }
11 }
```

| Figura 7.2 Programa Fatorial

O valor especial *undef* é usado para denotar o valor de um endereço de memória que esteja correntemente não definido (ainda não recebeu um valor).

Assim, o estado de um programa é o produto dos seus objetos ativos, seus locais de memória e os valores associados. Ele pode ser visto como a composição funcional dos dois mapas *ambiente* e *memória* como: *estado* = *memória* × *ambiente*. Esse tratamento formal será visto no Capítulo 9.[5]

Para os propósitos deste capítulo, é conveniente representar o estado de um programa de uma forma mais simplificada. Essa representação retira os endereços de memória e simplesmente define o *estado* do programa como um mapeamento das variáveis declaradas para seus valores atribuídos correntemente em algum momento durante a execução do programa.

O estado é como uma *janela de observação* em um ambiente de desenvolvimento integrado (IDE). Ele está sempre associado a determinado comando no programa-fonte, e mostra para cada variável do programa seu valor corrente.

Antes que um programa em C/C++ comece a ser executado, cada variável declarada possui o valor *undef* consistente com seu tipo. Qualquer operação sobre um valor indefinido torna o programa semanticamente indefinido.

Como exemplo, analise o programa da Figura 7.2, que calcula o fatorial de 3. A Tabela 7.3 mostra o rastreamento da execução desse programa em um formato que pode ser produzido pela maioria dos depuradores. A primeira linha mostra o estado do programa antes que o comando 4 (o primeiro comando executável) tenha sido executado. Em vez de mostrar o estado aqui como um conjunto de pares variável-valor, a tabela usa a exibição familiar em colunas rotuladas. Observe que há três estados separados associados ao teste do laço *while* no comando 7.

Uma característica que o rastreamento destaca é que uma atribuição modifica o estado do programa. Em comparação, já que o teste do laço `while` não tem efeitos colaterais, ele não pode modificar o estado do programa. Por outro lado, o teste do laço *while* determina se a seguir será executado o corpo do laço (linhas 8 e 9) ou o comando após este (linha 11).

5. Nos Capítulos 10 e 11, será importante representar o ambiente de uma forma mais realista, permitindo-nos caracterizar com precisão os papéis da pilha de tempo de execução e o *heap*. Isso nos permite lidar de maneira eficaz com chamadas de procedimentos, passagem de parâmetros, recursão e assim por diante.

Tabela 7.3 Rastreamento do Programa Fatorial

Passo	Antes do Comando	Variáveis		
		n	i	f
1	4	*indefinido*	*indefinido*	*indefinido*
2	5	3	*indefinido*	*indefinido*
3	6	3	1	*indefinido*
4	7	3	1	1
5	8	3	1	1
6	9	3	2	1
7	7	3	2	2
8	8	3	2	2
9	9	3	3	2
10	7	3	3	6
11	11	3	3	6

7.4 SEMÂNTICA DE ATRIBUIÇÃO

A declaração de atribuição é fundamental para o paradigma imperativo e sua ramificação mais nova, o paradigma orientado a objetos. Nesta seção, examinamos três questões relacionadas à atribuição:

1. Atribuição múltipla
2. Comandos de atribuição *versus* expressões de atribuição
3. Semântica de cópia *versus* semântica de referência em atribuições

Lembre-se da sintaxe abstrata para uma atribuição em Clite, apresentada no Capítulo 2:

Atribuição = *Variável*-`alvo`; *Expressão* `origem`

A semântica de uma atribuição é a seguinte:

> A expressão origem é avaliada no seu estado corrente, que resulta um valor, o qual substitui o valor da variável-alvo, que resulta um novo estado.

Por toda esta seção, usamos os termos *variável-alvo* e *expressão origem* com essa interpretação semântica específica.

7.4.1 Atribuição Múltipla

Muitas linguagens suportam múltiplas variáveis-alvo para uma única expressão origem. Um uso comum de tal construção é para inicializar duas ou mais variáveis com o mesmo valor, como em:

```
a = b = c = 0;
```

Este comando de atribuição inicializa todas as três variáveis com zero.

7.4.2 Comandos de Atribuição *versus* Expressões de Atribuição

Mesmo com uma atribuição com múltiplos alvos, na maioria das linguagens uma atribuição é um comando. Da mesma forma que qualquer outro comando, ela não pode ser colocada dentro de uma expressão.

Todavia, em linguagens como C, uma atribuição é uma expressão em vez de um comando. Assim, o operador de atribuição é um pouco diferente de qualquer outro operador, como +, – etc. Em C, a seguinte construção:

```
if (a = 0) ... else ...
```

é tanto sintática quanto semanticamente significativa.[6] Isso resulta no valor zero sendo atribuído à variável `a`, e a parte `else` do `if` sendo executada a seguir, já que zero é interpretado como falso. Em C, o valor de uma expressão de atribuição é o da expressão origem.

Há muitos usos práticos para a expressão de atribuição em C e C++:

- Copiar uma *string*: por exemplo, `while (*p++ = *q++) ...`
- Ler caracteres até o final do arquivo: por exemplo, `while (ch = getc(arquivo)) ...`
- Percorrer uma lista encadeada: por exemplo, `while (p = p->proximo) ...`

Entretanto, compensando essa conveniência está o erro comum apresentado anteriormente de digitar-se inadvertidamente o operador de atribuição (=) em vez da relação de iguais (==). Por esse motivo, Java discordou de C/C++ e requereu que todos esses testes fossem do tipo `boolean`.

7.4.3 Semântica de Cópia *versus* Semântica de Referência

A maioria das linguagens imperativas incluindo C e C++ usam a semântica de cópia para atribuições, na qual uma cópia do valor da expressão origem é atribuída à variável-alvo. Em comparação, muitas linguagens orientadas a objetos, incluindo Java, usam a semântica de *referência* para atribuições de objetos, na qual uma referência (ponteiro) para o valor da expressão origem é atribuída à variável-alvo.

Por exemplo, sob a semântica de cópia, a atribuição:

```
a = b;
```

deixa `a` e `b` com duas cópias do mesmo valor. Alterações em `a` ou `b` não têm efeito sobre o outro.

Sob a semântica da referência, a mesma atribuição deixa `a` e `b` apontando para o mesmo objeto, que é o valor original de `b`. Assim, alterar o valor (estado interno) de `a` ou `b` terá o efeito colateral de alterar o outro, ou seja, a atribuição anterior torna as variáveis `a` e `b` em dois aliases do mesmo objeto.

6. Embora o programador presumivelmente tenha desejado testar se a variável `a` é igual a zero. Os compiladores, muitas vezes, emitem um aviso para alertar o programador para o fato de que este pode ter pretendido usar == em vez de = .

Um exemplo mais ilustrativo é o método `add` (Tucker e Nonnan, 2002, p. 191):

```
public void add (Object word, Object number) {
    Vector set = (Vector) dict.get(word);
    if (set == null) { // vetor vazio
        set = new Vector( );
        dict.put(word, set);
    }
    if (allowDupl || !set.contains(number))
        set.addElement(number);
}
```

O primeiro comando atribui a `set` o valor da entrada com a chave `word` na tabela *hash* `dict`. Se essa chave não estiver na tabela *hash*, `set` recebe o valor `null`, em cujo caso um novo `set` é criado e colocado na tabela *hash* com a chave `word`. Em ambos os casos, o último comando do método adiciona o inteiro `Number` ao `set` da tabela *hash*. Sob a semântica de cópia, esse último comando não atualizaria a tabela *hash* apropriadamente.

7.5 SEMÂNTICA DE CONTROLE DE FLUXO

De acordo com o Teorema de Böhm-Jacopini (Böhm e Jacopini, 1966), uma linguagem é completa em relação a *Turing* se, além do comando de adição, possuir as seguintes estruturas de controle de fluxo: seqüência, um comando condicional e um comando de laço.

> **Definição:** Uma linguagem de programação é completa quanto a Turing se os seus programas forem capazes de computar qualquer função calculável.

Cada uma dessas estruturas será discutida nas seções a seguir. Chamadas a funções/métodos e retornos são discutidos no Capítulo 9.

Qualquer outro tipo de comando aumenta a automação ou a conveniência de uma linguagem de programação e sua expressividade, mas, em última instância, não o seu poder computacional.

7.5.1 Seqüência

A capacidade de escrever uma seqüência de comandos em uma linguagem é tão natural e básica que raramente é discutida. No caso mais simples, uma seqüência possui dois comandos justapostos:

$c_1 c_2$

A semântica de tal seqüência é intuitiva. Na ausência de alguma forma de comando de desvio, primeiro c_1 é executado, e em seguida, c_2.

Como vimos na Tabela 7.3, o estado de saída de c_1 é o de entrada de c_2.

Comandos de desvio incluem: `return`, `break`, `continue` e o comando irrestrito `goto`. Esse último comando e seus problemas associados são discutidos na Seção 7.5.4. Os outros são formas restritas de comandos `goto`.

O comando `return` é usado para terminar imediatamente uma função ou um método e é discutido mais integralmente no Capítulo 9. Os comandos `break` e `continue` são usados principalmente em laços e são discutidos na Seção 7.5.3.

7.5.2 Condicionais

O comando condicional é usado para selecionar caminhos alternativos durante a execução de um programa. Na maioria das linguagens, ele apresenta duas variações: o comando `if` básico e o comando `case` (ou `switch`).

Conforme vimos no Capítulo 2, o comando `if` (usando a sintaxe de C) apresenta duas variações:

ComandoIf → `if (` *Expressão* `)` *Comando*1 [`else` *Comando*2]

que são, muitas vezes, chamadas de *if-then* e *if-then-else*. A semântica de um *ComandoIf* pode ser definida da seguinte forma:

> Se o valor da *Expressão* for verdadeiro, o significado é o estado resultante da avaliação do *Comando1* no estado corrente. Caso contrário, o significado é o estado resultante da avaliação do *Comando2* no estado corrente.

Lembre-se do Capítulo 2, que diz que *if-then* pode ser visto como um *if-then-else* com um comando de salto inserido na parte do *else*. Assim, a existência de um comando *if-then* é apenas uma abreviação conveniente.

Um exemplo de um *if-then-else* é o seguinte trecho de código (em sintaxe C) para calcular o máximo entre dois números:

```
if (a > b)
    z = a;
else
    z = b;
```

Uma questão interessante é como especificar que o *Comando1* ou o *Comando2* possuem mais de um único comando. Duas abordagens gerais têm sido usadas.

A abordagem de Algol 60 é limitar tanto a parte do *then* quanto a do *else* em um único comando. Já que isso é claramente inconveniente, Algol 60 também usou um mecanismo de agrupamento que permitiu que uma seqüência de comandos fosse considerada sintaticamente como um único comando. Algol 60 usou `begin` e `end` para denotar tal seqüência, que é chamada de comando composto. Linguagens como C e Perl seguem essa tradição, mas usam uma chave esquerda para `begin` e uma chave direita para `end`.

Algol 68 via o *Comando1* e o *Comando2* como naturalmente compostos, precisando apenas de uma palavra-chave para terminar o *Comando2* (se existir) ou o *Comando1* (se o *Comando2* não existir). Algol 68 introduziu a convenção de usar a palavra-chave inicial escrita ao contrário. Assim, um comando *if* era terminado com um *fi*. Ada segue esta convenção, mas usa duas palavras-chave para terminar, a saber, *endif*. Python também segue essa tradição, mas usa indentação para o *Comando1* e o *Comando2*.

Para simplificar o fechamento de comandos *if* aninhados, linguagens que seguem a convenção do Algol 68 muitas vezes têm uma construção `elseif` nos seus comandos `if`. Usando Ada, por exemplo, a sintaxe de um comando `if` é:

ComandoIf → `if` *Expressão* `then` *Comandos*
{`elsif` *Expressão* `then` *Comandos*}
[`else` *Comando*]
`end if ;`

A ortografia da palavra-chave `elseif` varia de acordo com a linguagem; até Perl, que segue a tradição de C, possui um `elsif`. O comando *switch* (ou *case*) começou como uma tabela de saltos e foi implementado como um *goto* ou *jump*. Os *cases* individuais não tinham de estar fisicamente adjacentes, mas poderiam estar dispersos por todo o código. Um exemplo em Fortran é:

```
goto (100, 200, 300, 400), i
```

Se a variável `i` tiver valor igual a 1, o próximo comando executado é o rotulado como 100, se tiver valor igual a 2, então é 200, e assim por diante. Se o seu valor não estiver dentro da faixa 1 a 4, então o comando `goto` é pulado e o controle passa para o próximo comando.

Devido a essa tradição e implementação associada, a maioria das linguagens limita a variável de controle do comando *switch* ao tipo inteiro (incluindo caracteres únicos). Se os *cases* excederem uma pequena faixa de valores, muitos compiladores implementam um *switch* usando um *if*. Um exemplo de um comando *switch* em linguagens C aparece na classe tokenizer na Figura 3.5. Uma decisão infeliz de projeto para comandos *switch* em linguagens do tipo de C é o requisito de que cada *case* termine com uma declaração `break` explícita para evitar que um *case* caia acidentalmente no próximo. Isso se provou muito propenso a erros; assim, é lamentável que linguagens mais novas como Java e C# tenham continuado com esse projeto.

Em linguagens distintas de C e suas variantes, como Ada, existe um *break* implícito no final de cada *case*. Além disso, Ada também permite que o rótulo de um *case* seja uma subfaixa inteira de valores. Assim, o comando *switch* da Figura 3.5 apareceria em Ada como:

```
case ch is
    when 'A' .. 'Z' | 'a' .. 'z' => -- identifier
        ...
    when '0' .. '9' => -- numeric literal
        ...
    when
        ...
    when others => -- default
        ...
end case;
```

Observe que em Java os primeiros dois *cases* são escritos de forma mais conveniente como comandos *if*. Observe também que Ada usa `--` para começar um comentário e `end case` para terminar o comando `case`.

7.5.3 Laços

O *laço com teste no início* é adequado para todas as necessidades de laços. Em linguagens do tipo de C, esse laço é escrito como

ComandoWhile → `while` (*Expressão*) *Comando*

e é chamado de laço `while`. Seu significado é resumido a seguir:

> A *Expressão* é avaliada. Se ela for verdadeira, o *Comando* é executado e o *Laço* é executado novamente. Caso contrário, o laço termina.

Em comparação, o *laço com teste no final* em linguagens do tipo de C aparece sintaticamente da seguinte forma:

ComandoDoWhile → `do` *Comando* `while` (*Expressão*)

Seu significado é ligeiramente diferente daquele do *laço com teste no início:*

> O *Comando* é executado uma vez e então a *Expressão* é avaliada. Se a *Expressão* for verdadeira, o *Comando* é executado novamente, e assim por diante. Caso contrário, o laço termina.

Tal construção é, dessa forma, semanticamente equivalente a:

{ *Comando* `while` (*Expressão*) *Comando* }

Assim, o *laço com teste no final* é conveniente, porém redundante.

Ainda mais conveniente é o *laço com contador*. Um exemplo de C para somar os elementos de uma matriz está a seguir:

```
sum = 0.0;
for (i = 0; i < SIZE; i++)
    sum += a[i];
```

Na linguagem C, a variável `i` teria de ser declarada no início do bloco no qual o código aparece. Linguagens mais novas como C++, Java e C# permitem que a redeclaração de `i` ocorra como parte da inicialização do laço. Nesse caso, o escopo de `i` é limitado ao próprio comando *for*. Da mesma forma que com o *laço com teste no final*, o *laço com contador* também é conveniente, porém redundante.

Ainda mais convenientes, são os laços *for-each*, que usam alguma forma de iterador para atravessar uma estrutura de dados, acessando cada elemento da estrutura de dados uma vez.

> **Definição:** Um *iterador* é qualquer conjunto finito de valores sobre os quais um laço pode ser repetido.

Como exemplo, analise a impressão das chaves e os valores de uma matriz associativa (ou tabela *hash*). Na linguagem Perl, isso poderia ser escrito:

```
foreach $key (sort keys %list) {
   print $key, "\t", $list{$key}, "\n";
}
```

Construções de laços *for-each* aparecem em Python, Java (desde a versão 1.5) e C#.

Muitas linguagens fornecem duas formas limitadas de comandos `goto` para tornar o controle do laço mais flexível. O primeiro desses comandos é denominado comando `break` em linguagens do tipo de C. O propósito do comando `break` é o de permitir que um laço seja concluído de qualquer lugar do corpo do laço.

Um uso natural de tal construção ocorre na exibição de menus ou caixas de diálogo e para se obter a resposta do usuário. Um esboço de tal código na linguagem C é:

```
while (1) {
    resp = displayMenu( );
    if (resp == QUIT)
        break;
    ...
}
```

A outra forma limitada de desvio é denominada comando `continue` em linguagens do tipo C. Seu propósito é o de abortar a iteração corrente de um laço e começar a próxima. O comando `continue` pode ser pensado como um desvio para o final do laço.

Um exemplo do uso de um comando `continue` é o seguinte trecho em Java. Esse laço está lendo um arquivo de configuração, uma linha por vez. Cada linha é processada pela informação que contém. Entretanto, linhas em branco e linhas que começam com um sinal de cerquilha (*#*) são ignoradas.

```
while (true) {
    String line = infile.readLine( );
    if (line == null) // final de arquivo
        break;
    line = line.trim( );
    if (line.length == 0) continue;
    if (line.startsWith("#")) continue;
    ...
}
```

7.5.4 A Polêmica do GoTo

A controvérsia do comando `goto` começou com uma carta de Edsger Dijkstra para o editor da *Communications of the ACM* (Dijkstra, 1968b) intitulada "GoTo Considerado Prejudicial". Com essa carta, a revolução da *programação estruturada* estava lançada.

De modo geral, a programação estruturada se referia ao projeto de programas a partir de princípios básicos, por intermédio de apenas três estruturas de controle básicas: a seqüência, o comando condicional e o laço *while*. Em especial, a programação estruturada defendia um banimento do uso de qualquer forma de comando `goto`, já que era desnecessário e encorajava a escrita de código incompreensível. Böhm e Jacopini (Böhm e Jacopini, 1966) mostraram que o comando `goto` era logicamente desnecessário. Desde então, algumas linguagens (como Ada e Java) excluíram o comando `goto` dos seus vocabulários.

Na década de 1960, a principal ferramenta de projeto de programas era o fluxograma (veja um exemplo na Figura 12.1). O uso de fluxogramas para o projeto de programas encorajava o assim chamado *código espaguete*, ou seja, código cuja estrutura estava toda entrelaçada. A presença de estruturas lógicas equivalentes a laços e comandos condicionais era, muitas vezes, difícil de ser discernida. Um aspecto da programação estruturada era tornar a estrutura lógica dos programas mais óbvia, eliminando o código espaguete.

Analise o fragmento de programa em Fortran da Figura 7.3, que soma os valores não-negativos de um conjunto de números. Um contador dos números somados também é calculado.[7]

7. Em Fortran, o comando `continue` é usado para marcar o final de um laço ou qualquer outro ponto de desvio, diferentemente do seu uso em linguagens do tipo de C.

```
      i = 0
      ct = 0
      sum = 0.0
100   if (i .ge. n) goto 300
      i = i + 1
      if (a(i) .lt. 0) goto 100
      ct = ct+ 1
      sum = sum + a(i)
      goto 100
300   continue
```

| Figura 7.3 Exemplo de Código Espaguete em Fortran

Uma leitura cuidadosa deve nos convencer de que o programa funciona, mas a estrutura de controle não é óbvia. Um laço *while* teria fornecido uma solução mais simples e legível.

O desenvolvimento no final da década de 1960 das linguagens C e Pascal estimulou o desenvolvimento da programação estruturada. Diferentemente de Fortran 66, elas tinham estruturas de laço e condicionais necessárias, incluindo os laços com teste no início e com teste no final, laços *for*, comandos *if-then-else* e comandos *switch/case*, junto ao comando composto que permitia o aninhamento dessas construções. C também incluiu formas restritas de comandos `goto`, englobando os comandos `return`, `break` e `continue`, conforme discutido anteriormente.

Entretanto, Dijkstra e Wirth estavam interessados principalmente em substituir o fluxograma por métodos melhores de projeto de programas que facilitassem a demonstração da correção de programas (veja o Capítulo 18). O texto de Wirth (Wirth, 1973) e o relatório de Dijkstra (Dijkstra, 1972) foram extremamente influentes quanto a isso.

De modo geral, a revolução na programação estruturada foi um sucesso. Os programadores abandonaram o comando irrestrito `goto` em favor de construções explícitas de laços e condições. É extremamente raro atualmente se encontrar um comando `goto` incondicional, embora seu uso ainda seja ocasionalmente defendido (Rubin, 1987). Embora a programação estruturada esteja firmemente estabelecida como metodologia de projeto dentro do paradigma da programação imperativa, sua utilidade é menos óbvia no mundo orientado a objetos.

7.6 SEMÂNTICA DE ENTRADA/SAÍDA

Toda programação prática necessita do domínio das técnicas de recuperação de dados de uma fonte permanente remota e do armazenamento de dados em um destino permanente remoto. Coletivamente, essas técnicas são chamadas de *entrada/saída*, ou apenas *E/S*. As fontes e os destinos para as operações de entrada e de saída são denominados *arquivos*. A mídia em que os arquivos são armazenados é muito variada e inclui, por exemplo, o teclado, o monitor, um disco magnético ou um *pen-drive*. Nesta seção, revisamos algumas das características semânticas que se relacionam com a programação de E/S em diversas linguagens.

No decorrer dos anos, os projetistas de linguagens têm usado abordagens muito diferentes para a incorporação de recursos de entrada/saída nas suas linguagens. Os projetistas de Algol, por exemplo, viam a E/S como sendo tão dependente da implementação que omitiram sua definição da especificação Algol 60. Projetistas de linguagens posteriores, entretanto, foram forçados a incluir especificações de E/S, geralmente escolhendo incluí-las nas suas bibliotecas de funções-padrão.

7.6.1 Conceitos Básicos

Antes que um programa possa começar a acessar um arquivo, este deve ser localizado e aberto. Uma vez aberto, as informações de um arquivo podem ser acessadas de uma entre duas maneiras: *seqüencialmente* ou *aleatoriamente*. Um arquivo acessado seqüencialmente é um arquivo no qual cada entidade é armazenada na mesma ordem em que é lida/gravada pelo programa. Um arquivo acessado aleatoriamente é um arquivo cujas entidades podem ser lidas/gravadas em alguma ordem não-serial, de modo que uma única entidade possa ser lida/gravada zero, uma vez ou qualquer número de vezes enquanto o arquivo estiver aberto.

Quando um programa termina de ler/gravar entidades em um arquivo, este é "fechado". Dessa forma, o programa abandona o controle para que o arquivo possa ser acessado por outros programas posteriormente. Geralmente, nenhum arquivo é aberto por mais de um programa por vez.[8]

As entidades em um arquivo acessado seqüencialmente podem ser recuperadas/gravadas por um programa como um *fluxo* de caracteres ou como uma série de *registros* de tamanho fixo, cada um dos quais possuindo a mesma estrutura ou *formato* dos outros. Entidades em arquivos acessados aleatoriamente são geralmente lidas e gravadas como registros de tamanho fixo.

Arquivos-padrão Um requisito fundamental na programação é a capacidade de se ler dados de um teclado e exibi-los na tela. Essas duas mídias são identificadas como "arquivos-padrão" em linguagens do tipo de C. Essas linguagens também fornecem um "arquivo de erro-padrão", em que as mensagens de erro que ocorrerem durante a execução do programa são armazenadas serialmente. Java identifica esses três arquivos-padrão como `System.in`, `System.out` e `System.err`, respectivamente. Essa convenção é adaptada de C, na qual esses arquivos são chamados `stdin`, `stdout` e `stderr`, respectivamente.

Java, C++ e Ada requerem que o programa mencione explicitamente o nome dos pacotes de E/S (`java.io`, `iostream` e `Ada.Text_IO`, respectivamente) antes que possa abrir algum desses arquivos para leitura ou gravação. Um programa em Fortran, C++ ou Ada pode acessar dados usando E/S formatada ou não-formatada.

A Figura 7.4 mostra um exemplo de programa em C++ que lê valores inteiros para uma matriz `a` usando a entrada-padrão, e depois exibe os valores em uma lista usando a saída-padrão.

```
#include <iostream.h>
main()
{
   int i, a[8];
   cout << "Enter a series of eight integers: \n";
   for (i=0; i<8; i++)
      cin >> a[i];
   for (i=0; i<8; i++)
      cout << a[i] << '\n';
}
```

| Figura 7.4 Exemplo de Uso de E/S-Padrão em C++

8. Há exceções, como no caso de um banco de dados bancários que é acessado simultaneamente por programas que servem a clientes em diferentes caixas automáticos (ATMs).

Java 1.5 suporta a leitura de um valor numérico do teclado em um estilo não-formatado, por meio da classe `Java.util.Scanner`. Essa classe analisa um fluxo de entrada em uma série de *tokens* à medida em que ele é lido. A convenção para fazer isso é usar o método `nextInt`, que encontra o próximo *token*, o converte para `int` e o retorna.

```
Scanner reader = new Scanner(System.in);
int i = reader.nextInt();
```

Versões mais antigas de Java requeriam que a entrada passasse por um `InputStreamReader` e um `BufferedReader` e depois fosse analisada antes que pudesse ser atribuída a uma variável `int`, ou seja, o seguinte código seria necessário:

```
BufferedReader br = new BufferedReader(
                    new InputStreamReader(System.in));
String str = br.readLine();
int i = Integer.parseInt(str);
```

Fluxos de Entrada/Saída Como padrão, as linguagens suportam uma entrada e uma saída seqüenciais em um estilo "não-formatado", ou seja, as informações são lidas/gravadas como um fluxo de caracteres, do qual valores de *tokens* individuais são distinguidos, convertidos para o tipo apropriado e atribuídos a variáveis na ordem em que estas são listadas no comando de leitura ou gravação. Esse estilo de programação de E/S é especialmente útil para programadores iniciantes, conforme ilustrado na Figura 7.4.

A entrada/saída não-formatada em Fortran é denominada "direcionada a listas", o que é análogo a um "fluxo" de E/S em C++ ou Java. A aparência de um comando READ ou WRITE em Fortran usando a E/S-padrão é:

```
READ (*, *) variable-list
WRITE (*, *) expression-list
```

A seguir, encontra-se um programa em Fortran que imita o código em C++ da Figura 7.4.

```
INTEGER :: i, a(8)
WRITE (*, *) "Enter a series of eight integers: "
READ (*, *) a
WRITE (*, *) a
```

As referências a `a` nos comandos READ e WRITE designam laços na forma de `(a(i), i=1,8)`. Tal referência pode ser usada sempre que o laço envolver todos os elementos da matriz.

Em Java, a biblioteca `java.io` fornece muitos tipos diferentes de fluxos para a entrada/saída seqüencial, incluindo fluxos de arquivo, fluxos de *pipes*, fluxos de memória e fluxos de filtro. Um *fluxo de arquivo* é usado para transferir dados para o sistema de arquivos nativo. Um fluxo de *pipe* faz com que os dados sejam enviados diretamente de um fluxo para outro. Isso permite que um programa evite armazenar dados em estruturas de dados intermediárias. *Fluxos de memória* são usados para transferir dados entre matrizes e o programa, enquanto *fluxos de filtro* fornecem suporte a Java para a formatação da entrada e da saída.

Fluxos de E/S em Java também suportam um método `mark`, que identifica o local corrente de um fluxo. Se um método `reset` for chamado posteriormente, o programa continuará lendo no local identificado no fluxo. *Fluxos de leitura* possuem métodos `read` que obtêm um caractere do fluxo, armazenam toda a entrada disponível em uma matriz de caracteres ou armazenam a entrada em uma parte especificada de uma matriz de caracteres. *Fluxos de gravação* suportam métodos `write` semelhantes para gravar em fluxos e usam um codificador de caracteres para converter dados entre o formato de caractere e o de byte. Leitores e gravadores de *matrizes* são usados para executar operações de E/S em blocos inteiros de memória.

Java também possui convenções para *serializar* objetos usando fluxos. A serialização é um processo pelo qual os objetos podem ser arquivados para instanciação posterior no mesmo programa ou para transmissão entre computadores por meio de *sockets*. Por intermédio da conversão de valores buscados do fluxo de entrada para serialização de objetos, sejam como *strings* ou como o tipo original do objeto, é possível recuperar os objetos e recriá-los posteriormente na sua forma original.

A classe Java `StreamTokenizer` fornece quatro ferramentas que são úteis para diversas atividades de tokenização: `nextToken`, `TT_EOF`, `sval` e `nval`. O método `nextToken` varre o fluxo serialmente, começando na posição corrente e pulando os espaços em branco até localizar e retornar o próximo *token* que encontrar. `TT_EOF` é uma constante especial que indica que o final do arquivo foi encontrado. Os valores `sval` e `nval` contêm o valor do próximo *token* analisado como uma `String` e como um `double`, respectivamente.

Os procedimentos de Ada para abrir, fechar e acessar arquivos seqüenciais espelham aqueles de C++ e Java. Em Ada, `Open` associa um arquivo definido pelo programa com um arquivo externo que já existe, enquanto `Create` cria um novo arquivo externo. `Close` termina com a associação entre o programa e o arquivo, e `Delete` termina com essa associação e também apaga o arquivo externo.

Para um programa em Ada determinar a situação corrente de um arquivo, são fornecidas as funções `Is_Open`, `Name`, `Form` e `Mode`. `Is_Open` retorna um valor boleano que descreve se o arquivo está aberto ou não, enquanto `Name` retorna o nome associado ao arquivo. `Form` retorna informações específicas de sistema operacional sobre o arquivo, como privilégios de leitura/gravação e a quantidade de espaço em disco alocada inicialmente para ele. `Mode` informa se o arquivo é do tipo `In_File`, `Out_File` ou `Inout_File`. Este último modo é reservado para arquivos de acesso aleatórios, que são discutidos mais adiante nesta seção.

E/S Seqüencial Formatada E/S formatada é fornecida em C, Fortran, Java 1.5 e Ada. Entretanto, ela possui uma forma diferente em cada uma dessas linguagens. A seguir está um resumo sobre isso.

Em C, o comando `fopen` fornece o nome de arquivo e as permissões sob as quais ele deve ser aberto. A leitura e a gravação em um arquivo (uma vez aberto) podem ser executadas por meio das funções de biblioteca `fscanf` e `fprintf`. Aqui está um exemplo em C que lê informações para uma matriz a partir de um arquivo chamado `input.asc`.

```
FILE* input;
input = fopen("input.asc", "r");
for (i=0; i<8; i++)
    fscanf(input, "%d", &a[i]);
```

Aqui, o `%d` no segundo parâmetro é uma dica de que a função `fscanf` deve encontrar um número inteiro (em contraste com `%f` para um *float* ou `%s` para uma *string*), pulando espaços

em branco nesse ínterim. Esse é um processo híbrido entre E/S formatada e não-formatada: não-formatada porque quantidades arbitrárias de espaços em branco (espaços, novas linhas etc.) podem ocorrer entre valores adjacentes, e formatada porque os valores lidos devem ser do tipo especificado.

Além de `fscanf` e `fprintf`, C fornece as funções `getc` e `putc`, que recebem e enviam um único caractere, e `fgets` e `fputs`, que enviam e recebem uma linha inteira de caracteres de/para um arquivo para/de uma variável *string*. Todas as funções de entrada entre essas retornam `EOF` quando o final do arquivo é atingido.

Em Fortran, E/S formatada é mais rigorosa do que em C, já que o espaço em branco não é pulado automaticamente entre entidades de entrada. O formato de um comando READ ou WRITE em Fortran é:

```
READ (file_unit, format_expression) variable-list
WRITE (file_unit, format_expression) expression-list
```

Se `file_unit` for um asterisco, supõe-se a entrada ou a saída padrão. Caso contrário, `file_unit` é um inteiro que identifica um arquivo diferente do *buffer* de entrada-padrão, em cujo caso um comando OPEN deve preceder o comando READ:

```
OPEN (UNIT=file_unit, FILE='f', STATUS='OLD', &
      ACTION='READ',IOSTAT=s)
```

Aqui, as seguintes informações são fornecidas:

`file_unit` conecta o arquivo aos comandos `READ` subseqüentes.

O nome de arquivo `'f'` é o nome externo do arquivo na mídia em que ele reside permanentemente.

O arquivo deve ser `OLD`, indicando que já existe.

Opcionalmente, a variável `s` de valor inteiro do tipo `IOSTAT` pode ser configurada para indicar o status da operação de e/s (ou seja, que o arquivo foi aberto com sucesso).

O comando FORMAT permite a programas em Fortran controlarem o local e a aparência de valores de dados individuais em um fluxo de entrada e de saída. Muitos tipos diferentes de descritores de formatos estão disponíveis para esse propósito. Os quatro descritores seguintes são usados com freqüência:

`I`*n* – indica um inteiro na posição *n*.

`F`*n.d* – indica um valor de ponto flutuante na posição *n*, com *d* dígitos à direita do ponto decimal.

`A`*n* – indica um valor de *string* de *n* caracteres.

n`X` – indica um salto de *n* caracteres.

Na impressão, a primeira posição de cada linha de saída deve conter um caractere de controle que defina o espaçamento vertical para essa linha (espaço simples, espaço duplo etc.).[9] Aqui está uma variante em Fortran do código C anterior para inicializar a matriz `a` a partir do arquivo de entrada `input.asc`.

9. Este é um remanescente do tempo em que Fortran produzia saída principalmente para impressoras de linhas, que podiam imprimir exatamente 132 caracteres por linha, e cada caractere tinha uma fonte fixa (como a fonte *courier* de hoje).

```
OPEN (UNIT=3, FILE='input.asc', STATUS='OLD', &
   ACTION='READ', IOSTAT='ioerror')
IF (ioerror == 0) READ (3, 110) a
   110 FORMAT ( 8I5 )
```

Aqui, o comando FORMAT diz que cada valor lido e armazenado em `a` é recuperado do arquivo como um inteiro de 5 dígitos, existindo oito desses.

Uma string de caracteres pode ser colocada na expressão do FORMATO para incluir informações adicionais na saída. O comando a seguir exibe uma mensagem seguida pelos oito valores de `a` no arquivo de saída padrão; cada valor ocupa cinco posições decimais na linha.

```
WRITE (*, 120) a
120 FORMAT ('Contents of array a are: ', 8I5
```

Observe o espaço em branco no início da mensagem, que indica que essa linha de saída tem um único espaço.

Antes da versão 1.5, Java usava `FilterOutputStream`s e `FilterInputStream`s para gerenciar sua formatação de E/S. As versões básicas dessas classes fornecem outro fluxo de entrada/saída por meio do qual os programadores poderiam criar subclasses de fluxos de filtros para definir seus próprios fluxos de filtros. Esse método é, no mínimo, incômodo, e leva a inconsistências entre diferentes aplicações em Java para formatação de E/S.

O lançamento de Java 1.5 inclui a classe `java.util.Formatter`, que contém métodos de formatação de saída semelhantes àqueles encontrados em C. Esse recurso pode levar a convenções de formatação mais padronizadas e fáceis para programas em Java. Por exemplo, o seguinte código Java 1.5 exibe uma *string* e um inteiro de 3 dígitos:

```
System.out.printf("%s %3d", name, age);
```

Em Ada, a biblioteca Test_IO fornece dois procedimentos altamente sobrecarregados, `Get` e `Put`, que permitem aos programas especificar entrada e saída formatadas para diversos tipos de valores. Por exemplo, aqui estão algumas variantes do procedimento `Get` para a entrada de valores em *Character*, *Integer* e *String*, respectivamente.

```
procedure Get (File: in File_Type; Item: out Character);
procedure Get (File: in File_Type; Item: out Integer);
procedure Get (File: in File_Type; Item: out String);
```

Aqui, o parâmetro `out` designa uma variável na qual um valor character/string/integer deve ser armazenado.

Cada um desses valores possui um parâmetro opcional `Width`, que permite ao programa especificar o número de caracteres a serem lidos de um arquivo antes que ele encontre um terminador de linha. Isso é semelhante à convenção de Fortran, que define o comprimento de um campo que contém um valor de dado. Se `Width=0` for especificado, o programa tentará ler tantos caracteres quanto puder interpretar como sendo do tipo apropriado. Por exemplo, suponha que `i` seja uma variável do tipo `Integer` e:

```
myFile: File_type;
Get (myFile, Item => i, Width => 0);
```

seja executado sobre o fluxo de entrada:

```
213a345
```

Isso recuperará o valor `213` da entrada e o atribuirá a `i`.

A E/S formatada em Ada permite que o programa especifique o número de dígitos antes e depois do decimal, o número de dígitos exponenciais a serem incluídos e a base (binária, hexadecimal ou octal) a usar para exibir o valor. Por exemplo, o seguinte comando:

```
Put (myFile, Item => 1024, Base => B);
```

exibe

```
10000000000
```

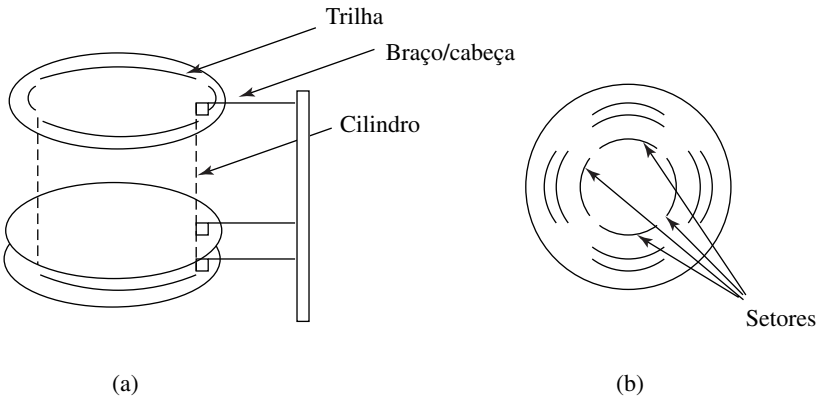
que é o inteiro 1.024 em notação binária.

Concluímos esta seção observando que, embora a E/S seqüencial formatada e não-formatada seja uma atividade de programação razoavelmente direta e central, seus detalhes e suas idiossincrasias de programação variam bastante entre diferentes linguagens.

7.6.2 Arquivos de Acesso Seqüencial

A E/S seqüencial possui duas limitações sérias quando são processados arquivos grandes. A primeira é que nem todas as aplicações acessam as informações em um arquivo na mesma seqüência na qual elas estão armazenadas. Em segundo lugar, já que registros individuais em um arquivo seqüencial são armazenados adjacentemente uns aos outros, qualquer tentativa de substituir um registro por um registro maior arrisca sobrescrever a parte inicial do próximo registro na seqüência.

Arquivos de *acesso aleatório (ou acesso direto)* permitem o processamento não seqüencial de informações em um arquivo. Tais arquivos ocorrem em diretórios e em outras aplicações que precisem evitar o processamento de todos os registros iniciais serialmente apenas para encontrar ou atualizar uma única entrada.



| Figura 7.5 A Estrutura de Dispositivos de Armazenamento de Discos

Dispositivos de armazenamento em disco são projetados para lidar com o acesso aleatório a arquivos. A Figura 7.5a mostra a estrutura de tal dispositivo. Um disco possui múltiplos pratos, cada um fornece diversas trilhas circulares concêntricas. Cada trilha possui o mesmo número de setores de tamanho fixo (Figura 7.5.b). Uma série de cabeças de leitura/gravação, uma por superfície de prato, fica posicionada para ler/gravar informações em um único cilindro antes que o braço as mova para outro cilindro. Esse movimento de braço é chamado "busca".

Em programas em C, uma `struct` pode ser usada para definir um registro individual em uma aplicação de acesso aleatório a arquivos. Como vimos no Capítulo 5, uma `struct` é simplesmente um rótulo para um conjunto de variáveis com tipos diferentes.[10] Aqui está um exemplo de uma `struct` em C que define um registro de dados de uma aplicação de fluxo de fluidos, os quais estão organizados em uma matriz com `nRows x nCols` de registros desse tipo, armazenados em um arquivo de acesso aleatório.

```
typedef struct grid_t {
 float nRows, nCols;
 int noData;
 int dir;
} grid_entry;
```

As bibliotecas C padrão contêm os seguintes métodos para acessar dados aleatoriamente: `fseek`, `ftell`, `rewind`, `fgetpos` e `fsetpos`. Estes métodos permitem a programas em C buscar um registro específico, informar qual registro será lido ou gravado no momento, retornar ao início do arquivo, informar a posição corrente do arquivo e redefinir uma nova posição corrente do arquivo, respectivamente.

Ao acessar aleatoriamente um registro em um arquivo, o comando `fseek` é usado primeiro para reposicionar o braço de leitura/gravação no cilindro em que o registro está localizado. Um comando `fseek` tem a seguinte forma:

```
fseek(FILE *stream, long offset, int origin)
```

Aqui, `offset` é o deslocamento em bytes do próximo registro a partir da `origem` do arquivo, em que a distância para se reposicionar o braço de leitura/gravação pode ser calculada.

A função `sizeof` de C retorna o número de bytes de que uma `struct` necessitará quando armazenada em um arquivo, e assim o parâmetro `offset` para `fseek`, para buscar o registro de índice `n` no arquivo, pode ser escrito como `(n-1) * sizeof(struct)`.

O parâmetro `origin` pode ter um entre três valores constantes, definindo a posição corrente (SEEK_CUR) no arquivo, o início (SEEK_SET) do arquivo ou o final (SEEK_END) deste. O programa então busca os dados movendo-se de uma `origin` dada até um número `offset` de bytes em preparação para a próxima operação de `fscanf` ou `fprintf` nesse arquivo.

A Figura 7.6 mostra uma aplicação de acesso aleatório em GIS, no qual uma posição na direção NORTHWEST a partir da posição corrente no arquivo `GIS.data` é recuperada de um modo não-seqüencial. Aqui, `x` e `y` definem as coordenadas da posição corrente, cujas informações estão armazenadas na `struct grid_entry` definida anteriormente.

A função `fseek` aqui reposiciona o arquivo no início daquela posição que está na linha e na coluna anteriores (ou seja, o par x-1 e y-1 designa NORTHWEST). A grade completa de posições é armazenada no disco com as linhas, identificando a maior ordem (semelhante

10. Ada possui o tipo `record`, que serve para o mesmo propósito da `struct` em C.

```
FILE* input;
input = fopen("GIS.data", "r");
...
x = /x column of entry */;
y = /* row of entry */;
/* seek to the "dir" variable NORTHWEST of grid_entry */
fseek(input, sizeof(grid_entry) * ((y-2) * nCols
   + (x-1)) + 2 * sizeof(float) + sizeof(int), SEEK_SET);
fscanf(input, "%d", &temp);
if (temp == SOUTHEAST)
   return true;
...
```

| Figura 7.6 Segmento de Código em C que Acessa Aleatoriamente uma Posição em uma Grade

ao esquema de armazenamento para matrizes discutido no Capítulo 5). Assim, o cálculo `offset = sizeof(grid_entry)*((y-2)*nCols+(x-1))` identifica o início dos dados referentes a essa posição no arquivo, e o valor incremental `2*sizeof(float)+sizeof(int)` indica a terceira posição após essa posição corrente. Assim, a função `fscan` que se segue ao comando `fseesk` na Figura 7.6 recupera o valor `dir` para a posição na direção NORTHWEST a partir da posição corrente.

Em Fortran, os programadores usam uma cláusula `RECL` e `ACCESS = DIRECT` no comando `OPEN` para especificar que o arquivo será acessado aleatoriamente. `RECL` é usado para especificar o comprimento de um registro individual no arquivo que estiver sendo aberto. Em seguida, cada comando READ ou WRITE que acesse um registro no arquivo deve especificar o número do registro, usando o parâmetro `REC` para especificar a posição do registro dentro do arquivo.

A seguir, está um par de comandos em FORTRAN que simulam os comandos `fopen`, `fseek` e `fscanf` da Figura 7.6, supondo que `x` e `y` representem a coluna e a linha da posição corrente.

```
OPEN(7, ACCESS= 'DIRECT', RECL=16)
READ(7, REC = (y-1)*nCols + x) nCols, nRows, noData, dir
```

O comprimento do registro representado aqui supõe que cada valor REAL e INTEGER ocupa 4 bytes de memória, de modo que o equivalente em FORTRAN a toda `struct grid_entry` em C ocupa 16 bytes.[11]

Observamos finalmente que a solução em FORTRAN executa automaticamente uma busca dentro do comando READ, no qual o número do registro da entrada procurado é especificado para facilitar a busca, ou seja, a posição do registro no arquivo é calculada pelo sistema como `rec*RECL`.

7.6.3 Semântica da Manipulação de Erros de E/S

A maioria dos erros em tempo de execução em situações práticas de programação ocorre durante operações de entrada e saída. A expressão muito comum "lixo para dentro, lixo

11. Lembre-se de que a indexação de matrizes em FORTRAN começa em 1 (em vez de em 0), de modo que o cálculo do número relativo do registro é ligeiramente diferente daquele da Figura 7.6.

para fora" (a abreviação em inglês é GIGO) foi cunhada para denotar que, se um programa tiver recebido entrada de dados ruins, provavelmente terá produzido resultados ruins. Todavia, linguagens mais antigas forneciam poucas ferramentas de programação para capturar erros de E/S. Linguagens mais recentes incorporam ferramentas sofisticadas para suportar a detecção e a correção de erros de E/S durante a execução. Assim, a expressão GIGO tem menos probabilidade de caracterizar programas escritos em linguagens modernas, assumindo que esses programas utilizam efetivamente os dispositivos de manipulação de erros disponíveis.

Nesta seção, resumimos os recursos de manipulação de erros de E/S de versões recentes de Fortran, C++ e Ada. Esse tratamento fornece uma introdução conveniente para a Seção 7.7, que cobre o tópico mais geral de semântica de manipulação de erros.

A Figura 7.7 mostra um pequeno programa em Fortran que tenta abrir e ler um valor inteiro de um arquivo e imprime a saída dessa operação na tela. O veículo de Fortran para a manipulação de erros de E/S é a cláusula IOSTAT, em que uma variável inteira é designada para monitorar o status de uma operação OPEN, READ ou WRITE. Quando usado com um comando OPEN, o valor 0 significa que o arquivo foi aberto com sucesso; caso contrário, um inteiro positivo é atribuído a ela.

Quando um comando READ ou WRITE encontra erros de formatação, condições de final de arquivo ou de final de registro, o programa abortará, a menos que uma cláusula IOSTAT esteja presente. Nesse caso, a variável IOSTAT recebe um valor negativo, se o final de um arquivo for encontrado, um valor positivo, se ocorrer um erro de entrada, ou zero, se não ocorrer nem o final de arquivo nem o erro de entrada.

A partir de Fortran, projetos de linguagens mais recentes suportam a manipulação de erros de E/S por meio do uso de mecanismos de "exceção". Esses mecanismos suportam não apenas a manipulação de eventos inesperados de E/S como também a manipulação de uma diversidade de outros eventos inesperados em tempo de execução. Tratamos do uso da manipulação de exceções para erros de E/S no restante desta seção, deixando um tratamento mais amplo da manipulação de exceções para outros tipos de eventos na Seção 7.7.

```
PROGRAM main
 INTEGER :: error
 OPEN(UNIT=7,FILE='myData',STATUS='OLD',IOSTAT=ioerror)
 IF (ioerror==0) THEN
   WRITE (*, '(1X,A)') 'opened without errors'
   READ (7, *, IOSTAT=ioerror) InValue
   IF (error < 0)
      WRITE (*, '(1X,A,I3)') 'End of file : ',ioerror
   ELSE IF (ioerror > 0)
      WRITE (*, '(1X,A,I3)') 'error on READ: ',ioerror
   ELSE WRITE (*, '(1X, I3,A)') InValue,' was read'
   END IF
  CLOSE (UNIT=7)
 ELSE IF (ioerror > 0) THEN
   WRITE (*, '(1X,A,I4,A)') 'error on OPEN ',ioerror, &
         ': file does not exist'
 END IF
END PROGRAM main
```

| Figura 7.7 Exemplo de Manipulação de Erros de E/S em Fortran

| **Tabela 7.4** Exceções de E/S Predefinidas em Ada

Exceção	Significado
Status_Error	Tentativa de leitura ou gravação em um arquivo que não esteja aberto
Mode_Error	Tentativa de leitura de um arquivo de saída ou gravação em um arquivo de entrada
Name_Error	Uso de um nome de arquivo inapropriado em um comando de abertura
Use_Error	Tentativa de enviar a saída para um dispositivo de entrada (por exemplo, o teclado)
Device_Error	Mal funcionamento de hardware
End_Error	Tentativa de leitura após o final do arquivo
Data_Error	O valor lido é incompatível com o tipo da variável-alvo
Layout_Error	Tentativa de referenciar o número da coluna ou da linha além do limite

C++, Ada e Java têm mecanismos abrangentes de manipulação de exceções, enquanto Fortran, Pascal e C não têm. O mecanismo de manipulação de exceções de C++ preenche a lacuna entre uma definição de classe e seu cliente, ou seja, embora a classe contenha informações detalhadas sobre quais tipos de erros podem ocorrer, ela não sabe onde tais erros podem ocorrer – apenas o programa cliente o sabe.

Ada suporta uma quantidade de exceções predefinidas de E/S e outros tipos. Da mesma forma que C++, Ada também permite aos programadores definir suas próprias exceções. Entretanto, Ada não manipula exceções por padrão, de forma que, caso uma exceção seja gerada, o programa terminará e exibirá informações sobre o tipo e a localização da exceção.

A Tabela 7.4 lista as exceções predefinidas em Ada, com uma breve descrição de cada. Por exemplo, o código de manipulação de exceções em Ada da Figura 7.8 simula um pouco do comportamento do código em Fortran da Figura 7.7.

Esses exemplos em C++ e Ada ilustram o progresso que ocorreu na manipulação de erros de E/S desde o projeto de Fortran. Especificamente, a manipulação de erros por meio do mecanismo de exceções de uma linguagem pode resultar em um código mais limpo e um conjunto mais poderoso de alternativas para o programa responder a erros de E/S. Essa flexibilidade se estende à questão mais geral de manipulação de exceções que ocorrem em outras partes do programa além da E/S, conforme discutimos na Seção 7.7.

7.7 SEMÂNTICA DE MANIPULAÇÃO DE EXCEÇÕES

A robustez tem se tornado criticamente importante à medida que os computadores têm sido embarcados em dispositivos de tempo real como motores de automóveis, sistemas de controle de aeronaves e assim por diante.

> **Definição:** Aplicações são *robustas* quando continuam a operar sob todas as situações de erro presumíveis.

Um computador que auxilia na operação de um automóvel ou no vôo de uma aeronave não pode parar simplesmente porque ocorreu uma condição excepcional.

A manipulação de exceções fornece um meio para um programa lidar com erros inesperados em tempo de execução, E/S e outros tipos. Geralmente é essencial para um programa poder responder a erros em tempo de execução quando eles ocorrem, em vez de deixá-los causar o encerramento do programa prematuramente.

> **Definição:** Uma *exceção* é uma condição de erro que ocorre a partir de uma operação que não pode ser resolvida por si só.

```
with Ada.Text_IO, Ada.Integer_IO;
use Ada.Text_IO, Ada.Integer_IO;
procedure main is
begin
   error: Integer;
   inValue: Integer;
   myData: File_Type;
   Open (Unit7, In_File, "myData");
   Get (Unit7, inValue);
   Put ("Opened file without errors");
   New_Line;
   Put ("Value read is: "); Put (inValue);
 exception
   when Name_error =>
      Put ("Error on open: improper file name");
   when Data_error =>
      Put ("Error on read: invalid integer");
 end;
end main;
```

| Figura 7.8 Exemplo de Manipulação de Erro de E/S em Ada

Exceções podem ocorrer em vários níveis de abstração. No nível de hardware, exceções incluem operações ilegais como divisão por zero e referências ilegais à memória como falhas de segmentação e erros de barramento. No nível de linguagem de programação, exceções podem ser causadas por eventos como um índice de matriz fora dos limites, uma tentativa de leitura de um valor do tipo errado ou uma tentativa de acesso a um objeto na pilha usando um ponteiro nulo. Exceções de níveis mais altos também são úteis para tipos específicos de estruturas de dados, como a tentativa de execução de uma operação de desempilhamento de uma pilha vazia.

Durante muito tempo, nenhuma linguagem com amplo uso forneceu um mecanismo de exceções verdadeiramente geral e usável. Algumas linguagens (como Cobol e PL/I) forneciam recursos limitados de exceção. O dispositivo de exceções de Cobol era bastante limitado, mas PL/I estendeu bastante a idéia de manipulação de exceções.

PL/I forneceu uma longa lista de exceções predefinidas, assim como a capacidade de um programa definir exceções específicas de aplicações. Os projetistas de Ada estenderam a abordagem de PL/I, ao mesmo tempo em que incorporaram as idéias de manipulação de exceções das linguagens experimentais CLU (Liskov e Snyder, 1979; Liskov e outros, 1981) e Mesa (Mitchell et al., 1979). Linguagens modernas (como C++, Ada e Java) agora incorporam um extenso mecanismo de exceções para facilitar aplicações robustas.

Gerar uma exceção é sinalizar que a condição que ela representa ocorreu. No caso da divisão por zero, a exceção é gerada pelo hardware. *Capturar uma exceção* significa transferir o controle para um manipulador de exceções, o qual define a resposta que o programa recebe quando a exceção ocorre. No caso de divisão por zero, o controle pode ser transferido pela interrupção de hardware para uma rotina de manipulação de interrupções, que poderia exibir uma mensagem e terminar o programa.

Embora exista um modelo semântico formal de exceções baseado em semântica denotacional (Alves-Foss (ed.), 1999), é matematicamente muito complexo para ser útil neste estudo. Na Seção 7.7.1, damos um tratamento mais conceitual de semântica de manipulação de exceções, tanto em linguagens de programação que não possuem recursos de exceções gerais quanto naquelas que os têm.

7.7.1 Estratégias e Questões de Projeto

Como os programadores lidam com situações excepcionais em linguagens como Pascal, C e Fortran, que não possuem um mecanismo de exceções? Geralmente eles inventam uma convenção para lidar com a exceção que permite ao programa continuar de forma elegante quando ela ocorre.

Outra técnica comum é definir uma função que retorne um valor incomum ou ilegal quando ocorrer uma falha em tempo de execução. Por exemplo, durante a pesquisa de determinado valor em uma matriz, se esse valor for encontrado, geralmente o índice deste será retornado. Contudo, se o valor não estiver na matriz, então (no caso de matrizes indexadas em 0) o valor -1 pode ser retornado, o que indica que o valor não foi encontrado.

Entretanto, um mecanismo definido pelo programador, muitas vezes, é um substituto fraco para um mecanismo de exceções interno. Analise, por exemplo, o seguinte fragmento de código simples em Pascal que lê uma série de números de um arquivo e calcula sua soma:

```
sum := 0.0;
while not eof(file) do
begin
   read(file, number);
   sum := sum + number;
end;
```

Diversas exceções possíveis impedem que esse código seja robusto. Primeiro, se houver espaço em branco após o último número, o teste de final de arquivo retornará *falso*, mas o `read` pulará esse espaço em branco e gerará um erro fatal quando o final do arquivo for atingido. Outro tipo de exceção pode ocorrer se um valor de entrada não for um número válido; por exemplo, se o valor contiver um caractere que não for dígito. Isso também gera um erro fatal. Esses tipos de erros são difíceis de contornar via programação.

Linguagens modernas incorporam estratégias em nível de programação nas suas bibliotecas de classe e de função. Por exemplo, quando o método Java `indexOf` é usado para pesquisar um Vetor, retorna –1 quando a pesquisa é malsucedida.

Definições similares ocorrem em muitas outras bibliotecas de funções Java. Ao recuperar um valor associado a determinada chave em uma tabela *hash*, Java retorna um objeto `null` se a chave não for encontrada. De forma semelhante, ao ler linhas de um `BufferedReader` usando o método `readLine`, um objeto `null` é retornado quando o final do arquivo for atingido. Essa abordagem também é usada nas bibliotecas C padrão.

Outra abordagem à programação de exceções é adicionar um parâmetro de erro (ou retornar um código de erro para uma função `void`). Um exemplo do Turbo Pascal é o da função `val`, que converte uma representação em *string* de um número para binário:

```
val(numberString, int, error);
```

Se a variável `numberString` contiver o valor '*123*' antes da chamada a `val`, então o *int* resultante possui o valor 123, e `error` possui o valor 0. Se, em vez disso, `numberString` contiver o valor '*1x3*', então `int` é indefinido e `error` possui o valor 2, que é a posição na *string* em que o caractere ilegal aparece.

Se tais situações são verdadeiramente exceções ou apenas parte do processamento normal de tabelas e arquivos é uma questão a ser debatida. Em Java, essas situações não são consideradas exceções, enquanto em Ada, por exemplo, espera-se continuar a ler até que seja encontrado um final de arquivo para disparar essa exceção.

Uma explicação possível para a diferença nas filosofias de manipulação de exceções entre Ada e Java é que, em Java, as exceções são objetos. Assim, criar uma exceção envolve uma alocação de *heap* que deve acabar sofrendo a coleta de lixo após a exceção ter sido manipulada. Entretanto, uma exceção em Ada é um tipo primitivo interno da linguagem, de modo que o custo de gerar e manipular exceções é menor do que em Java.

Quando uma exceção é disparada, o fluxo de controle é transferido diretamente para um mecanismo de manipulação de exceções. Duas questões importantes surgem ao se projetar um mecanismo efetivo de manipulação de exceções:

1. Como cada manipulador é associado a cada exceção possível de ser gerada?
2. Um manipulador de exceção deve continuar a executar o código que gera essa exceção?

Para abordar a primeira questão, exceções em nível de hardware são definidas para corresponder a interrupções específicas de hardware. Existe basicamente um manipulador de interrupções distinto para cada tipo de interrupção. Em uma aplicação grande, tal organização é menos útil do que permitir que diferentes partes de uma aplicação respondam diferentemente a determinada exceção quando ela ocorrer.

Ada, C++ e Java permitem que um manipulador de exceções seja associado a um bloco de código. Tal bloco pode ser um método ou uma função inteira ou simplesmente um grupo de comandos dentro de uma construção maior, como uma seqüência de comandos, uma condição ou um laço.

Se nenhum manipulador de exceções aparecer dentro da função em que a exceção for gerada, a pesquisa de um manipulador será propagada pela pilha de chamadas até a rotina que a chamou. Se a chamada ao método ou à função estiver dentro de um bloco com um manipulador para essa exceção, então ela é capturada aqui. Caso contrário, a pesquisa continua pela pilha de chamadas até o primeiro manipulador de exceções apropriado.

Ao abordar a segunda questão, duas abordagens alternativas foram usadas, denominadas modelo de *prosseguimento* e modelo de *finalização*, e estão resumidas de forma ilustrada na Figura 7.9 (usando sintaxe Java).

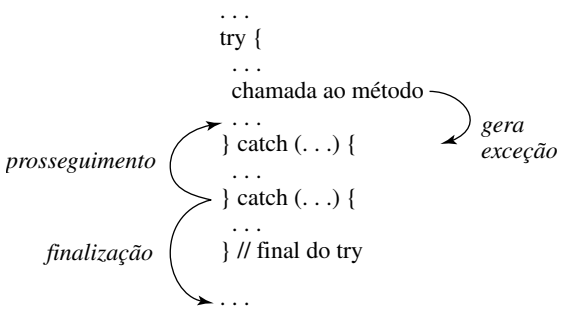


Figura 7.9
Modelos de Manipulação de Exceções – Prosseguimento *versus* Finalização

Manipuladores de interrupção em um sistema operacional implementam o modelo de recomeço, ou seja, após lidar com a interrupção, o manipulador fica livre para prosseguir com o código que gerou a exceção. PL/I usa o modelo de prosseguimento. Mecanismos de manipulação de exceções em Ada, C++ e Java, entretanto, usam o modelo de finalização; o código que sinaliza a exceção não pode ser continuado.

Programadores obtêm o efeito do modelo de finalização executando um comando *return* ou *goto* não-local dentro do manipulador de exceções. De forma semelhante, a programação cuidadosa em Ada, C++ e Java pode ser usada para simular o modelo de prosseguimento.

De modo geral, o modelo de prosseguimento é mais complexo de se implementar, já que deve restaurar o estado inteiro do programa antes de a exceção ter sido gerada. Isso pode incluir voltar pela pilha de tempo de execução e restaurar os valores de determinadas variáveis. O modelo de finalização, é claro, não tem tanta complexidade.

7.7.2 Manipulação de Exceções em Ada, C++ e Java

Um programa em Ada pode conter código que responda a exceções quando elas ocorrerem. Quando uma exceção ocorre, o sistema de tempo de execução procura um manipulador de exceções que corresponda à exceção gerada. Se ele não encontrar um, a rotina corrente termina e passa a exceção para a rotina que a chamou. Esse processo continua até que encontre um manipulador que lide com a exceção ou termine a rotina principal, em cujo caso o programa termina.

Se o sistema encontrar um manipulador para a exceção, ele executa o código no manipulador e então termina a rotina corrente. Se uma exceção for gerada na parte declarativa de uma rotina, Ada abandona os processamentos posteriores dessa declaração. O mesmo acontece com exceções geradas dentro de um manipulador de exceções, eliminando, assim, o perigo de laços infinitos na manipulação de exceções.

Um programa em Ada pode gerar uma exceção explicitamente com o comando `raise`. Esta exceção pode ser uma exceção predefinida em Ada ou definida no próprio programa. Todavia, já que exceções em Ada são definidas de forma geral, uma exceção mais específica definida pelo programa, muitas vezes, é mais útil. Dentro de um manipulador, a exceção `raise with no condition` pode ser empregada para propagar a exceção para um nível mais alto na seqüência de chamadas. Isso pode ser útil para assegurar a recuperação elegante em todos os níveis, quando uma exceção for fatal para o programa.

Exceções em C++ e Java são definidas como classes. Essas duas linguagens contêm dispositivos semelhantes para definir, gerar e capturar uma exceção.

C++ usa as seguintes palavras-chave para manipulação de exceções: `try`, `throw` e `catch`. A sintaxe de um comando de manipulação de exceções é:

```
try {
  // código que pode gerar uma exceção
}
catch ( // tipo da exceção)
{
  //código a ser executado quando esse tipo de exceção ocorrer
}
```

A semântica de um manipulador de exceções em C++ é bastante direta. Primeiro, o código dentro do bloco `try` é executado normalmente. Se nenhuma exceção ocorrer, o bloco `catch` é pulado. Se uma exceção ocorrer (por exemplo, se uma função `throw` for executada para essa exceção), o código do bloco `catch` é executado imediatamente e o controle não retorna para o comando em que a exceção foi gerada.

A seguir está um exemplo em C++, em que o programa pode capturar uma instância de um índice de matriz fora dos limites declarados e exibir uma mensagem quando isso ocorrer.

```
#include <iostream.h>
int main () {
  char A[10];
  cin » n;
  try
  {
   for (int i=0; i<n; i++)
   {
     if (i>9) throw “array index out of range”;
     A[i]=getchar();
   }
  }
  catch (char * s)
  {
   cout << “Exception: ” << s << endl;
  }
  return 0;
}
```

Assim que a exceção é capturada, o programa executa o bloco do *catch* e retorna 0. De modo geral, o parâmetro para `catch` pode ser de qualquer tipo válido. Além disso, `catch` pode ser sobrecarregado de modo que possa receber diferentes tipos como parâmetro. Neste caso, o bloco `catch` executado é o que corresponde à exceção gerada. Se uma exceção não for capturada por nenhum comando `catch`, o programa terminará e exibirá uma mensagem de erro de “Término anormal”.

A biblioteca padrão de C++ predefine algumas exceções que podem ser capturadas quando ocorrerem dentro de um bloco `try`. As exceções a seguir são subclasses da classe `std::exception`:

Exceção C++	Gerada Quando:
`bad_alloc`	O armazenamento no heap não puder ser alocado para um comando `new`
`bad_cast`	Uma conversão não puder ser completada
`bad_exception`	Uma exceção não corresponde a algum catch
`bad_typeid`	Um tipo não pode ser determinado por `typeid`
`logic_error`	Um erro de programação (por exemplo, índice fora do limite) ocorre
`runtime_error`	Uma circunstância fora do controle do programa ocorre

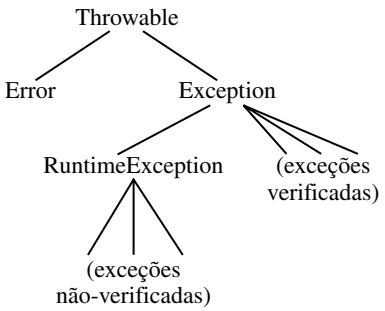


Figura 7.10
Hierarquia de Classes de Exceção em Java

Já que Java é uma linguagem mais nova do que C++, seu suporte a exceções é mais bem desenvolvido. A Figura 7.10 mostra a classe *Exception* de Java e algumas de suas classes relacionadas.

Algumas exceções em Java são predefinidas dentro da hierarquia de classes. Todas as exceções em Java são subclasses de `Throwable` e estão divididas em três categorias:

1. Erros na máquina virtual (`Error` e suas subclasses);
2. Exceções em tempo de execução (`RuntimeException` e suas subclasses), como o acesso a um objeto nulo ou um índice ilegal;
3. Todas as outras.

As duas últimas categorias são subclasses da classe `Exception` e, por convenção, têm a palavra *Exception* como parte do seu nome. De forma semelhante, erros na máquina virtual, por convenção, têm a palavra *error* como parte do seu nome.[12]

As exceções, em contraste com os erros na máquina virtual, estão divididas em *verificadas* e *não-verificadas*. Formalmente, uma exceção verificada é uma instância da classe `Exception` ou uma subclasse que não seja instância de `RuntimeException`. Exceções verificadas são exceções em nível de aplicação. As bibliotecas de classe Java contêm uma ampla variedade de exceções, incluindo exceções de E/S, exceções de rede e assim por diante.

Uma exceção não-verificada é uma instância da classe `RuntimeException` ou uma de suas subclasses. Java requer que exceções verificadas sejam capturadas por um manipulador de exceções, enquanto um manipulador de exceções é opcional para uma exceção não-verificada. Exemplos de exceções não-verificadas incluem índices fora de limites de matrizes ou sub*strings*, acesso por meio de um objeto nulo e divisão por zero.

Uma exceção definida pelo programador em Java é um objeto da classe `Exception` ou uma de suas subclasses; assim, uma exceção definida pelo programador pode ser verificada ou não-verificada. Uma exceção pode conter uma mensagem arbitrária que tenha informações. Assim, a classe `Exception` possui dois construtores, cujas assinaturas são:

```
public Exception(  );
public Exception(String s);
```

Assim, cria-se uma nova exceção por meio de uma subclasse de `Exception` ou de uma de suas subclasses, conforme mostrado a seguir.

12. Exceções em Java do tipo `Error` geralmente não são a principal preocupação do programador; elas incluem erros de máquina virtual tais como falta de memória, *overflow* de pilha de tempo de execução e assim por diante.

```
class StackUnderflowException extends Exception {
   public StackUnderflowException() {super(); }
   public StackUnderflowException(String s) {super(s); }
}
```

Um uso típico de uma exceção definida pelo programador seria:

```
if (stack == null)
   throw new StackUnderflowException();
```

Objetos que constituem exceções (em oposição a subclasses) raramente recebem nomes, já que seu único propósito é serem gerados. Subclasses de `Exception` podem ter informações adicionais ou fornecer comportamento especializado. Todavia, na maioria dos casos, elas existem apenas para fornecer novas subclasses de exceções. A subclasse mais comum é a `IOException` e suas subclasses no pacote `java.io`.

Um método em Java é necessário para especificar por meio de uma cláusula `throws` no seu cabeçalho qualquer exceção verificada que não capture a si própria. Um exemplo é o método `readLine` de um BufferedReader:

```
public String readLine( ) throws IOException;
```

Por meio disso, o compilador Java pode assegurar que cada exceção verificada seja capturada e manipulada pela aplicação.

Exceções são capturadas de duas formas. Uma chamada a um método que potencialmente gere uma exceção deve aparecer em um comando `try` ou o método que contenha a chamada deve declarar quais exceções ele gera. Analise o código a seguir que, dado um *filename*, tenta abrir esse arquivo para entrada:

```
try { BufferedReader infile = new BufferedReader(
                              new FileReader(filename));
    ... // processar arquivo
} catch (FileNotFoundException e) {
    System.out.printIn(“File not found: ” + filename);
    System.exit(1);
} // try
```

Um comando `catch` em Java é semelhante a um cabeçalho de método, pois seu argumento formal especifica o tipo (ou supertipo) da exceção que ele está capturando, que pode ser usado como qualquer outro objeto.[13] Nesse caso, se a exceção for gerada, uma mensagem será impressa e a aplicação será concluída por meio de uma saída diferente de zero (erro). Se não houver saída, o fluxo de controle continuará com o próximo comando após o `try`; ou seja, Java usará o modelo de finalização para manipular exceções. A forma geral de um comando `try` em Java é:

13. Essa semelhança é limitada, entretanto, já que um `catch` não possui tipo de retorno ou modificadores de visibilidade.

```
try {
    // código que pode dispar um ou mais tipos de exceções
    ...
} catch (Exception type e) {
    ...
} catch (Exception type e) {
    ...
} finally
    ...
}
```

Se uma exceção for gerada, cada cláusula `catch` é avaliada em ordem até que ocorra uma correspondência. A seguir, o bloco associado à cláusula `catch` é executado. Cada `try` pode ter tantas cláusulas `catch` quanto desejado. Observe que um único `try` pode ser usado para capturar múltiplas exceções por meio de um grupo de comandos, desde que cada possível exceção verificada seja capturada. Uma forma simples de simular um `catch` genérico para cada exceção verificada é usar `catch (Exceção)` como a última (ou única!) cláusula `catch`.

A cláusula `finally`, se estiver presente, sempre será executada, seja uma exceção gerada ou não. Não é possível evitar a execução do código na cláusula `finally`. Essa cláusula, muitas vezes, é usada para fechar arquivos que haviam sido abertos e processados dentro do seu comando `try`.

A outra alternativa para se usar um `try` em chamadas a métodos que geram exceções é fazer o próprio método que a contém propagar a exceção por meio de uma cláusula `throws`:

```
public void readFile (BufferedReader infile)
                      throws IOException
   String line;
   while ((line = infile.readLine()) != null) {
       ... // processar linha
   }
}
```

Essa rotina lê linhas por meio do método `readLine` da classe `BufferedReader`, que pode gerar uma `IOException`. Como a chamada a `readLine` não aparece em um comando `try`, o método `readFile` deve declarar que não trata a exceção.

Observe que, no final do arquivo, o método `readLine` retorna um valor nulo, em vez de gerar uma exceção de final de arquivo. Em contraste, outros métodos de leitura em Java geram uma exceção no final do arquivo, assim como Ada.

Exceções em Java são executadas pelo comando *throw*:

```
throw new Exception(  );
```

Neste ponto, o fluxo normal de controle é concluído, e o controle é transferido para o `catch` correspondente do `try` mais próximo. Se não houver um, o método chamado propaga a exceção e o processo continua pela pilha de tempo de execução (para obter mais detalhes sobre a dinâmica da pilha de tempo de execução, veja o Capítulo 10).

Observe que a exceção gerada anteriormente é anônima, no sentido de que nenhuma referência a ela é guardada pelo nome. Já que o fluxo de controle não continua após esse ponto, geralmente não serve a nenhum propósito manter uma referência à exceção.

Normalmente, a exceção gerada seria uma das subclasses de `Exception`, talvez até mesmo com uma mensagem anexada:

```
throw new StackUnderflowException("pop on an empty stack");
```

O principal propósito de se criar novas exceções é dar um nome a uma categoria que seja significativa para a aplicação e agrupar exceções por categoria. Um exemplo dessa última é `IOException`, que é a classe-mãe de todas as exceções de entrada/saída.

Exemplo: Manipulando a Entrada de Forma Robusta Exceções que causam o término de um processo de entrada em Java ocorrem com freqüência em aplicações que usam um argumento de linha de comando para identificar um arquivo de entrada.

Um método `main` típico para manipular tal cenário é mostrado na Figura 7.11. Quando o usuário fornece um nome de arquivo válido em tempo de execução, esse arquivo é aberto e processado normalmente. Todavia, diversos outros eventos podem ocorrer. Por exemplo, o usuário poderia esquecer de fornecer um argumento ou então poderia fornecer um argumento que fosse um nome de arquivo inválido ou o nome de um arquivo que não pudesse ser lido pela aplicação. O programa é projetado para responder a qualquer um desses eventos.

Se nenhum argumento for fornecido na linha de comando, então a referência a `arg[0]` gera uma `ArrayIndexOutOfBoundsException`, já que o índice 0 excede o tamanho real da matriz. Normalmente, a indexação não precisa aparecer em um comando `try`; fazer isso, nesse caso, permite que a aplicação forneça uma mensagem de erro específica da aplicação, em vez da mensagem de erro da linguagem Java. Essa última seria presumivelmente ininteligível para o usuário.

Em resposta, o `catch` captura uma `ArrayIndexOutOfBoundsException`, que é a classe-mãe da exceção realmente gerada. Além da relativa concisão do nome da classe-mãe, parece haver pouca razão, nesse caso, para se preferir uma em relação à outra. Um uso desse

```
public static void main(String[] arg) {
  BufferedReader rdr = null;
  try {
    rdr = new BufferedReader(new FileReader(arg[0]));
  } catch (IndexOutOfBoundsException e) {
    System.err.printIn("Missing argument");
    displayUsage( );
    System.exit(1);
  } catch (FileNotFoundException e) {
    System.err.printIn("Cannot open file: " + arg[0]);
    System.exit(1);
  }
  process(rdr);
}
```

| Figura 7.11 Manipulação Robusta de Entradas em Java

recurso se aplica para listar todas as exceções específicas primeiro e depois usar uma classe-mãe (por exemplo, `Exception`) efetivamente para capturar todas as outras exceções.

Outro evento que pode ocorrer é o arquivo não poder ser aberto para leitura. Isso pode ser causado por diversos tipos de situações, conforme explicado anteriormente. Em qualquer um desses casos, a classe `FileReader` gera uma `FileNotFoundException`. Nesse caso, a exceção é capturada no método `main`, e uma mensagem de erro apropriada é exibida. Observe que, nesse exemplo, cada `catch` termina com uma saída de erro diferente de zero.

A solução dada anteriormente possui outro aspecto notável, ou seja, a chamada ao método `process` não pode ser incluída no bloco `try`. Se fosse, qualquer exceção de limite de índice não capturada no método `process` resultaria na mensagem "Argumento ausente", o que seria enganoso. Manter a chamada a `process` fora do bloco `try-catch` requer declarar a variável `BufferedReader` fora do bloco `try-catch` e inicializá-la explicitamente como `null`.

Exemplo: Verificando Entrada Inválida pelo Teclado Neste exemplo, exploramos como o modelo de prosseguimento de manipulação de exceções pode ser simulado em Java com o uso de um laço. Esse exemplo tipifica aplicações que solicitam ao usuário uma entrada, e ocorre um erro de teclado. Uma resposta apropriada simples é notificar o usuário do erro e solicitar que ele tente novamente.

Antes da versão 1.5, Java não fornecia um método *read* sobrecarregado para os diferentes tipos primitivos de dados, de modo que fossem convertidos da representação em texto para, por exemplo, um inteiro em sua representação interna (binário de 32 bits para inteiros), como o que ocorre para `print`.[14] Em vez disso, uma representação em *string* do número é lida e convertida para (por exemplo) um `int`. Supondo que um objeto `BufferedReader in` tenha sido criado com o uso do código:

```
BufferedReader in = new BufferedReader(
                    new InputStreamReader(System.in));
```

então o código necessário é apresentado na Figura 7.12.

```
while (true) {
  try {
    System.out.print ("Enter number : ");
    number = Integer.parseInt(in.readLine ( ));
    break;
  } catch (NumberFormatException e) {
    System.out.printIn ("Invalid number, please reenter.");
  } catch (IOException e) {
    System.out.printIn("Input error, please reenter.");
  } // try
} // while
```

| Figura 7.12 Exemplo do Modelo de Prosseguimento

14. Lembre-se da discussão anterior sobre a E/S formatada em Java 1.5.

Se um número válido for digitado, nenhuma exceção será gerada e o laço será concluído normalmente. Contudo, se um número inválido for digitado, uma exceção será gerada, uma mensagem de erro será impressa e o laço será repetido, solicitando novamente um número ao usuário. Nesse código, o método `readLine` pode gerar uma `IOException`, que deve ser capturada pelo comando `try`.

O exemplo de entrada inválida usa um laço potencialmente infinito para contornar o modelo de finalização usado por Java para manipular exceções. O laço aqui só pode ser concluído se nenhuma exceção for gerada. Assim, o uso do laço permite ao programador obter efetivamente um modelo de prosseguimento de manipulação de exceções.

Exemplo: Definindo uma Exceção Específica de Aplicação Ao desenvolver um método ou uma classe reusável, pode haver erros que o desenvolvedor da classe gostaria que o programa cliente manipulasse. A maioria das linguagens que fornecem manipulação de exceções permite ao programador criar tais exceções específicas para a aplicação e gerá-las sob as condições apropriadas. Estas geralmente representam condições das quais o programa cliente pode querer se recuperar.

Um exemplo típico ocorre no desenvolvimento de uma classe de pilha. Suponha que temos a classe `StackUnderflowException` apresentada no início da Figura 7.13. Quando a pilha estiver vazia, uma operação `pop` não pode ser definida. Assim, o código típico para `pop` em uma classe `Stack` de `int`s usando uma implementação de matriz é apresentada no restante da Figura 7.13.

O código declara a matriz `stack` com alguma capacidade máxima; a variável `top` registra o topo corrente da pilha. `StackUnderflowException` é declarada como uma subclasse da classe `Exception` e se baseia nessa classe-mãe para seus métodos.

O método `pop` é simples: primeiro, ele decrementa `top` e então retorna o valor de `stack[top]` usando o valor decrementado de `top`. Entretanto, essa operação não é segura; se o valor original de `top` for menor ou igual a zero, uma `ArrayIndexOutOfBoundsException` será gerada.

```
class StackUnderflowException extends Exception {
  public StackUnderflowException( ) { super(); }
  public StackUnderflowException(String s){ super(s);}
}
...
class Stack {
  private int stack[];
  private int top = 0;
  ...
  public int pop() throws StackUnderflowException {
    if (top <= 0)
        throw new StackUnderflowException("pop on empty stack");
    return stack[--top];
  }
  ...
}
```

| Figura 7.13 Definindo e Usando uma Exceção Específica de Aplicação

Entretanto, quem chama esse método não deve ter de lidar com a exceção. Assim, em vez disso, uma `StackUnderflowException` é criada e gerada, conforme necessário, pelo método `pop`. O cabeçalho desse método deve listar qualquer exceção que ele possa gerar em uma cláusula `throws`. Assim, qualquer chamada ao método `pop` deve capturar explicitamente essa exceção.

Como esse exemplo ilustra, criar uma nova exceção é um recurso simples, porém poderoso, em Java.

7.7.3 Exceções e Asserções

Exceções e manipuladores de erros de E/S são dois dispositivos distintos que os programadores podem usar para responder a eventos inesperados em tempo de execução. Um terceiro dispositivo, que é especialmente útil durante o desenvolvimento de programas, é a assim chamada *asserção*,[15] um predicado inserido em algum ponto de um programa que descreve um estado que as variáveis ativas devem satisfazer sempre que esse ponto for atingido durante a execução. Se esse estado não for satisfeito, a asserção é dita como tendo "falhado", e a execução pode ser interrompida. Assim, asserções são um pouco semelhantes a exceções no sentido de que podem definir e responder a eventos inesperados em tempo de execução.

A linguagem Eiffel foi a primeira a incorporar integralmente asserções. Versões posteriores de Java também incluem asserções. Nesta seção, examinamos o mecanismo de Java (adicionado na versão 1.4) para definir e manipular asserções. Também examinamos o desafio de implementar um mecanismo de asserções em versões anteriores de Java.

De modo geral, as asserções são úteis na verificação de programas porque podem representar as assim chamadas "precondições", "pós-condições" e "invariantes dos laços". Um tratamento cuidadoso no uso de asserções na verificação de programas é o foco do Capítulo 18.

Informalmente, a *precondição* de uma função é uma restrição sobre os valores dos seus argumentos que deve ser satisfeita para que a função execute sua tarefa. De forma semelhante, uma *pós-condição* é um tipo de "promessa" de que, se a chamada fornecer os argumentos que satisfaçam à precondição, o resultado satisfará a pós-condição. Um *invariante do laço* é uma expressão sobre o que deve ser verdadeiro antes (e depois) de cada repetição do corpo de um laço. Todas essas três devem ser escritas na forma de uma expressão boleana em linguagens como Java, C++ e Ada.

Por exemplo, analise o cálculo de número de Fibonacci de índice *n* mostrado na Figura 7.14 – que é uma variante em Java do programa em C++ da Figura 10.5. Esse programa lê um inteiro como um argumento de linha de comando e relata um problema, se esse inteiro for negativo. Conforme mostrado aqui, a precondição da função `fibonacci` é que o valor inteiro de `n` fornecido por qualquer chamada deve ser não-negativo, e sua pós-condição é uma promessa de retorno do número de Fibonacci de índice *n* para esse determinado valor de `n`.

A sintaxe EBNF do comando de asserção de Java é definida a seguir:

ComandoAsserção → `assert` *expressãoBoleana* [: *expressãoString*];

Aqui, a *expressãoString* opcional define a mensagem que será exibida sempre que a *expressãoBoleana* for `false` quando o *ComandoAsserção* for atingido em tempo de execução.

Semanticamente, um *comandoAsserção* interromperá a execução do programa se sua *expressãoBoleana* for `false` quando for atingido durante a execução do programa, em cujo caso sua *expressãoString* também é exibida. Se sua *expressãoBoleana* for `true`, a

15. O termo está definido formalmente no Capítulo 18.

```
public class assertTest {
   public static int fibonacci (int n) {
      int fib0, fib1, temp, k;
      // precondition: n >= 0
      assert n >= 0 : "Invalid argument for fibonacci" ;
      fib0 = 0; fib1 = 1; k = n;
      while (k > 0) {
         temp = fib0;
         fib0 = fib1;
         fib1 = fib0 + temp;
         k = k - 1;
      }
      // postcondition: fib0 is the nth Fibonacci number.
      return fib0;
   }
   public static void main (String[] args) {
      int answer;
      answer = fibonacci(Integer.parseInt(args[0]));
   }
}
```

| Figura 7.14 Programa em Java 1.4 com um *ComandoAsserção*

mensagem do comando de asserção não será impressa e a execução do programa continuará normalmente.

Em Java, um *ComandoAsserção* pode ser colocado em qualquer lugar entre os comandos executáveis de um método, mas não entre as declarações.[16] Para o nosso exemplo, o programa passará a seguinte mensagem de erro, se o argumento para `fibonacci` for negativo:

```
Exception in thread "main" java.lang.AssertionError: Invalid
          argument for fibonacci
       at assertTest.fibonacci(assertTest.java:4)
       at assertTest.main(assertTest.java:16)
```

Em versões de Java anteriores à versão 1.4, podemos implementar a verificação de asserções definindo uma classe `AssertException` (Fowler, 2000) como um tipo especial de exceção.

```
class AssertException extends RuntimeException {
   public AssertException( ) { super( ); }
   public AssertException(String s) { super(s); }
}
```

16. Para permitir asserções em um programa em Java 1.4 ou 1.5, a execução requer a opção `-enableassertions` (abreviada `-ea`). Para desabilitar a verificação em tempo de execução, é usada a opção `-disableassertions` (abreviada `-da`).

Para se evitar ter de adicionar muitos comandos try-catch, tornamos `AssertException` uma subclasse de `RuntimeException`. Logicamente, uma asserção é como um índice fora dos limites: esperamos que ela ocorra apenas se o programador tiver cometido algum erro. Assim, devemos ser capazes de fazer asserções sem precisarmos incluir comandos try-catch.

A seguir, analisamos a própria classe `Assert`. Já que não existem dados para serem armazenados dentro de um objeto asserção, optamos por tornar a classe uma simples biblioteca de métodos estáticos. Um método claramente necessário serve para declarar que determinada condição boleana é verdadeira.

O código para esse método é direto; se a condição for falsa, o método gera uma `AssertException`. Outro método útil declara que a execução não deve atingir esse ponto; a implementação sempre gera uma exceção. Finalmente, incluímos uma constante boleana que, se desejado, pode ser usada para otimizar asserções fora do código do programa cliente. Nossa classe `Assert` é apresentada na Figura 7.15.

Como exemplo do uso dessa classe, analise a classe de pilha revisada apresentada a seguir:

```
class Stack {
  int stack[];
  int top = 0;
  ...
  public boolean empty() { return top <=; }
  public int pop() {
    Assert.isTrue(!empty());
    return stack[--top];
  }
  ...
}
```

Nesta versão, a responsabilidade de verificar se uma pilha não está vazia antes de chamar uma operação `pop` é do programa que usa a classe `Stack`. Portanto, o programador meramente declara que a pilha não está vazia como parte da operação `pop`. No caso de uma exceção de tempo de execução ser disparada pela asserção, isso seria considerado uma falha no código cliente.

Observe que, já que o método `pop` gera uma `RuntimeException`, uma cláusula `throws` não é necessária no cabeçalho do método. A filosofia usada aqui é bastante

```
class Assert {
   static public final boolean ON = true;
   static public void isTrue(boolean b) {
      if (!b) {
      throw new AssertException("Assertion failed"); }
   }
   static public void shouldNeverReachHere() {
      throw new AssertException("Should never reach here");
   }
}
```

| Figura 7.15 Uma Classe *Assert*

diferente da usada anteriormente. Neste caso, o desenvolvedor da classe impõe certas restrições aos clientes da classe, e essas restrições são reforçadas por meio do uso de asserções.

Se um programa cliente da nossa classe pilha executasse logicamente a seguinte seqüência de código:

```
Stack stack = new Stack();
stack.pop();
```

uma exceção de tempo de execução seria gerada, resultando em uma mensagem como a seguinte:

```
AssertException: Assertion failed
        at Assert.isTrue(Assert.java.4)
        at Stack.pop(Stack.java.13)
        at ...
```

Testes adequados deveriam eliminar a possibilidade de tais exceções de asserção serem disparadas. Inseri-las no código serve parcialmente para documentar suposições feitas a respeito do estado de um método ou objeto. Nossa classe de asserção permite que tais suposições sejam declaradas explicitamente, auxiliando assim na detecção de erros sutis em tempo de execução.

7.8 RESUMO

Este capítulo introduz a semântica de linguagens de programação a partir de um ponto de vista informal. Uma abordagem operacional da semântica é usada, destacando os recursos semânticos de diversas linguagens populares. As semânticas de expressões, atribuições, condições, ramificações e laços são analisadas, junto àquelas de manipulação de exceções e entrada/saída.

Esse tratamento enfoca o comportamento de programas em tempo de execução. Em capítulos posteriores, analisaremos a implementação de semânticas de linguagens assim como suas definições operacionais e denotacionais.

EXERCÍCIOS

7.1 Analise a seguinte seqüência de comandos C/C++, que são sintaticamente válidos, mas que não têm interpretação semântica razoável (supondo que i e j tenham sido declarados como variáveis `int`):

```
j = 0;
i = 3/j;
for (i=1; i>1; i++)
   i--;
```

Como essas situações são manipuladas quando executadas pelo seu sistema C/C++?

7.2 Dê outros tipos de comandos C/C++ (além daqueles ilustrados na questão anterior) que sejam válidos sintaticamente, mas cujo significado não possa ser definido razoavelmente na semântica de uma linguagem de programação.

7.3 (a) Como Java define a idéia numérica de infinito? (Você deve examinar a *Java Language Specification* (Gosling et al., 1996) para obter detalhes.)
(b) Analisando as especificações na *Java Language Definition* (Arnold e Gosling, 1998), como você pode explicar em linguagem simples o significado do comando i = 3/j; para todos os valores possíveis de j, incluindo 0?

7.4 Analise a expressão `x + y/2` na linguagem C. Quantos significados diferentes essa expressão possui, dependendo dos tipos de x e y ?

7.5 Reescreva o programa de fatorial da Figura 7.2 em Java de modo que o valor inicial de `n` seja lido como entrada e o valor final de `f` seja impresso na tela. Agora execute o programa para os valores 2, 3, ... até que o valor de `f` ultrapasse o tamanho de um valor `int`. Qual o valor de `n` para que isso ocorra? Como esse erro é tratado por Java?

7.6 Para seu compilador C/C++, verifique se a conversão do inteiro 257 para um `char` resulta em um erro.

7.7 Para o seu compilador/interpretador Java, verifique se a conversão do inteiro 65.537 em um `char` resulta em um erro.

7.8 Tanto para C/C++ quanto para Java, escreva um programa que calcule 2^{30} + 65.534. A seguir, converta esse inteiro (ou `long int`) para ponto flutuante, adicionando 1,0. Depois converta o número de ponto flutuante de volta para um inteiro e subtraia 1. Você obtém de volta o mesmo inteiro com o qual começou? *Não* use ponto flutuante de precisão dupla.

7.9 Sugira uma estratégia geral pela qual as convenções de E/S formatada das diversas linguagens (Fortran, C, Ada e Java) poderiam se tornar mais unificadas, ou seja, você pode projetar uma expressão de formato generalizado para a qual uma expressão formatada em cada uma dessas linguagens poderia ser mapeada de forma que preservasse seu significado? Quais são os desafios para a criação de tal mapeamento?

7.10 Responda à questão anterior novamente, mas para E/S não-formatada.

7.11 Analise o fragmento de programa em C da Figura 7.6, que ilustra o acesso aleatório.
(a) Esse é um código C++ válido? Por que sim ou por que não?
(b) Escreva um pequeno programa em C ou C++ que crie um arquivo de acesso aleatório `GIS.data` de tamanho razoável, que possa ser usado como entrada por esse código.
(c) Complete esse programa de forma que ele navegue pela grade de posições, começando pelo canto superior esquerdo (NOROESTE) e exiba cada par de coordenadas x-y que encontrar em cada passo.

7.12 Analise os erros de E/S em Ada resumidos na Tabela 7.4. Usando referências da Internet, determine se um conjunto semelhante de erros de E/S é definido para C. E quanto a C++?

7.13 Usando referências da Internet, examine os modelos de manipulação de erros de C++ e Ada mais detalhadamente.
(a) Como o modelo de C++ para a manipulação de exceções difere do modelo de Java discutido neste capítulo?
(b) Como o modelo de Ada para a manipulação de exceções difere do modelo de Java?

7.14 Modifique o exemplo de entrada numérica inválida da Seção 7.7.2 de modo que o usuário tenha no máximo três tentativas para informar um inteiro válido.

7.15 Incorpore o exemplo de entrada numérica inválida da Seção 7.7.2 em um programa que leia uma seqüência de números do usuário até que um número negativo seja informado, e depois calcule a soma e a

média da seqüência de números. Verifique se a aplicação se comporta corretamente na presença de entrada inválida de números.

7.16 Analise a classe *stack* da Seção 7.7.2.
(a) Estenda essa classe com um construtor apropriado e um método principal para testar a *stack*.
(b) Estenda essa classe com um método de empilhamento que gere uma exceção de *overflow* se o método for usado em uma pilha cheia.
(c) Estenda a classe de *stack* da Seção 7.7.2 com um construtor apropriado e um método principal para testar a pilha usando o comando `assert` de Java 1.4.

7.17 Estenda a classe de *stack* da Seção 7.7.2 com um método de empilhamento que gere uma exceção de *overflow* de pilha, se o método for usado em uma pilha cheia.

8 Interpretação Semântica

"Para entender um programa você deve se tornar tanto a máquina quanto o programa."

Atribuído a Alan Perlis

VISÃO GERAL DO CAPÍTULO

Este capítulo aborda tanto a semântica operacional quanto a denotacional. Ele começa descrevendo o estado de uma computação com bastante cuidado, fornecendo, assim, uma base para a discussão detalhada da semântica de Clite nas Seções 8.1 e 8.2.

As Seções 8.2 e 8.3 abordam a semântica operacional. Elas discutem os detalhes da semântica de Clite, com tipagem estática e dinâmica, respectivamente, usando um estilo informal, suplementado por partes de código em Java. Um interpretador completo para Clite fornece uma ferramenta de testes pronta para explorar em detalhes essas idéias semânticas.

A Seção 8.4, que é opcional, faz um exame mais formal de semântica denotacional sobre o mesmo tópico. Esse tópico fornece informações sobre o rigor matemático necessário para expor completamente a semântica de uma linguagem de programação completa.

8.1 TRANSFORMAÇÕES DE ESTADOS E FUNÇÕES PARCIAIS

As etapas individuais que ocorrem durante a execução de um programa podem ser modeladas como uma série de funções de transformação de estados. Lembre-se do que foi citado no Capítulo 7, que o *estado* de um programa é o conjunto de todos os objetos ativos e seus valores correntes.

> **Definição:** A *semântica denotacional* de uma linguagem define os significados de elementos abstratos dessa linguagem como um conjunto de funções de transformação de estados.

Para especificar a semântica de uma linguagem, primeiro precisamos ter como base um pequeno conjunto de entidades matemáticas e suas propriedades. Por exemplo, os números inteiros e reais, os caracteres e os boleanos e suas propriedades definem uma base para o estudo da semântica de Clite.

Essas suposições matemáticas são amenizadas pelas restrições de computadores reais, conforme discutido no Capítulo 5. Os inteiros, por exemplo, constituem um domínio semântico para linguagens de programação, mas o tipo `int` em linguagens como C não inclui a faixa completa de valores inteiros.

> **Definição:** Um *domínio de semântica* é um conjunto de valores cujas propriedades e operações são bem compreendidas independentemente, nas quais as regras que definem a semântica de uma linguagem podem ser baseadas.

O segundo elemento de que necessitamos é um modelo preciso de um estado, incluindo os valores que possam ser armazenados lá; isso foi introduzido no Capítulo 7. Para a linguagem simples Clite, cujo ambiente é estático, o estado pode ser representado como um conjunto de pares variável-valor, ou seja:

$$estado = \{\langle var_1, val_1\rangle, \langle var_2, val_2\rangle, \ldots, \langle var_m, val_m\rangle\}$$

Aqui, cada var_i denota uma variável, e cada val_i denota seu valor atribuído correntemente.

Expandiremos essa definição de estado nos Capítulos 10 e 11, quando será importante representar o ambiente de uma forma mais dinâmica. Lá, um ambiente dinâmico nos permite caracterizar precisamente as idéias de *heap* e a pilha de tempo de execução e, assim, lidar de forma eficaz com chamadas a procedimentos, criação de objetos, passagem de parâmetros, recursão e assim por diante.

O terceiro elemento de que precisamos é a idéia de que as funções de transformação de estados na definição de semântica são necessariamente *funções parciais*.

> **Definição:** Uma *função parcial* é uma função que não está bem definida para todos os valores possíveis de seu domínio (estado de entrada).

Isso significa que determinadas construções de programas em determinados estados não podem ter representações de significados, embora essas construções sejam válidas quanto à sua sintaxe e às suas características de tipos.

Por exemplo, considere o seguinte trecho (traiçoeiro) de programa em Clite:

```
i = 1;
while (i > 0)
  ;
```

Supondo que `i` seja uma variável `int`, esse trecho não tem estado final, já que o valor de `i` nunca muda e o laço se repete infinitamente.

O significado de uma *Expressão* Clite é também uma função parcial. Por exemplo, a divisão por zero é indefinida para muitos computadores e muitas linguagens de programação. De forma semelhante, adicionar 1 ao maior valor `int` de uma máquina não pode ser definido semanticamente, já que o resultado é um valor fora do domínio semântico da máquina.

8.2 A SEMÂNTICA DE CLITE

Com essa introdução, podemos definir o significado de um programa abstrato em Clite[1] como um conjunto de funções *S* (de significado) que transformam o estado do programa. Essas funções definem individualmente o significado de *Programa* e cada tipo de comando que ocorre na sintaxe abstrata: *Salto, Bloco, Condicional, Laço, Atribuição*.

Essas funções são implementadas como uma classe `Semantics` de Java que, combinada com a classe `AbstractSintax`, forma um interpretador para Clite. Tal interpretador é valioso para o projetista de linguagem porque:

- Pode ser usado para testar a validade das definições de semântica;
- Pode revelar as vantagens e as desvantagens que ocorrem entre definições alternativas de semântica.

Entretanto, um interpretador Java para Clite, para ser completamente válido, deve supor que o próprio Java tenha sido completamente definido. De fato, este é quase o caso – uma definição formal de Java foi proposta em artigos de pesquisas recentes. Leitores interessados devem se reportar a Alves-Foss (ed.), 1999, para obter mais informações.

8.2.1 O Significado de um Programa

Suponha que *Estado* represente o conjunto de todos os estados do programa. Então o significado *S* de um *Programa* abstrato em Clite é definido pelas três funções a seguir:

$$S\text{: } Programa \rightarrow Estado$$

$$S\text{: } Comando \times Estado \rightarrow Estado$$

$$S\text{: } Expressão \times Estado \rightarrow Valor$$

Isso quer dizer que o significado de um *Programa* é uma função que produz um *Estado*. De forma semelhante, o significado de um *Comando* é uma função que, dado um *Estado* corrente, produz um novo *Estado.* Finalmente, o significado de uma *Expressão* é uma função que, dado um *Estado* corrente, produz um *Valor*.

Observe que o significado de uma *Expressão* é normalmente um *Valor*. Isso faz sentido intuitivamente, já que uma expressão é normalmente um *Valor* constante (por exemplo, 1), uma *Variável* (por exemplo, `n`), um *Binário* (por exemplo, `i + 1`), um *Unário* (por exemplo, `-x`) ou alguma combinação deles. O *Valor* de uma *Variável* é seu valor no estado corrente.

1. A semântica de matrizes em Clite é omitida dessa discussão, já que matrizes requerem um modelo dinâmico de memória (veja o Capítulo 11).

Já que a sintaxe abstrata de um programa em Clite é uma estrutura de árvore cuja raiz é o elemento abstrato *Programa*, o significado de um programa em Clite pode ser definido pela aplicação de uma série de funções à árvore. Lembre-se do que foi citado no Capítulo 2, que a sintaxe abstrata de um *Programa* é:

Programa = *Declarações* `partedec`; *Bloco* `corpo`

> **Regra de Significado 8.1** *O significado de um Programa é definido como sendo o significado do seu* corpo *quando dado um estado inicial que consista das variáveis da* `partedec`*, cada uma inicializada com um valor indefinido correspondente ao seu tipo declarado.*

A regra define o significado de um programa usando as definições de sintaxe abstrata dos constituintes do *Programa*. Assim, essa regra diz que o significado de um programa é o significado do corpo do programa com o estado inicial em que todas as variáveis declaradas tenham valores indefinidos do tipo apropriado. À medida que o significado do programa se desdobra, esse estado inicial mudará quando forem atribuídos e reatribuídos valores às variáveis, conforme mostrado, por exemplo, na Tabela 7.3.

Esse estilo funcional para definir o significado de um programa é especialmente direto de se implementar em Java. Uma implementação é a seguinte:

```
State M (Program p) {
    return M (p.body, initialState(p.decpart));
}
```

O método `initialState( )` constrói um estado guardando cada variável declarada inicializada com o valor `undef` apropriado para o seu tipo.

Já que um estado é um conjunto de pares chave-valor únicos, é implementado naturalmente com uma tabela *hash*, usando a classe `HashMap` de Java:

```
public class State extends Hashmap { ... }
```

Lembrando que *Declarações* é implementada como uma `ArrayList` de Java e a sintaxe abstrata de *Declaração* é definida:

Declaração = *Variável* `v`; Tipo `t`;

A implementação necessária de `initialState` é:

```
State initialState (Declarations d) {
    State state = new State();
    for (Declaration decl : d)
        state.put(decl.v, Value.mkValue(decl.t));
    return state;
}
```

Os pares variável-valor são colocados na tabela *hash* usando o método `put` de `HashMap`. O método `Value.mkValue`, dado o tipo, retorna um valor indefinido do tipo apropriado.

8.2.2 Semântica de Comandos

O significado de um comando abstrato é uma função de transformação de estado da forma:

S: Comando × *Estado* → *Estado*

Há cinco tipos de comandos em Clite, identificados na sintaxe abstrata como:

Comando = *Salto* | *Atribuição* | *Condicional* | *Laço* | *Bloco*

Assim, o significado de qualquer *Comando* em Clite é do tipo particular do comando que represente. Uma implementação consiste em um método com um comando `if` para cada tipo:

```
State M (Statement s, State state) {
    if (s instanceof Skip) return M(Skip)s, state);
    if (s instanceof Assignment)
        return M((Assignment)s, state);
    if (s instanceof Conditional)
        return M((Conditional)s, state);
    if (s instanceof Loop) return M(Loop)s, state);
    if (s instanceof Block) return M(Block)s, state);
    throw new IllegalArgumentException(
        "nunca deve chegar aqui");
}
```

Observe nosso uso do comando `throw` como uma técnica de programação defensiva para capturar tanto erros lógicos no modelo semântico quanto erros na implementação do modelo.

Salto Um comando *Salto* em Clite não altera o estado, de forma que seu significado é simples.

> **Regra de Significado 8.2** *O significado de um comando Salto é uma função identidade do estado, ou seja, o estado permanece inalterado.*

A implementação da semântica de um `Salto` também é simples; ele retorna o *Estado* que foi passado:

```
State M (Skip s, State state) {
    return state;
}
```

Embora um comando *Salto* não faça nada, ainda assim desempenha um papel importante na programação. Por exemplo, C e C++ permitem efeitos colaterais em testes de laço, de modo que o corpo de um laço pode ser um *Salto*. Para entender melhor, veja a Figura 5.6. Outro uso de um comando *Salto* em Clite é que o analisador converte todos os comandos `se-então` na sintaxe concreta para comandos `se-então-senão` na sintaxe abstrata, inserindo um *Salto* como a ramificação do "senão" de um comando abstrato *Condicional*.

Atribuição A sintaxe abstrata de uma *Atribuição* em Clite possui as seguintes partes:

Atribuição = *Variável* `alvo`*; Expressão* `origem`

Informalmente, o significado de uma *Atribuição* avalia a expressão no estado corrente e produz um novo valor para a variável `alvo` no estado de saída.

> **Regra de Significado 8.3** *O significado de um comando de atribuição é o resultado de substituir o valor da Variável* `alvo` *pelo valor da Expressão* `origem` *no estado corrente.*

O significado de uma *Atribuição* supõe as restrições de tipos discutidas no Capítulo 6. Lembre-se do seguinte exemplo da Figura 7.2:

```
f = f * i;
```

Já que Clite é fortemente tipada, essa *Atribuição* é válida, quanto ao tipo, quando a função seguinte de significado for aplicada, ou seja, o tipo do resultado da expressão, com certeza, é o mesmo da variável alvo. Assim, erros de tipo em tempo de execução (como a atribuição de um valor boleano a um alvo de ponto flutuante) não podem ocorrer.

A Tabela 7.3 mostra que, quando o comando 4 é executado, o valor inteiro 3 substitui o valor *undef* para a variável `n`. Os valores das outras variáveis são inalterados por essa *Atribuição*.

A implementação de uma *Atribuição* usa o método chamado `onion( )`[2] para substituir o valor da variável na implementação da tabela *hash* do *Estado*.

```
State M (Assignment a, State state) {
    return state.onion(a.target, M (a.source, state));
}
```

Basicamente, o método `onion` executa um `put` para substituir o valor da *Variável* `alvo` na tabela *hash* (`estado`) pelo valor da *Expressão* `origem`.

Condicional Um *Condicional* em Clite tem três partes:

Condicional = *Expressão* teste; *Comando* ramificação_então, ramificação_senão

No caso de a `ramificação_senão` ser omitida na sintaxe concreta, o *Condicional* possui um comando *Salto* no seu lugar.

> **Regra de Significado 8.4** *O significado de um Condicional depende do resultado* (*verdadeiro ou falso*) *do seu* `teste` *boleano no estado corrente. Se o* `teste` *for verdadeiro, então o significado da Condição é o significado do Comando* `ramificação_então`*; caso contrário, é o significado do Comando* `ramificação_senão`.

2. O motivo desse nome estranho é explicado na Seção 8.4, que define a matemática e o modelo formal relacionado à implementação.

Analise o seguinte código para calcular o maior entre dois números `a` e `b`:

```
if (a > b)
    max = a;
else
    max = b;
```

Se for verdade que `a > b`, então o significado do *Condicional* é definido como o significado de `max = a`. Caso contrário, é porque `max = b`. Assim, independentemente do resultado do teste `a > b`, o estado da saída reflete uma atribuição a `max` com o valor de `a` ou de `b`.

A implementação imita a regra anterior:

```
State M (Conditional c, State state) {
    if (M(c.test, state).boolValue( ))
        return M (c.thenbranch, state);
    else
        return M (c.elsebranch, state);
}
```

O `teste` *boleano* do *Condicional* é avaliado no estado corrente. Se o `teste` for verdadeiro, o significado do *Condicional* é o estado retornado pela `ramificação_então`; caso contrário, o significado do estado é retornado pela `ramificação_senão`.

Laço O *Laço* de Clite possui um teste e um corpo:

Laço = *Expressão* teste; *Comando* corpo

> **Regra de Significado 8.5** *Se o* `teste` *for falso, o significado* (*estado da saída*) *de um Laço é o mesmo do estado de entrada. Caso contrário, o significado é o resultado da aplicação dessa regra novamente ao significado do seu* `corpo` *no estado corrente.*

Essa é uma definição recursiva. Intuitivamente, um *Laço* especifica a repetição do `corpo` zero ou mais vezes, e cada iteração começa com o estado que resulta do anterior.

Analise novamente o laço (comandos 7–10) no programa de fatorial da Figura 7.2. Na primeira vez em que esse laço é executado, o estado do programa é

$$\{\langle n, 3\rangle, \langle i, 1\rangle, \langle f, 1\rangle\}$$

(veja a Tabela 7.3). Assim, esse estado se torna o estado de entrada para o corpo do laço, efetivamente os comandos 8–9, cujo estado de saída é

$$\{\langle n, 3\rangle, \langle i, 2\rangle, \langle f, 2\rangle\}$$

Esse estado se torna o estado de entrada para uma função recursiva de significado para o laço. O processo se repete assim até o estado

$$\{\langle n, 3\rangle, \langle i, 3\rangle, \langle f, 6\rangle\}$$

o que resulta da terceira iteração do laço retornando o estado de entrada como seu estado de saída, já que `i < n` é falso. Neste ponto, a recursão termina com esse estado como o estado de saída do significado do *Laço*.

A implementação do significado de um *Laço* é uma codificação direta da Regra 8.5:

```
State M (Loop l, State state) {
    if (M (l.test, state).boolValue( ))
        return M(l, M (l.body, state));
    else return state;
}
```

Observe que, se o *Laço* nunca terminar, a recursão também não termina (já que nunca alcança a cláusula `senão`). Isso reflete o fato de que a semântica de um programa é uma função parcial.

Bloco Abstratamente, um *Bloco* em Clite é apenas uma seqüência de comandos a serem executados na ordem em que ocorrem.

Bloco = *Comando**

> **Regra de Significado 8.6** *O significado de um Bloco é o significado agregado dos seus comandos quando aplicados ao estado corrente. Se um Bloco não tiver comandos, o estado não é alterado. Caso contrário, o estado resultante do significado do primeiro Comando no Bloco se torna a base para a definição do significado do resto do bloco.*

Essa também é uma definição recursiva. Por exemplo, analise o *Bloco* que consiste nos comandos 8–9 da Figura 7.2 com estado inicial:

$\{\langle n, 3\rangle, \langle i, 2\rangle, \langle f, 2\rangle\}$

O estado de saída do *Bloco* deve primeiro determinar o estado de saída da *Atribuição* `i = i + 1`, que é:

$\{\langle n, 3\rangle, \langle i, 3\rangle, \langle f, 2\rangle\}$

Esse estado de saída se torna o estado de entrada da *Atribuição* `f = f * 1`, cujo estado de saída é:

$\{\langle n, 3\rangle, \langle i, 3\rangle, \langle f, 6\rangle\}$

Esse último estado se torna o estado de saída do *Bloco*.

A implementação:

```
State M (Block b, State state) {
    for (Statement s : b.members)
        state = M (s, state);
    return state;
}
```

usa um laço para atravessar os *Comandos* do *Bloco*, e usa o estado de saída de um *Comando* como estado de entrada para o próximo.

Isso completa as definições dos diversos tipos de comandos em Clite. Agora passamos para a definição do significado de uma expressão.

8.2.3 Semântica das Expressões

O significado de uma *Expressão* é complicado. De um lado, a principal tarefa de uma *Expressão* é produzir um valor para ser atribuído a uma variável ou ser usado como teste em uma *Condição* ou *Laço*. De outro lado, as *Expressões* podem ter os assim chamados *efeitos colaterais*:

> **Definição:** Um *efeito colateral* ocorre durante a avaliação de uma expressão se, além de retornar um valor, a expressão alterar o estado do programa.

Devido ao tratamento de efeitos colaterais ser um tópico importante na semântica de linguagens, nós o discutiremos integralmente na Seção 8.2.4.

Assim, por enquanto, o significado de uma *Expressão* em Clite é um *Valor*:

$$S\text{: } Expressão \times Estado \rightarrow Valor$$

Uma *Expressão* é sempre avaliada no estado corrente porque pode conter variáveis cujos *Valores* podem ter mudado em estados anteriores.

Lembre-se de que uma *Expressão* em Clite possui a seguinte sintaxe abstrata:

$$\begin{aligned} Expressão &= Variável \mid Valor \mid Binário \mid Unário \\ Binário &= OpBinário\ \texttt{op};\ Expressão\ \texttt{termo1}, \texttt{termo2} \\ Unário &= OpUnário\ \texttt{op};\ Expressão\ \texttt{termo} \\ Variável &= String\ \texttt{id} \\ Valor &= ValorInt \mid ValorBool \mid ValorFloat \mid ValorChar \end{aligned}$$

em que as expressões *Binário* e *Unário* envolvem operadores binários e unários e seus operandos, respectivamente.[3] O significado de uma *Expressão* é então definido com base nas suas quatro formas alternativas:

> **Regra de Significado 8.7** *O significado de uma Expressão em um estado é um Valor definido da seguinte maneira:*
>
> **1** *Se a Expressão for um Valor, então seu significado é o do próprio Valor. Um exemplo de Valor é o inteiro 3.*
>
> **2** *Se a Expressão for uma Variável, então seu significado é o Valor da Variável no estado corrente. Na Tabela 7.3, o significado da variável* `i` *no estado*
>
> $$\{\langle n, 3\rangle, \langle i, 2\rangle, \langle f, 2\rangle\}$$
>
> *é* 2.
>
> **3** *Se a Expressão for um Binário, então o significado de cada um de seus operandos* `termo1` *e* `termo2` *será determinado primeiro. Então a Regra 8.8 determina o significado da expressão aplicando o Operador* `op` *aos Valores desses dois operandos.*
>
> **4** *Se a Expressão for um Unário, então o significado do seu operando* `termo` *é determinado. A Regra 8.9 determina então o significado da expressão aplicando o Operador* `op` *ao Valor do operando.*

3. Para mantermos o foco nas questões principais, supomos, na discussão a seguir, que todos os operadores em Clite, exceto a atribuição, tenham um tipo específico, de modo que não há sobrecarga de operadores (veja o Capítulo 4) e a adição de inteiros + é diferente da adição de pontos flutuantes +. As regras de transformação de estados também supõem que transformações e alguns tipos primitivos sejam predefinidos e constituam, assim, um domínio de semântica para Clite.

Essa regra possui uma implementação direta:

```
Value M (Expression e, State state) {
    if (e instanceof Value)
        return (Value)e;
    if (e instanceof Variable)
        return (Value)(state.get(e));
    if (e instanceof Binary) {
        Binary b = (Binary)e;
        return applyBinary (b.op,
            M(b.term1, state), M(b.term2, state));
    }
    if (e instanceof Unary) {
        Unary u = (Unary)e;
        return applyUnary(u.op, M(u.term, state));
    }
    throw new IllegalArgumentException(
        "nunca deve chegar aqui");
}
```

Na implementação, o significado de um *Valor* ou de uma *Variável* é exatamente conforme foi especificado na regra anterior. Para uma expressão *Binária*, ambos os operadores são avaliados no estado corrente e então o método `applyBinary` é chamado para aplicar o operador `op` aos valores dos dois operandos. A explicação da avaliação de uma expressão *Unária* é semelhante.

Analise, por exemplo, a avaliação da *Expressão* `origem` do comando 8 da Figura 7.2:

```
i = i + 1;
```

com o estado de entrada mostrado no passo 5 da Tabela 7.3. O valor de `i` é 1 (Regra 4.7.2), e o valor de 1 é 1 (Regra 4.7.2) Assim, o valor de `i + 1` é 2 (Regras 4.7.3 e 4.8).

Lembre-se do que vimos no Capítulo 6, que operadores concretos sobrecarregados em Clite eram transformados em operadores abstratos e específicos de tipo. Por exemplo, os operadores aritméticos binários +, −, * e / eram convertidos para operadores específicos de tipo `int+`, `int-`, `int*`, `int/` para operandos `int` e `float+`, `float-`, `float*` e `float/` para operandos `float`. Além disso, quaisquer conversões de tipos necessárias eram inseridas na árvore de sintaxe abstrata. Assim, um `float+`, por exemplo, garante que ambos os seus operandos sejam do tipo `float`.

Regra de Significado 8.8 *O significado de uma expressão Binária é um Valor definido da seguinte maneira:*

1. *Se o operando* `termo1` *ou* `termo2` *estiver indefinido, o programa não tem significado.*
2. *Se o operador for um aritmético inteiro, então um* `int+`, `int-`, `int*` *executam adição, subtração ou multiplicação sobre seus operandos inteiros, resultando em um inteiro. Se o operador for* `int/`*, então o resultado é o mesmo da divisão matemática com truncagem da parte fracionária. Por exemplo, um valor matemático de −3,9 é truncado na direção do zero, resultando em um valor* `int` *−3. De forma semelhante, um resultado matemático de 3,9 é truncado para 3.*

3 *Se o operador for de ponto flutuante, então a aritmética de ponto flutuante usando o padrão IEEE é executada sobre os operandos* `float`*, revertendo em um resultado do mesmo tipo.*

4 *Se o operador for relacional, então os operandos são comparados com o resultado* `verdadeiro` *ou* `falso`*. O resultado é o mesmo resultado matemático correspondente, exceto que* `falso < verdadeiro`*.*

5 *Se o operador for boleano, então:*

(a) *O operador && é interpretado como:*

a *&&* $b \equiv$ se a então b senão *falso*

(b) *O operador* || *é interpretado como:*

$a \mid\mid b \equiv$ se a então *verdadeiro* senão b

A implementação a seguir mostra apenas os operadores aritméticos inteiros sobre operandos inteiros. Os casos restantes constituiriam um comando *switch* gigante.

```
Value applyBinary (Operator op, Value v1, Value v2) {
    StaticTypeCheck.check( ! v1.isUndef( ) &&
        ! v2.isUndef( ), "reference to undef value");
    if (op.val.equals(Operator.INT_PLUS))
        return new IntValue(
        v1.intValue( ) + v2.intValue( ));
    if (op.val.equals(Operator.INT_MINUS))
        return new IntValue(
        v1.intValue( ) - v2.intValue( ));
    if (op.val.equals(Operator.INT_TIMES))
        return new IntValue(
        v1.intValue( ) * v2.intValue( ));
    if (op.val.equals(Operator.INT_DIV))
        return new IntValue(
        v1.intValue( ) / v2.intValue( ));
    ...
}
```

Assim, a implementação completa imitaria o que a máquina real faz, exceto pelo fato de os computadores reais não filtrarem variáveis não inicializadas como o nosso modelo deve fazer.

Nesse código, cada operador inteiro possui um operador Java correspondente aplicado aos operandos inteiros, produzindo um novo *Valor* inteiro, que é o resultado. Nosso interpretador detecta uma tentativa de interpretar um caractere ou um valor de ponto flutuante como um inteiro, o que seria um erro de tipo na implementação. C++, e assim Clite, define falso como menor que verdadeiro, o que pode ser implementado em Java mapeando o valor falso para 0 e o verdadeiro para 1. A não ser a nossa implementação de operadores relacionais sobre operandos boleanos, nosso verificador de tipos não permite que boleanos sejam interpretados como valores inteiros, o que é um distanciamento de C/C++.

Regra de Significado 8.9 *O significado de uma expressão Unária é um Valor definido da seguinte maneira:*

1 *Se o operando* `termo` *for indefinido, o significado da expressão é indefinido.*

2 *Se o operador for* `!` (*não*), *então o operando boleano é invertido.*

3 *Se o operador for* `int-/float-`, *então o sinal do inteiro/ponto flutuante é invertido.*

4 *Se o operador for* `i2f`, *então o operando inteiro é convertido para ponto flutuante. Por exemplo, o valor 2 seria convertido para 2,0.*

5 *Se o operador for* `c2i`, *então o operando* `char` *é convertido para inteiro usando o código ASCII do caractere. Efetivamente, exceto pela adição de bits com valor zero, o valor não é alterado, exceto pelo seu tipo.*

6 *Se o operador for* `i2c`, *então o operador inteiro é convertido para caractere. Se o operando inteiro for maior que 255 ou menor que 0, o significado da expressão é indefinido. Se a conversão for bem-sucedida, então, exceto pela perda de zeros à esquerda, o padrão de bits do valor é o mesmo, apenas o tipo é alterado.*

7 *Se o operador for* `f2i`, *então o valor de ponto flutuante é convertido para inteiro, descartando-se as posições após o ponto decimal, ou seja, ele é truncado na direção do zero. Por exemplo, o valor 2,9 é convertido para 2, e −2,9 é convertido para −2. Se a parte inteira do valor de ponto flutuante for grande demais para ser armazenada em um inteiro, o significado da expressão é indefinido.*

A implementação segue da regra anterior:

```
Value applyUnary (Operator op, Value v) {
    StaticTypeCheck.check( ! v.isUndef( ),
        “reference to undef value”);
    if (op.val.equals(Operator.NOT))
        return new BoolValue(!v.boolValue( ));
    else if (op.val.equals(Operator.INT_NEG))
        return new IntValue(-v.intValue( ));
    else if (op.val.equals(Operator.FLOAT_NEG))
        return new FloatValue(-v.floatValue( ));
    else if (op.val.equals(Operator.I2F))
        return new FloatValue((float)(v.intValue( )));
    else if (op.val.equals(Operator.F2I))
        return new IntValue((int)(v.floatValue( )));
    else if (op.val.equals(Operator.C2I))
        return new IntValue((int)(v.charValue( )));
    else if (op.val.equals(Operator.I2C))
        return new CharValue((char)(v.intValue( )));
    throw new IllegalArgumentException(
        “nunca deve chegar aqui”);
}
```

Diferentemente da aplicação de `applyBinary`, todos os casos são mostrados.

Isso completa a definição e a implementação da semântica de tempo de execução de Clite. Uma descrição formal da semântica é apresentada na Seção 8.4.

8.2.4 Expressões com Efeitos Colaterais

Além de retornar um valor, uma expressão tem um efeito colateral sobre o estado da computação. Por exemplo, o aparecimento de uma chamada de função `f(x)` em uma expressão como `f(x) + x` parece inócua sob circunstâncias normais. Todavia, se essa chamada alterar o valor de `x` no estado em que ele é avaliado, o valor retornado será diferente, dependendo da ordem em que os dois operandos forem avaliados, da esquerda para a direita ou da direita para a esquerda.

Observe na Regra de Significado 8.7 que a ordem de avaliação de operandos em um *Binário* não é especificada. Entretanto, o código Java que acompanha essa definição claramente avalia os operandos da esquerda para a direita, já que a chamada de função Java:

```
applyBinary(b.op, M(b.term1, state), M(b, term2, state)
```

avalia seus argumentos da esquerda para a direita.

Efeitos colaterais também podem alterar o comportamento normal dos comandos de *Condicional* e *Laço*. Por exemplo, suponha que a chamada de função `g(x)` produza um resultado boleano e também tenha um efeito colateral sobre `x`. Então o seguinte:

```
if (g(x))
    y = x;
```

pode ou não atribuir a `y` um valor anterior de `x` quando `g(x)` for verdadeira.

Uma forma simples de lidar semanticamente com efeitos colaterais em uma linguagem é simplesmente proibi-los. Todavia, isso teria conseqüências sérias em linguagens como C, em que mesmo expressões como `x++` dentro de uma expressão maior não seriam permitidas.

Outra forma de lidar com efeitos colaterais é expandir a definição do significado de uma expressão de modo que ela produza tanto um *Valor* quanto um *Estado*. A seguir está uma revisão da Regra de Significado 8.7 que permite que uma *Expressão* tenha um efeito colateral alterador de estado.

S : *Expressão* $\times$ *Estado* $\rightarrow$ *Valor* $\times$ *Estado*

Regra de Significado 8.10 *O significado de uma Expressão em um estado é um Valor e um Estado, definido da seguinte maneira:*

1. *Se a Expressão for um Valor, então seu significado é o significado do próprio Valor e o estado corrente não é alterado.*
2. *Se a Expressão for uma Variável, então seu significado é o Valor da Variável no seu estado corrente, o qual permanece inalterado.*
3. *Se a Expressão for um Binário, o significado do* `termo1` *no estado corrente é determinado primeiro, dando um valor* v_1 *e um estado* s_1. *A seguir, o significado do* `termo2` *no estado* s_1 *é determinado, dando um valor* v_2 *e um estado* s_2. *A seguir, a Regra 8.8 determina o significado da expressão aplicando o Operador* `op` *aos valores resultantes* v_1 *e* v_2 *no estado* s_2, *e o estado resultante é* s_2.
4. *Se a Expressão for Unária, então o significado do seu operando* `termo` *no estado corrente é determinado, dando o valor* v_1 *e estado* s_1. *Então a Regra 8.9 determina o significado da expressão, aplicando o Operador* `op` *ao Valor do operando, e o estado resultante é* s_1.

Observe que o terceiro passo nessa regra define a avaliação da esquerda para a direita. Além disso, o terceiro e o quarto passos supõem que os operadores binários e unários na linguagem não tenham eles próprios efeitos colaterais. Se tiverem, então as Regras de Significado 8.8 e 8.9 precisariam ser expandidas de modo que esses operadores também produzam um par valor-estado.

Finalmente, observe que a introdução de efeitos colaterais dessa forma terá impacto até mesmo sobre o significado de comandos *Condicional* e *Laço*. Por exemplo, quando a avaliação de uma *Expressão* `teste` de um *Laço* tiver um efeito colateral, os valores ou algumas variáveis podem ser alteradas antes que seu `corpo` comece a ser interpretado.

8.3 SEMÂNTICA COM TIPAGEM DINÂMICA

Aqui analisaremos a semântica de uma linguagem tipada dinamicamente para examinar a diferença entre linguagens tipadas estática e dinamicamente de forma mais completa. Usamos o mesmo método de apresentação da Seção 8.2.

Muitas linguagens tipadas dinamicamente diferentes poderiam servir como veículo para este estudo, incluindo JavaScript, Perl, Python e Scheme. Contudo, preferimos usar uma versão modificada de Clite para essa discussão, porque ela é familiar e é suficiente para ilustrar os principais pontos por trás da tipagem dinâmica.

Chamamos essa versão de Clite$_D$, já que usa o mesmo analisador léxico, o analisador e a sintaxe abstrata que foram usados para Clite. Nosso modelo semântico para Clite$_D$ interpreta a sintaxe abstrata diretamente, sem pré-processamento com um verificador ou transformador de tipos.

Analise o programa para cálculo do fatorial apresentado na Figura 8.1. Ele se parece muito com o da Figura 7.2, exceto pela ausência de declarações. Além de eliminar declarações, que outros recursos Clite$_D$ deveria ter?

Por exemplo, Perl possui operadores distintos para comparações de *strings versus* numéricos. Em Perl, `2<10` é verdadeiro usando um teste numérico, mas `2 lt 10` é falso, já que o teste é uma comparação de *strings*, ou seja, `"2" lt "10"` é o teste realmente executado. Em comparação, Python sobrecarrega os operadores como C/C++; para operadores sobrecarregados como comparações, se um operando for uma *string*, o outro também deve sê-lo. Na especificação da semântica de Clite$_D$, usaremos bastante Python como nosso modelo.

Nesta apresentação, discutimos apenas as partes da semântica de Clite$_D$ que diferem da semântica de Clite. A sintaxe abstrata de um *Programa* em Clite$_D$ é:

Programa = *Bloco* `corpo`

```
1  int main ( ) {
2     n = 3;
3     i = 1;
4     f = 1.0;
5     while (i < n) {
6        i = i + 1
7        f = f * float(i);
8     }
9  }
```

| Figura 8.1 Programa Fatorial em Clite$_D$

A primeira diferença entre os dois modelos semânticos é no significado de um programa. Lembre-se da Regra 8.1, em que a interpretação de um *Programa* começava com um *Estado* no qual todas as variáveis declaradas eram inicializadas com o valor especial *undef*. Aqui não há variáveis declaradas, de modo que o *Programa* começa com um *Estado* vazio.

> **Regra de Significado 8.11** *O significado de um Programa é definido como o de seu* `corpo` *quando dado um estado inicial vazio.*

Usando essa regra, a Tabela 8.1 mostra um rastro da execução do programa fatorial da Figura 8.1. Observe que o estado no passo 1 é vazio. À medida que os comandos de atribuição são executados, novas variáveis `alvos` são adicionadas ao estado. O tipo da variável é o tipo do seu valor no estado corrente. Por exemplo, no passo 4, as variáveis `n` e `i` têm tipo inteiro, enquanto a variável `f` é de ponto flutuante.

Lembre-se de que, para linguagens tipadas dinamicamente, nenhuma verificação de tipo é executada no programa. Apesar disso, o significado de uma *Expressão* nas duas linguagens (Regra 8.7) é quase idêntico. Por exemplo, o significado de um *Valor*, já que a função *S* deve retornar um *Valor*, é esse próprio *Valor*. Na definição do significado de uma *Variável* em Clite$_D$, insistimos que a *Variável* tem um *Valor*. Aqui está a definição completa:

> **Regra de Significado 8.12** *O significado de uma Expressão em um estado é um Valor definido da seguinte maneira:*
>
> **1** *Se a Expressão for um Valor, então seu significado é o significado do próprio Valor.*
>
> **2** *Se a Expressão for uma Variável, então seu significado é o significado da Variável no estado corrente. Se a Variável não estiver definida no estado corrente, o programa é semanticamente sem significado. Por exemplo, na Tabela 8.1, a variável* `f` *no passo 2 não ocorre no estado corrente; portanto, uma referência a* `f` *no comando 4 seria um erro.*
>
> **3** *Se a Expressão for um Binário, então o significado de ambos os seus operandos* `termo1` *e* `termo2` *são determinados. Então a Regra 8.13 determina o significado da Expressão, aplicando o Operador* `op` *aos Valores dos operandos.*

Tabela 8.1 Rastro do Programa Fatorial

Passo	Comando	Estado
1	3	
2	4	<*n*, 3>
3	5	<*n*, 3>, <*i*, 1>
4	6	<*n*, 3>, <*i*, 1>, <*f*, 1,0>
5	7	<*n*, 3>, <*i*, 1>, <*f*, 1,0>
6	8	<*n*, 3>, <*i*, 2>, <*f*, 1,0>
7	6	<*n*, 3>, <*i*, 2>, <*f*, 2,0>
8	7	<*n*, 3>, <*i*, 2>, <*f*, 2,0>
9	8	<*n*, 3>, <*i*, 3>, <*f*, 2,0>
10	6	<*n*, 3>, <*i*, 3>, <*f*, 6,0>
11	10	<*n*, 3>, <*i*, 3>, <*f*, 6,0>

4 *Se a Expressão for Unária, então o significado do seu operando* `termo` *é determinado. Portanto, a Regra 8.14 determina o significado da Expressão, aplicando o Operador* `op` *ao Valor do operando.*

Esta regra tem uma implementação direta:

```
Value M (Expression e, State sigma) {
    if (e instanceof Value)
        return (Value)e;
    if (e instanceof Variable) {
        StaticTypeCheck.check( sigma.containsKey(e),
            "reference to undefined variable");
        return (Value)(sigma.get(e));
    }
    if (e instanceof Binary) {
        Binary b = (Binary)e;
        return applyBinary (b.op,
                    M(b.term1, sigma), M(b.term2, sigma));
    }
    if (e instanceof Unary) {
        Unary u = (Unary)e;
        return applyUnary (u.op, M(u.term, sigma));
    }
    throw new IllegalArgumentException(
        "nunca deve chegar aqui");
}
```

Nessa implementação, os significados de um *Binário* e de um *Unário* são deferidos para os métodos `applyBinary` e `applyUnary`, respectivamente. Para o significado de uma *Variável* em Clite$_D$, o interpretador semântico verifica se o *Estado* corrente contém a variável quando for referenciada em uma *Expressão.*

Para um *Binário*, ambos os operandos são avaliados no estado corrente e então o método `applyBinary` é chamado para aplicar o operador `op` aos valores dos dois operandos. A explicação da avaliação da expressão *Unário* é semelhante.

Lembre-se do que foi dito no Capítulo 6, que os operadores sobrecarregados em Clite são transformados em operadores dependentes de tipos. Não existe tal fase para uma linguagem tipada dinamicamente. Em vez disso, os operadores permanecem sobrecarregados na sintaxe abstrata, da mesma forma que foram construídos pelo analisador. Assim, a verificação de se ambos os operandos de um operador binário são do mesmo tipo e compatíveis com o operador deve ser executada pela regra de significado (por exemplo, em tempo de execução).

Regra de Significado 8.13 *O significado de um Binário é um Valor, definido da seguinte forma:*

1 *Se o operador for aritmético, então:*

(a) *Se algum operando for do tipo inteiro, o outro deve ser desse tipo também. Se os operadores forem um* +, − ou *, *o resultado é a soma, a diferença ou o produto dos operandos. Se o operador for /, o resultado é o quociente matemático com truncagem na direção do zero. Por exemplo, o quociente* −3,9 *é truncado para o valor* `int` −3.

(b) *Se algum operando for do tipo ponto flutuante, o outro deve ser do mesmo tipo. A aritmética de ponto flutuante usando o padrão IEEE é executada sobre operadores* `float`*, resultando em* `float`.

2 *Se o operador for relacional, então os operandos são comparados com um resultado de* `verdadeiro` *ou* `falso`*. O resultado é o mesmo resultado matemático correspondente. Seguindo C/C++,* `falso` < `verdadeiro`.

3 *Se o operador for boleano, então:*

(a) *O significado de* `&&` *é:* se *a* então *b* senão *falso*

(b) *O significado de* `||` *é:* se *a* então *verdadeiro* senão *b*

em que tanto a quanto b devem ser do tipo boleano.

4 *Qualquer outra combinação operador/tipo não é permitida.*

A seguir está uma implementação da semântica aritmética inteira em Clite$_D$:

```
Value applyBinary (Operator op, Value v1, Value v2) {
    StaticTypeCheck.check( v1.type( ) == v2.type( ),
                           "mismatched types");
    if (op.ArithmeticOp( )) {
        if (v1.type( ) == Type.INT) {
            if (op.val.equals(Operator.PLUS))
                return new IntValue(
            v1.intValue( ) + v2.intvalue( ));
            if (op.val.equals(Operator.MINUS))
                return new IntValue(
            v1.intValue( ) - v2.intvalue( ));
            if (op.val.equals(Operator.TIMES))
                return new IntValue(
            v1.intValue( ) * v2.intvalue( ));
            if (op.val.equals(Operator.DIV))
                return new IntValue(
            v1.intValue( ) / v2.intvalue( ));
        }
        ...
}
```

Observe o teste no tipo do operando `v1` à esquerda para determinar se ele é um inteiro. Apenas os primeiros casos são mostrados do que é basicamente um comando *case* gigante.

Para cada operador inteiro, o operador Java correspondente é aplicado aos operandos inteiros, e então um novo *Valor* inteiro é construído contendo o resultado. Nosso interpretador detecta uma tentativa de interpretar um caractere ou um valor de ponto flutuante como um inteiro, o que seria um erro de tipo na implementação.

Regra de Significado 8.14 *O significado de um Unário é um Valor definido da seguinte maneira:*

1 *Se o operador for* `!` *(não), então o operando boleano é invertido.*

2 *Se o operador for unário* −*, então o operando inteiro/ponto flutuante tem seu sinal invertido.*

3 *Se o operador for* `float`*, então o operando inteiro é convertido para ponto flutuante. Por exemplo, o valor 2 seria convertido para 2,0.*

4 *Se o operador for* `int`*, então:*

(a) *Um operando* `char` *é convertido para inteiro, por meio do uso do código ASCII do caractere. Efetivamente, exceto pela adição de bits zero à esquerda, o valor é inalterado, exceto pelo seu tipo.*

(b) *Um valor* `float` *é convertido em inteiro descartando a parte fracionária. Por exemplo, o valor −2,9 para −2. Se a parte inteira do valor de ponto flutuante for grande demais para ser armazenada como um* `int`*, o resultado é indefinido.*

5 *Se o operador for* `char`*, então o operando inteiro é convertido para caractere. Se o operando inteiro for maior que 255 ou menor que zero, o resultado é indefinido. Se a conversão for bem-sucedida, então, exceto pela perda de zeros à esquerda, o padrão de bits do valor é o mesmo; apenas o tipo é alterado.*

6 *Qualquer outra combinação operador/tipo não é permitida.*

A implementação desta regra também é direta:

```
...
    else if (op.val.equals(Operator.FLOAT))
        return new FloatValue((float)(v.intValue( )));
    else if (op.val.equals(Operator.INT))
        return new IntValue((int)(v.intValue( )));
    else if (op.val.equals(Operator.INT))
        return new IntValue((int)(v.intValue( )));
    else if (op.val.equals(Operator.CHAR))
        return new CharValue((char)(v.intValue( )));
    throw new IllegalArgumentException(
        "should never reach here");
...
```

Da mesma forma que a implementação de `applyBinary`, apenas alguns casos são mostrados do que é basicamente um comando *case* gigante.

π 8.4 UM TRATAMENTO FORMAL DE SEMÂNTICA

Esta seção apresenta a semântica de Clite pelo uso de um estilo denotacional formal. Ela espelha as regras de significado para comandos e expressões apresentadas na Seção 8.2.

O propósito desta apresentação é mostrar como a semântica de uma linguagem de programação pode ser definida com precisão matemática. Tal definição pode convencer projetistas e implementadores de linguagens de que uma linguagem é bem definida. A boa definição é uma característica essencial para linguagens de programação, especialmente àquelas cujos programas devem se provar corretos (veja o Capítulo 18).

Leitores que decidam estudar esta seção devem estar familiarizados com os tópicos matemáticos necessários que são revistos no Apêndice B.

8.4.1 Estados e Transformação de Estados

Na determinação do significado de uma variável na Seção 8.2, tivemos de acessar o valor da variável no seu estado corrente. Assim como com os tipos no Capítulo 5, uma variável e seu valor são modelados como um par ordenado, por exemplo, $\langle x, 3 \rangle$.

Assim, um estado é um conjunto de pares ordenados, em que o primeiro elemento é o nome da variável e o segundo é seu valor. Por exemplo, a expressão a seguir descreve o

estado do nosso programa fatorial, que corresponde à linha 8 da Tabela 7.3:

$$estado = \{\langle n,3\rangle, \langle i,2\rangle, \langle f,2\rangle\}$$

Para este determinado estado, podemos recuperar o valor de uma variável, digamos *f*, escrevendo a expressão *estado*(*f*), que apresenta o valor 2. Já que os nomes de variáveis são únicos, um estado, como um conjunto de pares ordenados, é equivalente a uma função. Assim, a expressão *estado*(*v*) denota a função que recupera o valor da variável *v* do *estado* corrente.

Uma transformação de estado ocorre sempre que uma atribuição for interpretada em um programa imperativo ou orientado a objeto. Por exemplo, analise a seguinte atribuição:

```
i = i + 1;
```

cujo efeito é alterar o estado anterior da seguinte maneira:

$$estado = \{\langle n,3\rangle, \langle i,3\rangle, \langle f,2\rangle\}$$

Em Clite, a variável alvo sempre ocorre no estado. Por outro lado, uma atribuição em Clite_D, como:

```
f = 1;
```

pode ter o efeito de transformar o estado de:

$$estado = \{\langle n,3\rangle, \langle i,1\rangle\}$$

para o estado:

$$estado = \{\langle n,3\rangle, \langle i,1\rangle, \langle f,1\rangle\}$$

Esse último pode ser modelado com o uso da união comum de conjuntos. Entretanto, usar união de conjuntos para o primeiro criaria o estado:

$$estado = \{\langle n,3\rangle, \langle i,3\rangle, \langle i,2\rangle, \langle f,2\rangle\}$$

o que é claramente incorreto.

Transformações de estado que representem ambos os tipos de atribuições podem ser representadas matematicamente por uma função especial denominada *união de substituição*, representada pelo símbolo $\overline{\cup}$. Essa função é semelhante à união comum de conjuntos, exceto quando altera o valor de uma variável que já esteja presente no estado corrente.

> **Definição:** A *união de substituição* de *X* e *Y*, escrita $X \overline{\cup} Y$, é o resultado da substituição em *X* de todos os pares $\langle x,v\rangle$ cujo primeiro membro corresponda a um par $\langle x,w\rangle$ de *Y* por $\langle x,w\rangle$, e depois adicionam-se a *X* todos os pares restantes em *Y*.

Por exemplo, suponha que $estado_1 = \{\langle x,1\rangle, \langle y,2\rangle, \langle z,3\rangle\}$ e $estado_2 = \{\langle y,9\rangle, \langle w,4\rangle\}$. Então, $estado_1 \overline{\cup} estado_2 = \{\langle x,1\rangle, \langle y,9\rangle, \langle z,3\rangle, \langle w,4\rangle\}$.

Outra forma de visualizar a união de substituição é por meio da junção natural desses dois conjuntos. A *junção natural* $estado_1 \otimes estado_2$ é o conjunto de todos os pares de $estado_1$ e $estado_2$ que tenham o mesmo primeiro membro. Por exemplo,

$$\{\langle x,1\rangle, \langle y,2\rangle, \langle z,3\rangle\} \otimes \{\langle y,9\rangle, \langle w,4\rangle\} = \{\langle y,2\rangle, \langle y,9\rangle\}$$

Lembre-se de que a diferença de conjunto, $estado_1 - (estado_1 \otimes estado_2)$, remove efetivamente cada par de $estado_1$ cujo primeiro membro seja idêntico ao primeiro mem-

bro de algum par no $estado_2$. Agora a união de substituição pode ser definida em termos do seu conjunto de diferenças da seguinte maneira:

$$estado_1 \overline{\cup}\, estado_2 = (estado_1 - (estado_1 \otimes estado_2)) \cup estado_2$$

Os leitores podem ter observado que o operador $\overline{\cup}$ é um modelo formal e generalizado da operação familiar de atribuição na programação imperativa. Ele desempenha, assim, um papel central na semântica formal de linguagens imperativas e orientadas a objeto, como C/C++, Ada e Java. Ilustraremos essa característica a seguir.

8.4.2 Semântica Denotacional de um Programa

Lembre-se da Seção 8.2, em que a função de significado *S* é um mapeamento:

$$S\text{: } Programa \rightarrow Estado$$

$$S\text{: } Comando \times Estado \rightarrow Estado$$

$$S\text{: } Expressão \times Estado \rightarrow Valor$$

Isso quer dizer que o significado de um *Programa* é uma função que produz um *Estado* de um *Programa*. De forma semelhante, o significado de um *Comando* é uma função que produz um *Estado* desse *Comando* e um *Estado*. Finalmente, o significado de uma *Expressão* é uma função que, dada uma expressão e um *Estado*, produz um *Valor*.[4]

Lembre-se também do que foi citado na Seção 8.2, que essas funções são funções parciais, já que não são bem definidas por todos os membros do seu domínio. Assim, as representações abstratas de certas construções de programas em determinados estados não têm representações finitas de significados, embora essas construções sejam sintaticamente válidas.

Já que a sintaxe abstrata de um programa em Clite é uma estrutura de árvore cuja raiz é o elemento abstrato *Programa*, o significado de um programa Clite pode ser definido como uma série de funções, a primeira das quais define o significado de *Programa*. Os leitores devem perceber essa função como uma redeclaração matemática da Regra de Significado 8.1.

$$S\text{ (}Programa\text{ p)} = S\text{ (p.corpo, }EstadoInicial\text{ (p.partedec))}$$

A primeira linha dessa definição é um protótipo da função *S*, enquanto a segunda linha define a própria função, usando as definições de sintaxe abstrata dos constituintes diretos do *Programa*:

$$Programa = Declarações\ \texttt{partedec};\ Bloco\ \texttt{corpo}$$

$$Declarações = Declaração*$$

$$Declaração = Variável\ \texttt{v};\ Tipo\ \texttt{t}$$

Assim, essa definição funcional diz que o significado de um programa é o significado do corpo do programa com o estado inicial produzido pela função *EstadoInicial* aplicada à lista de declarações.

A função *EstadoInicial*, dada uma lista de declarações, produz:

$$\{\langle v_1, undef_{t_1}\rangle, \ldots, \langle v_m, undef_{t_m}\rangle\}$$

Isso significa que todas as variáveis declaradas do programa são armazenadas no estado inicial com seus valores configurados como o valor especial *undef* apropriado para cada tipo.

4. Para simplificar, essa definição exclui a questão de efeitos colaterais abordada na Seção 8.2.4.

8.4.3 Semântica Denotacional dos Comandos

Lembre-se de que um *Comando* em Clite é definido na sintaxe abstrata como:

$$
\begin{aligned}
\textit{Comando} &= \textit{Salto} \mid \textit{Bloco} \mid \textit{Atribuição} \mid \textit{Condição} \mid \textit{Laço} \\
\textit{Salto} &= \\
\textit{Bloco} &= \textit{Comando}^* \\
\textit{Atribuição} &= \textit{Variável}\ \texttt{alvo} \quad ; \textit{Expressão}\ \texttt{origem} \\
\textit{Condicional} &= \textit{Expressão}\ \texttt{teste} \ ; \textit{Comando}\ \texttt{ramificação\_então}, \texttt{ramificação\_senão} \\
\textit{Laço} &= \textit{Expressão}\ \texttt{teste}; \textit{Comando}\ \texttt{corpo}
\end{aligned}
$$

O significado de um comando é, assim, o do tipo determinado do comando que ele representa:

$$
\begin{aligned}
&S : \textit{Comando} \times \textit{Estado} \rightarrow \textit{Estado} \\
&S\,(\textit{Comando}\ \mathrm{d},\ \textit{Estado estado}) = S\,((\textit{Salto})d,\ \textit{estado}) && \text{se } d \text{ for um } \textit{Salto} \\
&\qquad\qquad\qquad\qquad\quad = S\,((\textit{Atribuição})d,\ \textit{estado}) && \text{se } d \text{ for uma } \textit{Atribuição} \\
&\qquad\qquad\qquad\qquad\quad = S\,((\textit{Condicional})d,\ \textit{estado}) && \text{se } d \text{ for um } \textit{Condicional} \\
&\qquad\qquad\qquad\qquad\quad = S\,((\textit{Laço})d,\ \textit{estado}) && \text{se } d \text{ for um } \textit{Laço} \\
&\qquad\qquad\qquad\qquad\quad = S\,((\textit{Bloco})d,\ \textit{estado}) && \text{se } d \text{ for um } \textit{Bloco}
\end{aligned}
$$

Salto O significado de um comando *Salto* é efetivamente a função identidade, já que não altera o estado corrente da computação (lembre-se da Regra de Significado 8.2).

$$S\,(\textit{Salto}\ \mathrm{p},\ \textit{Estado estado}) = \textit{estado}$$

Atribuição O significado de uma *Atribuição* pode ser expresso como uma união de substituição (lembre-se da Regra de Significado 8.3):

$$S : \textit{Atribuição} \times \textit{Estado} \rightarrow \textit{Estado}$$

$$S\,(\textit{Atribuição}\ \mathrm{a},\ \textit{Estado estado}) = \textit{estado} \cup \{\langle \mathrm{a.alvo},\ S(\mathrm{a.origem},\ \textit{estado})\rangle\}$$

Essa é uma maneira formal de definir o estado que resulta da transformação do estado corrente em um novo *estado* que difira do antigo apenas pelo par cujo primeiro membro é a *Variável* alvo `a.target` da *Atribuição.* O segundo membro desse par é o valor da origem *Expressão* `a.origem` da *Atribuição.* O significado dessa *Expressão* será definido formalmente por outra função *S* na Seção 8.4.4.

Analise o exemplo da Figura 8.2 e suponha que o *estado* corrente seja:

$$\textit{estado} = \{\langle x, 2\rangle, \langle y, -3\rangle, \langle z, 75\rangle\}$$

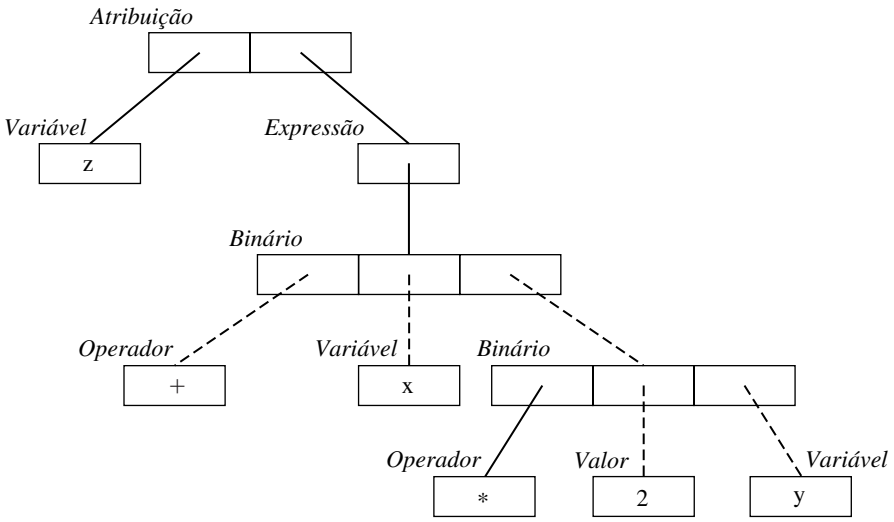
Intuitivamente, esperamos que o significado da expressão origem neste estado, ou

$$S\,(\textit{a.origem, estado}) = S\,(x + 2 * y, \{\langle x, 2\rangle, \langle y, -3\rangle, \langle z, 75\rangle\})$$

é o *Valor* -4. Assim,

$$
\begin{aligned}
\mathrm{S}\,(z = x + 2 * y, \{\langle x,2\rangle, \langle y,-3\rangle, \langle z,75\rangle\}) &= \{\langle x,2\rangle, \langle y,-3\rangle, \langle z,75\rangle\} \\
&\quad \times \cup \{\langle z,-4\rangle\} \\
&= \{\langle x,2\rangle, \langle y,-3\rangle, \langle z,-4\rangle\}
\end{aligned}
$$

o que completa a transformação de estado para essa atribuição.



| Figura 8.2 Sintaxe Abstrata para a Atribuição z = x + 2 * y;

Condicional O significado de um *Condicional* é o significado de uma ou outra de suas ramificações, dependendo de o teste ser verdadeiro ou falso (veja a Regra de Significado 8.4).

$$S\,(\textit{Condição}\ \text{c},\ \textit{Estado estado}) = S\,(c.\text{ramificação\_então}, \textit{estado}) \text{ se } S\,(c.\text{teste}, \textit{estado}) \text{ for verdadeiro}$$
$$= S\,(c.\text{ramificação\_senão}, \textit{estado}) \text{ caso contrário}$$

Analise o código a seguir para calcular o maior entre dois números `a` e `b`:

```
if (a > b)
    max = a;
else
    max = b;
```

Dado o estado *estado* = {⟨a, 3⟩, ⟨b, 1⟩}, o significado dessa condição apresenta a seguinte interpretação:

S (se (a > b) max = a; senão max = b;, *estado*)
= S (max = a;, *estado*) se S (a > b, *estado*) for *verdadeiro*
= S (max = b;, *estado*) caso contrário

Usando a definição de S(*Expressão, estado*), vemos que S(a > b, *estado*) é verdadeiro nesse estado em particular, de modo que o significado desse *Condicional* é o mesmo daquele da primeira *Atribuição*, ou S(max = a;, *estado*). Considerando o que sabemos sobre o significado de *Atribuições*, vemos que o estado final se torna:

estado = {⟨a, 3⟩, ⟨b, 1⟩, ⟨max, 3⟩}

Bloco O significado de um *Bloco* é a função identidade (se não tiver comandos) ou o significado do resto do *Bloco* aplicado ao novo estado obtido pela computação do significado do primeiro comando do *Bloco.*

$$S\ (\textit{Bloco}\ \mathrm{b}, \textit{Estado, estado}) = \textit{estado} \qquad \text{se b} = \{\ \}$$
$$= S\ (\mathrm{b}_n, S\ (..., S\ (\mathrm{b}_1, \textit{estado})...)) \text{ se b} = \{b_1 b_2 ... b_n\}$$

Essa é uma maneira formal de expressar a Regra de Significado 8.6.

Por exemplo, analise o *Bloco* da Figura 7.2. Aqui está um resumo do seu significado para *estado* = {⟨i, 2⟩,⟨f, 2⟩,⟨n, 3⟩}, conforme definido pela função anterior.

$$\begin{aligned}
&S\ (\{\mathrm{i} = \mathrm{i} + 1; \mathrm{f} = \mathrm{f} * \mathrm{i};\}, \textit{estado})\\
&\quad = S\ (\{\mathrm{f} = \mathrm{f} * \mathrm{i};\}, S\ (\mathrm{i} = \mathrm{i} + 1;, \{\langle \mathrm{i}, 2\rangle, \langle \mathrm{f}, 2\rangle, \langle \mathrm{n}, 3\rangle\}))\\
&\quad = S\ (\{\}, S\ (\mathrm{f} = \mathrm{f} * \mathrm{i};, S\ (\mathrm{i} = \mathrm{i} + 1;, \{\langle \mathrm{i}, 2\rangle, \langle \mathrm{f}, 2\rangle, \langle \mathrm{n}, 3\rangle\})))\\
&\quad = S\ (\{\}, S\ (\mathrm{f} = \mathrm{f} * \mathrm{i}; ,\{\langle \mathrm{i}, 3\rangle, \langle \mathrm{f}, 2\rangle, \langle \mathrm{n}, 3\rangle\}))\\
&\quad = S\ (\{\}, \{\langle \mathrm{i}, 3\rangle, \langle \mathrm{f}, 6\rangle, \langle \mathrm{n}, 3\rangle\})\\
&\quad = \{\langle \mathrm{i}, 3\rangle, \langle \mathrm{f}, 6\rangle, \langle \mathrm{n}, 3\rangle\}
\end{aligned}$$

As primeiras duas linhas e a última nessa interpretação revelam o significado de um *Bloco*, enquanto as linhas restantes aplicam as duas atribuições ao estado original na ordem em que aparecem no *Bloco*. Isso ilustra que podemos explicar formalmente o que acontece quando alguma seqüência de comandos é executada.

Laço O significado de um *Laço* é a função identidade (se o `teste` boleano não for verdadeiro) ou o significado do mesmo laço quando aplicado ao estado resultante da execução do seu corpo uma vez (veja a Regra de Significado 8.5).

$$S\ (\textit{Laço l, Estado estado}) = S\ (l, S\ (l.\textit{corpo, estado})) \quad \text{se } S\ (l.\textit{teste, estado}) \text{ é } \textit{verdadeiro}$$
$$= \textit{estado} \qquad \text{caso contrário}$$

Por exemplo, considere o *Laço* da Figura 7.2. Está claro que os dois comandos que precedem o laço neste exemplo formam *estado* = {⟨*i*, 1⟩, ⟨f, 1⟩, ⟨n, 3⟩}. Lembrando o que sabemos sobre o significado do bloco {*i* = *i* + 1; *f* = *f* ∗ *i*;} do parágrafo anterior, podemos começar a análise como segue:

$$\begin{aligned}
&S\ (\text{enquanto (i<n)}\{\mathrm{i} = \mathrm{i} + 1; \mathrm{f} = \mathrm{f} * \mathrm{i};\}, \textit{estado})\\
&\quad = S\ (\text{enquanto (i<n)}\{\mathrm{i} = \mathrm{i} + 1; \mathrm{f} = \mathrm{f}*\mathrm{i};\}, S\ (\{\mathrm{i} = \mathrm{i} + 1; \mathrm{f} = \mathrm{f} * \mathrm{i};\}, \{\langle i, 1\rangle, \langle \mathrm{f}, 1\rangle, \langle \mathrm{n}, 3\rangle\}))
\end{aligned}$$

Isso significa que interpretamos o corpo uma vez no determinado estado e então passamos o estado resultante de volta para a função *S* para o próprio *Laço*. Disso resulta a seguinte série de transformações de estados:

$$\begin{aligned}
&S\ (\text{enquanto (i<n)}\{i = i + 1; f = f * i;\}, \textit{estado})\\
&\quad = S\ (\text{enquanto (i<n)}\{\mathrm{i} = \mathrm{i} + 1; \mathrm{f} = \mathrm{f} * \mathrm{i};\}, S\ (\{\mathrm{i} = \mathrm{i} + 1; \mathrm{f} = \mathrm{f} * \mathrm{i};\}, \{\langle i, 1\rangle, \langle \mathrm{f}, 1\rangle, \langle \mathrm{n}, 3\rangle\}))\\
&\quad = S\ (\text{enquanto (i<n)}\{\mathrm{i} = \mathrm{i} + 1; \mathrm{f} = \mathrm{f} * \mathrm{i};\}, S\ (\{\mathrm{i} = \mathrm{i} + 1; \mathrm{f} = \mathrm{f} * \mathrm{i};\}, \{\langle i, 2\rangle, \langle \mathrm{f}, 2\rangle, \langle \mathrm{n}, 3\rangle\}))\\
&\quad = S\ (\text{enquanto (i<n)}\{\mathrm{i} = \mathrm{i} + 1; \mathrm{f} = \mathrm{f} * \mathrm{i};\}, S\ (\{\mathrm{i} = \mathrm{i} + 1; \mathrm{f} = \mathrm{f} * \mathrm{i};\}, \{\langle i, 3\rangle, \langle \mathrm{f}, 6\rangle, \langle \mathrm{n}, 3\rangle\}))\\
&\quad = \{\langle i, 3\rangle, \langle \mathrm{f}, 6\rangle, \langle \mathrm{n}, 3\rangle\}
\end{aligned}$$

Para cada uma das três linhas intermediárias, a função *S* para *Bloco, Atribuição* e *Expressão* é usada para transformar o estado que é mostrado na linha abaixo dela. A última linha resulta do teste `i < n` se tornando *falso*, em cujo caso o estado é inalterado. Esta função é, claro, uma função parcial, já que um *Laço* pode ser escrito de forma a nunca parar. Nesse caso, deve ficar claro que a última linha em interpretações como a anterior nunca será alcançada.

8.4.4 Semântica Denotacional de Expressões

Lembre-se de que uma *Expressão* em Clite possui a seguinte sintaxe abstrata:

Expressão = *Variável* | *Valor* | *Binário* | *Unário*
Binário = *OpBinário* `op`; *Expressão* `termo1`, `termo2`
Unário = *OpUnário* `op`; *Expressão* `termo`
Variável = *String* `id`
Valor = *ValorInt* | *ValorBool* | *ValorFloat* | *ValorChar*

O significado de uma *Expressão* em um *Estado* depende do seu tipo:

S : *Expressão* $\times$ *Estado* $\rightarrow$ *Valor*
S (*Expressão e*, *Estado estado*)

$= e$ se e for um *Valor*
$= estado(e)$ se e for uma *Variável*
$= ApplyBinary(e.\text{op}, S(e.\text{termo1}, estado), S(e.\text{termo2}, estado))$ se e for um *Binário*
$= ApplyUnary(e.\text{op}, S(e.\text{termo}, estado)$ se e for um *Unário*

Esta função é uma formalização da Regra de Significado 8.7.

A função *ApplyBinary* calcula um *Valor* quando recebe um *Operador* binário e dois *Valores* que sejam seus operandos (veja a Regra de Significado 8.8).

ApplyBinary : *Operador* $\times$ *Valor* $\times$ *Valor* $\rightarrow$ *Valor*
ApplyBinary(*Operador op*, *Valor v1*, *Valor v2*)

$= v1 + v2$ se $op =$ `int+`
$= v1 - v2$ se $op =$ `int-`
$= v1 \times v2$ se $op =$ `int*`
$= floor\left(\left|\frac{v1}{v2}\right|\right) \times sign\,(v1 \times v2)$ se $op =$ `int/`
...

Esta função supõe que os operadores `int+`, `int-`, ... carregam seus significados costumeiros no domínio semântico dos inteiros **I**, números de ponto flutuante **F**, *strings* **S** e boleanos **B**.[5] Ela também supõe que os tipos de $v1$ e $v2$ devam ser os mesmos e concordar com o tipo requerido pelo operador. Assim como na Seção 8.2.3, mostramos apenas os primeiros casos de uma definição semântica bastante grande, dado o número de operadores distintos.

A função *ApplyUnary* calcula um *Valor* quando recebe um *Operador* unário e um *Valor* que é seu operando (veja a Regra de Significado 8.9).

5. Observe que a definição do operador `int/` é uma forma matemática de especificar o quociente inteiro de $v1$ e $v2$ (a implementação Java é mais simples de expressar).

$$
\begin{aligned}
&\mathit{ApplyUnary} : \mathit{Operador} \times \mathit{Valor} \rightarrow \mathit{Valor} \\
&\mathit{ApplyUnary}\ (\mathit{Operador}\ op,\ \mathit{Valor}\ v) = \neg v \quad \text{se } op = \texttt{!} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad = -v \quad \text{se } op = \texttt{int-} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad = -v \quad \text{se } op = \texttt{float-} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \ldots
\end{aligned}
$$

em que o tipo de v deve concordar com o tipo requerido pelo operador. Assim como na Seção 8.2.3, mostramos apenas os primeiros casos de uma definição de semântica bastante grande, dado o número de operadores distintos.

Para ilustrar como essas funções atuam, analise novamente a expressão `x+2*y` cuja sintaxe abstrata é mostrada na Figura 8.2. Suponha que essa expressão seja avaliada no estado *estado* $= \{\langle x, 2\rangle, \langle y, -3\rangle, \langle z, 75\rangle\}$, ou seja, queremos usar essas definições funcionais para mostrar que:

$$S\ (\mathrm{x} + 2 * \mathrm{y}, \{\langle x, 2\rangle, \langle y, -3\rangle, \langle z, 75\rangle\} = -4$$

Já que esta expressão é um *Binário*, temos:

$$
\begin{aligned}
S\ (\mathrm{x} + 2 * \mathrm{y}, \{\langle x, 2\rangle, \langle y, -3\rangle, \langle z, 75\rangle\} &= \mathit{ApplyBinary}(+, A, B) \\
\text{em que } A &= S\ (x, \{\langle x, 2\rangle, \langle y, -3\rangle, \langle z, 75\rangle\}) \\
\text{e } B &= S\ (2 * \mathrm{y}, \{\langle x, 2\rangle, \langle y, -3\rangle, \langle z, 75\rangle\})
\end{aligned}
$$

Agora o significado de A é o valor de x no *estado* $= \{\langle x, 2\rangle, \langle y, -3\rangle, \langle z, 75\rangle\}$, ou 2, já que x é uma *Variável*. O significado do *Binário* B, entretanto, vem de outra aplicação da função S, que é:

$$
\begin{aligned}
S\ (2 * \mathrm{y}, \{\langle x, 2\rangle, \langle y, -3\rangle, \langle z, 75\rangle\} &= \mathit{ApplyBinary}(*, C, D) \\
\text{em que } C &= S\ (2, \{\langle x, 2\rangle, \langle y, -3\rangle, \langle z, 75\rangle\}) \\
\text{e } D &= S\ (\mathrm{y}, \{\langle x, 2\rangle, \langle y, -3\rangle, \langle z, 75\rangle\})
\end{aligned}
$$

Aqui, o significado de C é 2, já que 2 é um *Valor* e o significado de D é -3, já que y é uma *Variável* no *estado* $= \{\langle x, 2\rangle, \langle y, -3\rangle, \langle z, 75\rangle\}$.

Com essas informações, a definição de *ApplyBinary* nos dá o significado de $2 * \mathrm{y}$:

$$
\begin{aligned}
S\ (2 * \mathrm{y}, \{\langle x, 2\rangle, \langle y, -3\rangle, \langle z, 75\rangle\}) &= \mathit{ApplyBinary}\ (*, 2, -3) \\
&= -6
\end{aligned}
$$

Assim, o significado da nossa expressão original se revela o seguinte:

$$
\begin{aligned}
S\ (\mathrm{x} + 2 * \mathrm{y}, \{\langle x, 2\rangle, \langle y, -3\rangle, \langle z, 75\rangle\}) &= \mathit{ApplyBinary}\ (+, 2, -6) \\
&= -4
\end{aligned}
$$

Observe que este exemplo usa apenas as funções *ApplyBinary* e *S* definidas anteriormente, junto às propriedades matemáticas da aritmética de números inteiros, para derivar esse resultado. Deve ficar claro para os leitores que os significados de *Expressões* mais complexas com operações binárias e operandos inteiros são definidos integralmente por essas funções.

Isso completa nosso tratamento formal da semântica para a linguagem Clite. Uma implementação completa em Java da sintaxe Clite, sistema de tipos e semântica podem ser encontrados no site do livro indicado no Prefácio. Essas ferramentas podem ser usadas para exercitar as funções discutidas neste capítulo com diversos exemplos de programas em Clite.

8.4.5 Limites de Modelos Semânticos Formais

A semântica formal é difícil de aplicar a algumas construções das linguagens, como transferências de controle (os comandos `goto`, `break` e `continue`), geração e captura de exceções e operações dinâmicas de memória relacionadas com chamadas de funções e criação de objetos.

Em semântica denotacional, transferências de controle são manipuladas por um dispositivo conhecido como uma *continuação* (veja, por exemplo, Tennet, 1981), cuja cobertura está além do escopo deste texto.

Um recurso de linguagem que correntemente não pode ser definido por um modelo semântico denotacional é a concorrência. Em semântica denotacional, o significado de um *Programa* é um estado. Já que significado é uma função, dado o mesmo estado de entrada e o mesmo programa, o resultado é sempre o mesmo estado de saída. Entretanto, o significado de um programa concorrente pode ser um estado ligeiramente diferente cada vez que o programa for executado.

8.5 RESUMO

Este capítulo introduz a idéia de semântica, apresentando uma descrição semântica completa de Clite, primeiro em um estilo operacional e depois em um estilo denotacional formal. As vantagens e as limitações de modelos semânticos formais para linguagens de programação também são resumidas.

Muitos dos exercícios pedem que você execute ou modifique o interpretador Clite para explorar diversos recursos semânticos. Esse interpretador pode ser baixado do site do livro mencionado no Prefácio.

EXERCÍCIOS

8.1 Discuta a inclusão de operações $++$ e $--$ ao Clite. Suponha que elas sejam permitidas apenas como comandos isolados.
(a) Discuta as alterações necessárias para o verificador de tipos e o transformador de tipos.
(b) Acrescente novas regras para comandos que definam a semântica desses operadores.
(c) Modifique o interpretador Clite_D para implementar esses novos comandos.
(d) Discuta por que não adicionamos esses operadores a expressões em geral. Dica: pense em como isso altera o interpretador.

8.2 Adicione comandos de E/S `cin/cout` no estilo de C++ a Clite:
(a) Definindo sua sintaxe concreta,
(b) Definindo sua sintaxe abstrata,
(c) Modificando o *lexer* e o analisador para reconhecê-los e
(d) Adicionando esses comandos ao interpretador Clite_D.

8.3 Discuta as vantagens e as desvantagens de não permitir comandos de atribuição na forma `b = a`, em que `a` tenha o valor *undef*. Que alterações você teria de fazer nas regras de semântica e no interpretador de Clite para impor essa mudança?

8.4 Adicione o operador `%` ao Clite:
(a) Definindo sua sintaxe concreta,
(b) Definindo sua sintaxe abstrata,
(c) Modificando o *lexer* e o analisador para reconhecê-lo e
(d) Adicionando-o ao interpretador Clite_D.

8.5 Modifique a definição dos operadores && e || de modo que não usem avaliação de curto-circuito. Apresente as novas regras semânticas. Discuta as modificações necessárias para a implementação.

8.6 Modifique o interpretador Clite de modo que o estado corrente seja exibido imediatamente antes de cada comando ser executado. Crédito adicional: inclua o número do comando na exibição.

8.7 Modifique o interpretador Clite de modo que o término do programa devido a um erro identifique em qual linha esse erro ocorreu.

8.8 Usando a Tabela 7.3 como guia, construa um rastro de execução para o programa Clite:

```
1   void main ( ) {
2         int i, a, z;
3         i = 5;
4         a = 2;
5         z = 1;
6         while (i > 0) {
7           if (i - 1 / 2 * 2 == 1)
8                z = z * a;
9           i = i / 2;
10          a = a * a;
11      }
12  }
```

8.9 Para o programa do Exercício 8.8:
(a) Apresente a sintaxe abstrata do comando 8 como ela é produzida pelo analisador.
(b) Apresente a sintaxe abstrata do comando 8 como ela é produzida pelo transformador de tipos.
(c) De acordo com as regras de semântica da Seção 8.2.3, explique *precisamente* como o valor da expressão no comando 8 é calculado.

8.10 Modifique o interpretador Clite de modo que *overflows* de inteiros (conforme visto no Exercício 7.5) interrompam o interpretador com uma mensagem de erro apropriada. Demonstre que sua modificação funciona no programa fatorial.

8.11 Modifique o interpretador Clite de modo que as expressões com efeitos colaterais sejam suportadas. Use as definições da Seção 8.2.4 como guia.

8.12 Usando a Tabela 7.3 como guia, construa um rastro de execução para o programa Clite_D:

```
1   void main ( ) {
2         i = 5;
3         a = 2.0;
4         z = 1.0;
5         while (i > 0) {
6           if (i - 1 / 2 * 2 == 1)
7                z = z * a;
8           i = i / 2;
9           a = a * a;
10      }
11  }
```

8.13 Para o programa do Exercício 8.12, apresente:
(a) A sintaxe abstrata do comando 7 como ela é produzida pelo transformador de tipos.
(b) De acordo com as regras de semântica da Seção 8.2.3, explique **precisamente** como o valor da expressão é calculado.

8.14 Modifique o interpretador Clite_D de modo que conversões implícitas de inteiro para ponto flutuante sejam permitidas.

8.15 Suponha que $estado_1 = \{\langle x,1\rangle, \langle y,2\rangle, \langle z,3\rangle\}$, $estado_2 = \{\langle y,5\rangle\}$ e $estado_3 = \{\langle w,1\rangle\}$. Quais são os resultados das seguintes operações? Observação: o símbolo $\overline{\cup}$ denota o método `onion` introduzido na Seção 8.4.
(a) $estado_1 \overline{\cup} estado_2$
(b) $estado_1 \overline{\cup} estado_3$
(c) $estado_2 \overline{\cup} estado_3$
(d) $\phi \overline{\cup} estado_2$
(e) $estado_1 \otimes estado_3$
(f) $estado_2 \otimes estado_3$
(g) $(estado_1 - (estado_1 \otimes estado_3) \cup estado_3)$

8.16 Mostre como o significado de cada uma das expressões a seguir e os estados dados são derivados das regras de semântica apresentadas na Seção 8.2.3. (A sintaxe abstrata de cada uma dessas expressões foi desenvolvida como um exercício no Capítulo 2.)
(a) S ((z + 2) * y, $\{\langle x,2\rangle, \langle y,-3\rangle, \langle z,75\rangle\}$)
(b) S (2 * x + 3 / y − 4,$\{\langle x,2\rangle, \langle y,-3\rangle, \langle z,75\rangle\}$)
(c) S (1, $\{\langle x,2\rangle, \langle y,-3\rangle, \langle z,75\rangle\}$)

8.17 Mostre todas as etapas da derivação do significado do seguinte comando de atribuição, quando executado em um determinado estado, usando as regras de semântica apresentadas na Seção 8.2.3.

$$S\,(\text{z} = 2 * \text{x} + 3 / \text{y} - 4,\{\langle x,6\rangle, \langle y,-12\rangle, \langle z,75\rangle\})$$

8.18 Complete as regras de semântica para operadores aritméticos, boleanos e lógicos para Clite que foram introduzidos na Seção 8.2.3.

8.19 Modifique as funções semânticas denotacionais para comandos e expressões conforme for necessário para não permitir uma atribuição `b = a`, na qual `a` possui o valor *undef*.

8.20 Modifique as funções semânticas denotacionais para expressões de modo que os efeitos colaterais sejam suportados. Use as definições da Seção 8.2.4 como guia.

8.21 Apresente uma função semântica denotacional para os operadores `++` e `--`, conforme discutido no Exercício 8.1.

8.22 Apresente uma função semântica denotacional para os operadores `cin` e `cout` conforme discutido no Exercício 8.2.

9 Funções

"É melhor fazer cem funções operarem sobre uma estrutura de dados do que dez funções sobre dez estruturas de dados."

Alan Perlis

VISÃO GERAL DO CAPÍTULO

Funções são elementos fundamentais em toda linguagem de programação, já que são ferramentas essenciais para abstração em programação. Em linguagens diferentes, as funções são conhecidas variavelmente como procedimentos, sub-rotinas, subprogramas ou métodos, e possuem diversas características em comum, assim como algumas diferenças importantes nas mais variadas linguagens.

O comportamento das funções é altamente relacionado com o gerenciamento dinâmico de memória. As características de uma estrutura de memória fundamental, a pilha, ajudam a compreendermos melhor como a memória é organizada para implementar funções, especialmente funções recursivas.

Este capítulo discute os elementos-chave de funções em linguagens de programação, incluindo chamada e retorno, parâmetros e argumentos e recursão. A implementação de funções é discutida no Capítulo 10.

9.1 TERMINOLOGIA BÁSICA

Fortran e Ada fazem distinção entre funções que retornam um valor e aquelas que não retornam. As primeiras são denominadas *funções,* e as últimas, *sub-rotinas* em Fortran e *procedimentos* em Ada. Linguagens do tipo de C não fazem tal distinção; ambas são chamadas de funções. Em C++ e Java, um *método* é uma função declarada dentro de uma classe.

Alguns têm argumentado que funções deveriam ser como funções matemáticas. Elas não deveriam ter efeitos colaterais e seu único valor de retorno deveria ser o valor da própria função. Chamadas para funções, como em matemática, aparecem como parte de uma expressão. Um exemplo de Fortran é a fórmula para calcular uma solução `x` para uma equação quadrática, $ax^2 + bx + c$:

```
x= (b*b - sqrt(4*a*c))/(2*a)
```

Nesse caso, a função `sqrt` calcula a raiz quadrada do seu argumento e não possui efeitos colaterais.

Alguns também têm argumentado que sub-rotinas e procedimentos não devem retornar um valor, exceto modificando um ou mais dos seus argumentos. Uma sub-rotina ou um procedimento também pode ler ou gravar saída ou modificar uma ou mais variáveis globais, que são os efeitos colaterais. Em linguagens tipo C, um procedimento é uma função com um tipo de retorno `void`. Em contraste, Fortran e Ada têm uma sintaxe distinta para funções e procedimentos.

Chamadas a procedimentos aparecem como declarações separadas em vez de como partes de uma expressão. Mesmo em C, uma chamada a função pode aparecer como uma declaração separada, em cujo caso seu tipo de retorno normalmente é `void`. Um exemplo de uma chamada C aparece a seguir:

```
strcpy(s1, s2);
```

Essa chamada copia uma string de caracteres de `s1` para `s2`.

Deixando os argumentos anteriores à parte, a maioria das linguagens orientadas a objeto e imperativas não tentam impor a regra do efeito colateral em funções. Mantendo-se a convenção de C, usaremos o termo *função* para denotar tanto procedimentos quanto funções.

9.2 CHAMADA E RETORNO DE FUNÇÕES

O programa na Figura 9.1, primeiramente introduzido no Capítulo 4, pode ilustrar a dinâmica de chamada e o retorno de funções. Nesse programa, `main` é inicialmente chamado e as variáveis globais `h` e `i` estão disponíveis para ele, junto às variáveis locais `a` e `b`. Após as três primeiras declarações de atribuição em `main` terem sido executadas, as variáveis `h`, `a` e `b` têm valor 5, 3 e 2, respectivamente.

```
int h, i;
void B(int w) {
   int j, k;
   i = 2*w;
   w = w+1;
}
void A(int x, int y) {
   bool i, j;
   B(h);
}
int main( ) {
   int a, b;
   h = 5; a = 3; b = 2;
   A(a, b);
}
```

| Figura 9.1 Exemplo de Programa C/C++

Quando `A` é chamada por `main`, seus parâmetros `x` e `y` obtêm os valores 3 e 2, e `A` também tem acesso à variável global `h`. As variáveis locais de `A`, `i` e `j` também são acessíveis nesse ponto, assim como a variável global `h`. Entretanto, a variável global `i` não é acessível a `A`, conforme destacado na discussão sobre escopos no Capítulo 4.

Quando `B` é chamada por `A`, o parâmetro `w` se torna acessível (com o valor 5), assim como as variáveis locais `j` e `k` e as globais `h` e `i`. Dessa forma, a primeira atribuição dentro de `B` atribui o valor 10 para a variável global `i`.

O controle retorna de uma função chamada de uma entre duas formas. Para uma função ou um procedimento `void`, o controle retorna quando o final do corpo da função é atingido ou quando uma declaração `return` é encontrada. Para uma função não-`void`, o controle retorna quando uma declaração `return` é encontrada. Nesse caso, a declaração `return` tem uma expressão que designa o valor a ser retornado.

Funções em Pascal designam um resultado retornado por meio de uma declaração de atribuição cuja variável-alvo é identicamente o nome da própria função. Funções em Haskell designam o resultado gravando uma expressão. Muitos exemplos do estilo Haskell de escrita de funções aparecem no Capítulo 14.

9.3 PARÂMETROS

Funções derivam sua grande utilidade tanto em matemática quanto em linguagens de programação pela sua capacidade de receber parâmetros. Por exemplo, uma função de raiz quadrada não seria muito útil se não pudesse receber um argumento que especificasse o valor cuja raiz quadrada devesse ser calculada.

> **Definição:** Uma expressão que aparece em uma chamada de função ou em um procedimento é denominada *argumento*.

Definição: Um identificador que aparece em uma declaração de função ou em um procedimento é denominado *parâmetro*.

A tradição Algol/Pascal é usar o termo *parâmetro real* para uma expressão na chamada e *parâmetro formal* para um parâmetro em uma declaração de função. A tradição C/Java é usar os termos *argumento* para o primeiro e *parâmetro* para o segundo. Alguns textos usam parâmetros como sinônimo de argumento. Este texto segue a tradição C/Java.

Seguindo a convenção matemática, a maioria das linguagens de programação associa argumentos na chamada de funções com parâmetros na declaração de funções pela posição. Assim, na Figura 9.1, a chamada em `main` para a função `A`:

```
A(a, b)
```

associa o argumento `a` ao parâmetro `x` e o argumento `b` ao parâmetro `y`, e nunca o contrário.

Exceto para nomes de funções sobrecarregadas (veja o Capítulo 4), os argumentos devem corresponder em número e tipo. Exceções a essa regra ocorrem nas linguagens Perl, Ada e Smalltalk.

Perl não permite a declaração de parâmetros em um cabeçalho de função. Em vez disso, ela disponibiliza os parâmetros em uma matriz chamada `@_`. Os parâmetros podem ser acessados por posição usando essa matriz. Essa convenção foi adaptada do *script* do shell Unix. Perl é discutida na Seção 12.7.

Como alternativas à correspondência estrita por posição, Ada fornece dois mecanismos. No primeiro, os parâmetros podem ser declarados com valores-padrão:

```
function sum(list : realist;
   length : integer := list'length) return real is
   ...
```

Nesse exemplo, se o segundo parâmetro for omitido, ele passa para o comprimento declarado do primeiro parâmetro. Assim que um parâmetro é declarado com um valor-padrão, todos os parâmetros depois dele também devem ter valores-padrão. Essa característica é desejável porque ajuda a reduzir o número de argumentos em uma chamada. Um número de funções de bibliotecas científicas em linguagens anteriores, principalmente Fortran, tinha contadores de parâmetros de dígito duplo.

O outro mecanismo fornecido por Ada permite que os argumentos tenham uma ordem diferente da usada pelos nomes dos parâmetros na chamada. Na Figura 9.1, a chamada de `A` em `main` pode ser escrita de qualquer uma das seguintes formas:

```
A(a, b);
A(a, y => b);
A(x => a, y => b);
A(y => b, x => a);
```

Esse mecanismo é mais útil quando os nomes de parâmetros forem identificadores com significado.

Smalltalk (veja a Seção 13.3) fornece um mecanismo interessante. Uma chamada do método `at:put` em uma matriz `a` é escrita:

```
a at: i put: x + y
```

Supondo que dois-pontos seja um identificador válido, o equivalente em Java seria:

```
a.at:put:(i, x+y);   // significando que a[i] = x+y
```

Falando estritamente, isso não se desvia da regra de *correspondência por posição*, já que o método `put:at:` pode não existir ou pode ter uma semântica diferente.

9.4 MECANISMOS DE PASSAGEM DE PARÂMETROS

Os *valores* associados aos parâmetros durante a vida de uma função chamada são determinados com base em como esses argumentos são *passados* para a função pela chamada. Há cinco formas principais de passar um argumento para uma função:

1. Por *valor,*
2. Por *referência,*
3. Por *resultado-valor,*
4. Por *resultado* e
5. Por *nome*.

Entre esses, os dois primeiros são os mais proeminentes.

9.4.1 Passagem por Valor

A passagem por valor (às vezes chamada de avaliação *ávida*) geralmente ocorre quando um argumento fornece um valor de *entrada* para uma função chamada.

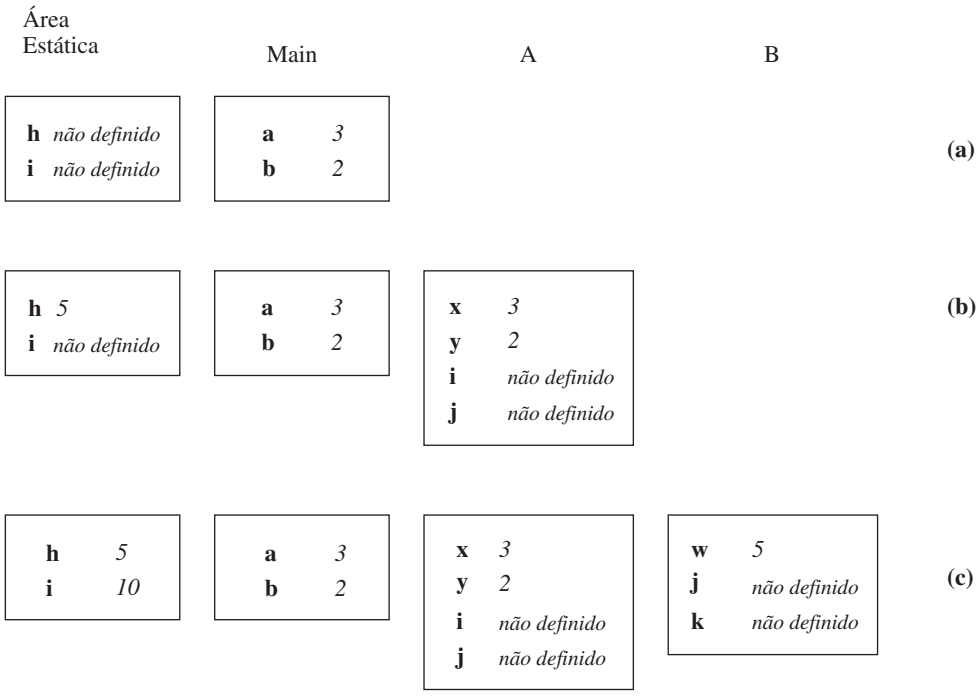
> **Definição:** Passar um argumento *por valor* significa que o valor do argumento é calculado no tempo da chamada e copiado para o parâmetro correspondente.

Usando o programa do exemplo da Figura 9.1, a passagem por valor é ilustrada na Figura 9.2, tanto na chamada `A(a,b)` de `main` (parte (b)) quanto na chamada `B(x)` de `A` (parte (c)). Nessas chamadas, cada um dos argumentos `a`, `b` e `x` é passado por valor. Observe que o rastro fornecido mostra valores 3 e 2 para os parâmetros `x` e `y` de `A`. Agora todas as referências a `x` e `y` de dentro de `A` usam essas cópias dos valores de argumentos `a` e `b`, de modo que seus valores originais são efetivamente protegidos contra gravação durante a chamada.

Assim, a seguinte função de troca C/C++ não trocará os valores dos argumentos associados aos parâmetros `a` e `b`:

```
void swap (int a, int b) {
   int temp = a;
   a = b;
   b = temp;
}
```

Embora os valores dos parâmetros `a` e `b` sejam de fato trocados, esses valores não são *copiados* para os argumentos. Assim, a passagem por valor é, muitas vezes, chamada de mecanismo de *cópia*.



| Figura 9.2 Passando Argumentos por Valor

Em C, uma função de troca pode ser escrita com o uso de ponteiros, da seguinte forma:

```
void swap(int *a, int *b) {
   int temp = *a
   *a = *b;
   *b = temp;
}
```

Nesse caso, os valores dos parâmetros são ponteiros para outras variáveis. Considere a chamada:

```
swap(&x, &y);
```

Os parâmetros `a` e `b` contêm os endereços `&x` e `&y`, respectivamente. Devido ao fato de os ponteiros serem explicitamente desreferenciados, como no exemplo

```
*a = *b;
```

os valores dos argumentos `x` e `y` são trocados.

Assim como em C, todos os argumentos em Java são passados por valor. O valor de uma variável para os assim chamados *tipos de referência* (matrizes e objetos de classes) na verdade é um endereço em vez de um valor comum (como seria o caso para int, float, char etc.). Argumentos que sejam tipos de referência são passados por intermédio de cópias de suas referências (endereços) para o parâmetro em vez do seu valor.

Assim, a seguinte função de troca em Java não substituirá os valores dos argumentos associados aos parâmetros `a` e `b`:

```
void swap(Object a, Object b) {
   Object temp = a;
   a = b;
   b = temp;
}
```

O raciocínio é basicamente o mesmo que em C/C++.

Entretanto, a seguinte função `swap` em Java trocará os valores das entradas de índices *i* e *j* na matriz associada ao parâmetro `A`.

```
void swap(Object[ ] A, int i, int j) {
   Object temp = A[i]
   A[i] = A[j];
   A[j] = temp;
}
```

9.4.2 Passagem por Referência

A passagem por referência ocorre quando um argumento fornece uma variável de *saída* à qual a função chamada pode atribuir um resultado.

> **Definição:** Passar um argumento *por referência* (ou *por endereço*) significa que o *endereço de memória* do argumento é copiado para o parâmetro correspondente, de modo que o parâmetro se torna uma *referência* (ponteiro) indireta ao argumento real.

Assim, todas as atribuições ao parâmetro formal dentro da vida da chamada afetam diretamente o valor do argumento. Por exemplo, suponha que o programa da Figura 9.1 tivesse o argumento para o parâmetro `w` de `B` passado por referência em vez de por valor. Agora, a atribuição

```
w = w+1;
```

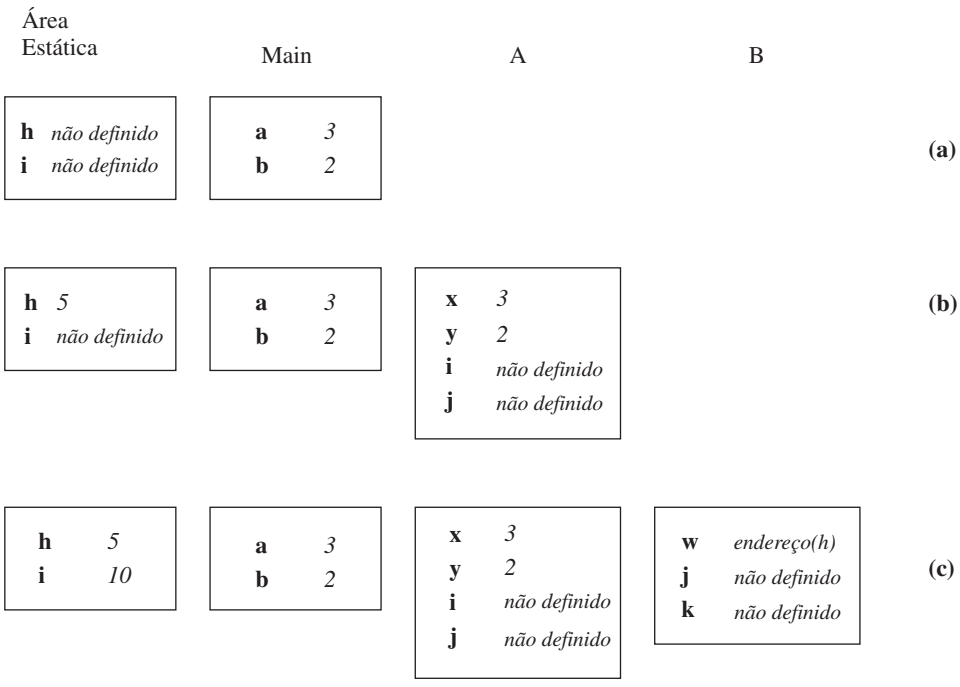
dentro de `B` irá incrementar diretamente o valor de `h`, em vez de uma cópia desse valor, dentro da variável `w` de `B`.

Uma forma de se entender a passagem por referência é visualizando como um compilador poderia traduzir o programa da Figura 9.1 para C, conforme mostrado na Figura 9.3. Nesse exemplo, o endereço de `h`, a saber, `&h`, é passado para a variável ponteiro `*w`. Dentro da função `B`, todas as ocorrências de `w` são desreferenciadas explicitamente como `*w`. Assim, na passagem por referência o argumento `h` é potencialmente alterado por qualquer atribuição ao parâmetro `w`. Devido à desreferenciação na versão "compilada", não é possível mudar para onde `w` aponta. Um rastro da versão "compilada" do programa é apresentado na Figura 9.4.

Em linguagens que suportam mecanismos de passagem de múltiplos parâmetros, a distinção entre um parâmetro por valor e um de referência deve se tornar explícita dentro da declaração da função. Por exemplo, em C++, é usado um & para indicar um parâmetro por referência e a sua ausência indica um parâmetro por valor. Ou seja, se

```
int h, i;
void B(int* w) {
    int j, k;
    i = 2*(*w);
    *w = *w+1;
    ...
}
void A(int* x, int* y) {
    bool i, j;
    B(&h);
    ...
}
int main( ) {
    int a, b;
    h = 5; a = 3; b = 2;
    A(&a, &b);
    ...
}
```

| Figura 9.3 Exemplo de Passagem por Referência



| Figura 9.4 Exemplo de Passagem de um Argumento por Referência

a função B da Figura 9.1 fosse escrita em C++, sua primeira linha seria alterada da seguinte maneira, para indicar um parâmetro por referência:

```
void B(int& w) {
```

e a chamada de A permaneceria B(h).

Essa declaração de w como um parâmetro por referência em B requer que o argumento em qualquer chamada tenha um endereço, ou seja, uma variável, matriz ou estrutura, por exemplo, mas não uma expressão com operadores ou um literal.

Os argumentos são passados por referência nos casos em que o valor real do argumento deva mudar durante a vida da chamada. A passagem por referência também é preferida quando o argumento passado ocupa muita memória, como uma matriz grande. Nesses casos, o *overhead* de espaço e tempo necessário para criar uma cópia completa do objeto, se ele fosse passado por valor, é evitado pela cópia de um único endereço para o parâmetro.

A maior desvantagem da passagem por referência é que os efeitos colaterais sobre o valor original do argumento passado podem não ser desejados por quem realizou a chamada. Por exemplo, se a chamada de B por A com argumento h não quiser que o valor de h seja alterado durante a vida dessa chamada, deve ser evitada a passagem de h por referência.

Fortran usa exclusivamente a passagem por referência, o que causou alguns efeitos colaterais interessantes em versões mais antigas de Fortran. Por exemplo, a sub-rotina sub de Fortran, que adiciona 1 ao seu parâmetro p, junto à chamada call sub(1), poderia deixar o valor da "constante" 1 alterado para 2 por todo o resto da execução do programa. Compiladores Fortran modernos avaliam cada parâmetro real que não pode ser interpretado como um valor l e o atribuem a uma variável temporária; o endereço da temporária é então passado para o parâmetro.

9.4.3 Passagem por Resultado-valor e Resultado

A passagem por resultado-valor é usada quando um argumento fornece tanto *entrada para* quanto *saída da* função chamada.

> **Definição:** Um argumento passado por *resultado-valor* é implementado por meio da cópia do valor do argumento para o parâmetro no início da chamada e depois por meio da cópia do resultado calculado de volta para o argumento correspondente no final da chamada.

Por ser um mecanismo *copy-in-copy-out*, a passagem por resultado-valor combina a *passagem por resultado* com a *passagem por valor.*

> **Definição:** Um argumento passado por *resultado* é implementado pela cópia do valor final, calculado para o parâmetro de saída, para o argumento no final da vida da chamada.

Alguns compiladores Fortran passam variáveis não-matrizes usando a passagem por resultado-valor. As passagens por referência e por resultado-valor são equivalentes na falta de *aliasing*.

> **Definição:** *Aliasing* ocorre quando, dentro de uma função ou de um procedimento, a mesma localização de memória pode ser acessada com o uso de diferentes nomes.

Exemplos de *aliasing* incluem:

- Passar uma variável como parâmetro e referenciá-la como uma global dentro da mesma função.
- Passar uma variável como argumento duas ou mais vezes com o uso de qualquer mecanismo de passagem de parâmetro que não seja por valor.

- Ter duas ou mais referências (por exemplo, ponteiros) para a mesma localização.

Como uma comparação da passagem por referência *versus* passagem por resultado-valor, analise as seguintes funções semelhantes em C++ e Ada:

```
void f (int& x, int& y)
{
   x = x + 1;
   y = y + 1;
}
```

```
procedure f (x, y: in out Integer) is
begin
   x := x + 1;
   y := y + 1;
end f;
```

A chamada `f(a, b)` possui o efeito, em ambos os casos, de incrementar tanto `a` quanto `b` por um. Entretanto, a chamada `f(a, a)` deixará o valor *a priori* de `a` incrementado por 2 no caso de C++ (passagem por referência) e por 1 (passagem por valor) no caso de Ada.[1]

Entretanto, assim como em muitas outras questões, os padrões da maioria das linguagens especificam que tais programas são inválidos. Embora casos simples como o recém-citado possam ser detectados por muitos compiladores, o problema de *aliasing* é, de modo geral, intratável.

9.4.4 Passagem por Nome

A passagem por nome é um exemplo de *associação tardia*, pelo fato de a expressão na chamada não ser avaliada até que seja usada pela função.

> **Definição:** Um argumento passado por *nome* se comporta como se fosse substituído textualmente para cada ocorrência do parâmetro.

Uma visão de implementação da passagem por nome é uma macroexpansão, efetivamente o mecanismo `#define` de C/C++, mas implementada para cada chamada, em vez de em tempo de compilação, como é usado por `#define`.

A passagem por nome foi usada originalmente em Algol 60. Em casos simples, era um mecanismo ineficiente para se obter chamada por referência. Infelizmente, para Algol 60 isso tinha diversas conseqüências não previstas.

Uma dessas conseqüências era a incapacidade de escrever um procedimento completamente geral de troca:

```
procedure swap (a, b);
   integer a, b;
begin integer t;
   t := a;
   a := b;
   b := t
end;
```

1. Tecnicamente, um parâmetro `in out` Ada pode ser implementado como passagem por referência ou por resultado-valor. Nesse exemplo, estamos supondo que a implementação seja resultado-valor.

A chamada `swap(a[i], i)` troca os dois valores, mas a chamada `swap(i, a[i])` não troca, porque a segunda declaração de atribuição muda o valor de `i`.

Ainda mais interessante é o dispositivo de Jensen (Knuth, 1967). Analise a seguinte função de soma:

```
procedure sigma (a, j, m) : integer;
   integer a, j, m;
begin integer s;
   s := 0; j := 1;
   while (j <= m) do
   begin
      s := s + sigma;
      j := j + 1;
   end;
   sigma := s;
end;
```

À primeira vista, parece ser uma forma custosa de se calcular a quantidade $m \times a$.

Analise, entretanto, cada uma das seguintes chamadas:

```
s := sigma (z, i, n);
s := sigma (x[i], i, n);
s := sigma (x [i] * y[i], i, n);
```

Essas chamadas calculam, respectivamente:

$n \times z$

$x_1 + \ldots + x_n$

$x_1 \times y_1 + \ldots + x_n \times y_n$

Com o advento da programação estruturada (veja o Capítulo 7, Seção 7.5.4), um prêmio foi colocado sobre a escrita de código limpo e facilmente compreensível. O dispositivo de Jensen forneceu evidência clara de que as funções que usam passagem por nome não poderiam ser totalmente compreendidas, independentemente de chamadas sobre essas funções. Nenhum dos sucessores de Algol usou a passagem por nome.

Entretanto, a associação tardia pode ser implementada com o uso de um mecanismo chamado *lazy evaluation*, que é muito importante em linguagens funcionais. Essas idéias são discutidas no Capítulo 14.

9.4.5 Passagem de Parâmetros em Ada

Entre todas as linguagens discutidas neste livro, Ada é a única que não especifica como parâmetros são passados, mas, sim, como são usados: in, out ou in-out. Em Ada um *parâmetro in* é um parâmetro que é referenciado, mas não modificado. Como tal, podem ser usadas tanto a passagem por valor quanto a por referência, dependendo do tamanho do argumento.

Em contraste, um *parâmetro out* é um parâmetro cujo valor deve ser estabelecido pela rotina chamada antes que possa ser referenciado. Em especial, a rotina chamada deve atribuir um valor a um *parâmetro out* antes de referenciá-lo ou retornar. Outra forma de visualizar um parâmetro *out* é o fato de sua entrada nunca ser referenciada. Um parâmetro *out* pode ser implementado usando chamada por resultado ou por referência, dependendo do seu tamanho.

Finalmente, um parâmetro *in-out* é um parâmetro cujo valor pode ser referenciado antes que seja atribuído e ser reatribuído antes de retornar. *Parâmetros in-out* podem ser implementados com o uso da passagem por resultado-valor para tipos primitivos de dados ou passagem por referência para matrizes e registros.

Como esperado, o padrão da linguagem Ada especifica que um programa não é válido se o mecanismo de implementação real alterar o resultado do cálculo. Ada é a única linguagem discutida aqui que especifica o uso de um parâmetro, em vez do mecanismo de implementação.

9.5 REGISTROS DE ATIVAÇÃO

Para implementar uma chamada de função, a noção de um registro de ativação se tornou amplamente aceita. Esta seção resume as idéias básicas, em preparação para um tratamento mais completo da chamada de função do Capítulo 10.

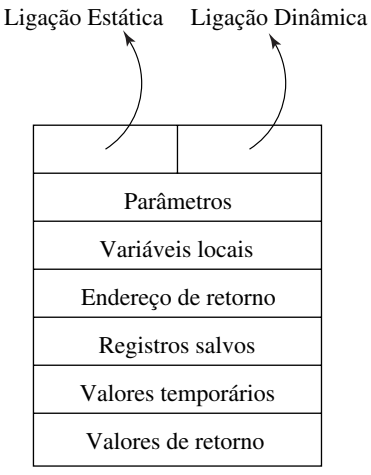
> **Definição:** Um *registro de ativação* é o bloco de informação associado a uma ativação de função, incluindo os parâmetros e as variáveis locais da função.

A estrutura básica de um *registro de ativação* é mostrada na Figura 9.5. Um registro de ativação individual possui espaço para:

- Os *parâmetros* e as *variáveis locais* da função chamada.
- Um endereço de retorno.
- Uma cópia dos *registros salvos* da função que chama.
- *Valores temporários*.
- O *valor de retorno* da função chamada, se houver algum.
- Um ponteiro de *ligação estática* para o pai estático da função.
- Um ponteiro de *ligação dinâmica* para o registro de ativação da função que chama.

A necessidade da maioria dos itens no registro de ativação deve ser óbvia. Por exemplo, o endereço de retorno é necessário de modo que a função chamada possa retornar apropriadamente para seu solicitante. A ligação dinâmica é usada para facilitar o fluxo de informações entre a função chamada e seu solicitante.

Figura 9.5 Estrutura de um Registro de Ativação de uma Função Chamada



A ligação estática geralmente só está presente naquelas linguagens que permitem que uma função seja aninhada estaticamente dentro de outra função. Por exemplo, C não precisa de uma ligação estática, já que não permite tal aninhamento. Entretanto, Algol, Pascal, Ada e Java requerem uma ligação estática, já que nessas linguagens o solicitante pode não ser o pai estático da função chamada. Sem a ligação estática, a função chamada não consegue localizar o registro de ativação do seu pai estático. Um exemplo da necessidade de ligação estática é apresentado na Seção 9.7.

Em linguagens que não suportam funções recursivas (por exemplo, Fortran), os registros de ativação podem ser alocados estaticamente. Fazer isso é muito eficiente no tempo, mas não necessariamente no espaço. A eficiência de tempo resulta do acesso direto tanto às variáveis locais quanto às globais. Entretanto, é ineficiente em espaço porque espaço é alocado para funções que podem nunca ser chamadas.

Mais comumente, registros de ativação são alocados dinamicamente em uma pilha de tempo de execução quando uma função é chamada e desalocados quando o retorno de uma função é executado. Em tais casos, o termo *frame de pilha* é comumente usado. Um *frame de pilha* é um sinônimo de registro de ativação quando este último é alocado e desalocado dinamicamente usando uma pilha. Na Seção 9.6, examinamos funções recursivas para motivar a necessidade de uma pilha em tempo de execução de registros de ativação.

9.6 FUNÇÕES RECURSIVAS

Funções recursivas têm uma implementação natural usando uma pilha de tempo de execução que registra o *status* de todas as chamadas ativas associando um registro de ativação a cada chamada. A maioria das linguagens atuais (C, C++, Java, Ada) suporta recursão, e de fato muitas linguagens têm a recursão como uma estrutura de controle central (Scheme, Haskell, Prolog). Analise a seguinte função recursiva em C++ que calcula o fatorial de um inteiro não-negativo `n`:

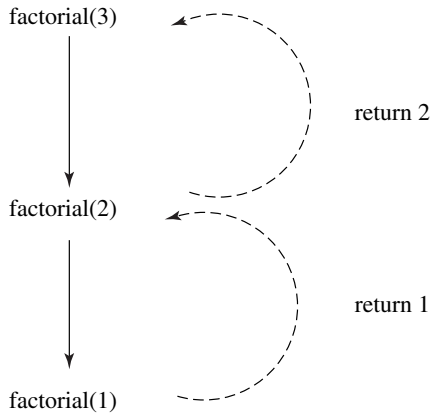
```
int factorial (int n) {
// calcula o fatorial de n, dado n >= 0
   if (n < 2)
      return 1;
   else
      return n*factorial(n-1);
}
```

Suponha que alguma função execute a chamada `factorial(n)`, na qual `n` possui valor igual a 3. Isso resultará em chamadas recursivas para calcular `factorial(2)` e `factorial(1)`, conforme apresentado na árvore de chamadas da Figura 9.6. Já que a chamada original usou uma variável, cujo valor poderia ter sido lido do teclado, um compilador não pode determinar quantos registros de ativação simultâneos podem ser necessários para uma determinada função recursiva.

A solução é alocar dinamicamente (empilhar) registros de ativação em uma pilha quando as funções forem chamadas e desalocá-los (desempilhá-los) quando as funções retornarem a seus solicitantes. A Figura 9.7a mostra um registro de ativação para a chamada `factorial(3)` sendo alocado (empilhado) em uma pilha de tempo de execução.

A parte do `else` inicia uma nova chamada, `factorial(2)`, fazendo com que mais um registro de ativação seja empilhado, conforme mostrado na Figura 9.7b. Nesse ponto, o registro de ativação para a primeira chamada permanece na pilha esperando que a segunda chamada seja completada.

Figura 9.6 Árvore de Chamada para `factorial(3)`



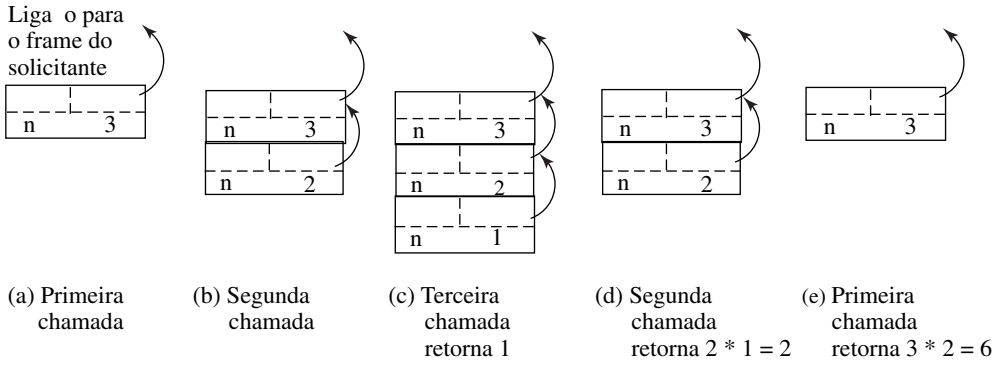
Entretanto, a segunda chamada inicia uma terceira, já que a condição `n < 2` ainda não está satisfeita, conforme mostrado na Figura 9.7c. Observe que os três registros de ativação agora na pilha guardam diferentes valores para o parâmetro `n`, já que cada chamada está efetivamente calculando um fatorial diferente.

Agora a terceira chamada retorna 1 para a segunda, que pode então completar o cálculo `2*factorial(1) = 2` e retornar 2 para a chamada original, conforme mostrado na Figura 9.7c e d. Nesse ponto, os dois registros do topo foram removidos (desempilhados) da pilha e a chamada original retorna `3*factorial(2) = 6` para o solicitante original, e seu registro de ativação é desempilhado.

Este exemplo ilustra como o conceito geral de pilha de tempo de execução é um modelo efetivo para funções recursivas. Cada chamada recursiva empilha um novo registro de ativação, e cada retorno desempilha esse registro de ativação quando retorna seu resultado para o solicitante.

9.7 PILHA DE TEMPO DE EXECUÇÃO

Uma forma natural para modelar a semântica de chamada e retorno de funções é usar uma pilha para armazenar o registro de ativação de cada função chamada.



| Figura 9.7 Atividade da Pilha para a Chamada Recursiva de `factorial(3)`

Um *push* de um registro de ativação ocorre sempre que uma função é chamada, e um *pop* também ocorre sempre que o controle retornar de uma função chamada.

> **Definição:** Uma *pilha de tempo de execução* é uma pilha de registros de ativação usada para modelar a semântica de chamadas e retornos de funções.

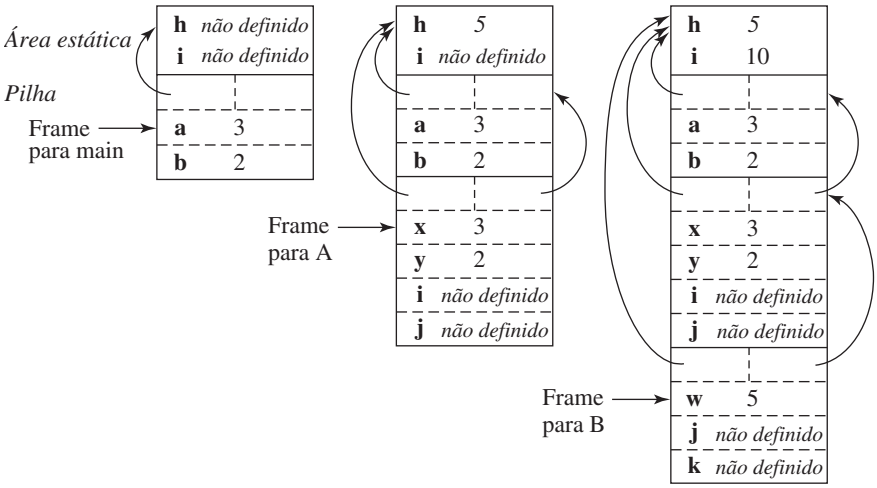
Quando uma função chamada retorna o controle à solicitante, seu registro de ativação é removido da pilha. Já que a função chamada pode chamar outra função (e geralmente o faz), muitos registros de ativação podem estar ativos na pilha de tempo de execução a qualquer momento. O fato de que registros de ativação são removidos na ordem inversa à qual são empilhados fornece um modelo consistente com as idéias fundamentais de escopo e visibilidade.

Analise o programa C/C++ na Figura 9.1 cuja pilha de tempo de execução pode ser modelada conforme mostrado na Figura 9.8. Aqui, vemos à esquerda o único registro de ativação no topo das globais `h` e `i` quando a função `main` é chamada. O desenho do meio mostra a pilha quando a função `A` é chamada de `main`.

O desenho à direita na Figura 9.8 mostra a pilha quando a função `B` é chamada por `A`. Três registros de ativação agora estão na pilha, refletindo que todas as três funções estão ativas. Cada registro de ativação contém uma ligação estática para as globais `h` e `i` e uma ligação dinâmica de volta para o registro de ativação para a função que o chamou. Cada registro de ativação também contém seus próprios parâmetros e as variáveis locais de função.

Finalmente, considere um programa em Ada apresentado na Figura 9.9. Analise a cadeia de chamadas: `Quicksort`, `Sort`, `Sort`, `Swap`. Mesmo sem desenhar explicitamente a pilha de tempo de execução, deve estar claro que a ligação dinâmica é inadequada para localizar o pai estático de `Swap`. Daí a necessidade do ponteiro *ligação estática*.

Java possui exatamente o mesmo problema com classes internas, que são comumente usadas para manipuladores de eventos que precisam de acesso a variáveis de instância de suas classes externas. Esses manipuladores de eventos podem ser chamados por alguma ação do usuário, como o clique em um botão. Sem a ligação estática, a pilha de tempo de execução precisaria ser pesquisada seqüencialmente para localizar o pai estático. Diversos exemplos de tais manipuladores de eventos ocorrem na classe `Controller` da Seção 16.3.3, que discute a interface gráfica em um *Jogo da Velha*.



| Figura 9.8 Pilha de Tempo de Registro de Ativação

```
procedure Quicksort(List : in out Intarray;
                    Length: Integer := List'Length) is
   procedure Swap(I, J : Integer) is
      Temp : Integer;
   begin
      Temp := List(I);
      List(I) := List(J);
      List(J) := List(I);
   end Swap;

   procedure Sort (M, N : Integer) is
      I, J, Key : Integer;
   begin
      if M < N then
         I := M; J := N + 1;
         Key := List(M);
         loop
           I := I + 1;
           exit when List(I) >= Key
         end loop;
         loop
           J := J - 1;
           exit when List(J) <= Key;
         end loop;
         Swap (I, J);
         Sort (M, J-1);
         Sort (J+1, N);
      end if
   end Swap;

begin
   Sort(1, Length);
end Quicksort;
```

| Figura 9.9 Necessidade de um Ponteiro de Ligação Estática

9.8 RESUMO

Este capítulo estuda o importante assunto de funções, seus mecanismos de passagem de parâmetros e registros de ativação. Uma pilha de tempo de execução facilita o gerenciamento normal em tempo de execução de variáveis locais e a passagem de parâmetros para chamadas de funções, assim como para recursão. Seu comportamento dinâmico é explorado mais inteiramente no Capítulo 10, no qual discutimos uma implementação completa de funções em Clite.

EXERCÍCIOS

9.1 Analise o seguinte programa em Clite:

```
void swap(int[] list, int i, int j) {
   int temp = list[i];
   list[i] = list[j];
   list[j] = temp;
}
void main () {
   int x[3] = {5, 2, 4};
   swap(x, 1, 2);
}
```

Qual é o valor final da matriz `x` para cada uma das seguintes suposições de passagem de parâmetro?

(a) O argumento `x` é passado por valor.
(b) O argumento `x` é passado por referência.
(c) O argumento `x` é passado por resultado-valor.

9.2 Usando o software disponível no site deste livro, exercite o interpretador Clite para o programa `functions.cpp`. Quais são os valores das variáveis globais `h` e `i` nesse programa quando cada uma das funções `main`, `A` e `B` começam a ser executadas?

9.3 No software disponível no site deste livro, exercite o programa `recFib.cpp`.

(a) Qual o valor do parâmetro `n` no programa `recFib.cpp` no registro de ativação mais acima da pilha cada vez que a função `fibonacci` é chamada?
(b) Quantos registros de ativação na pilha são ativados para a chamada `fibonacci(13)`?
(c) Você pode imaginar uma forma diferente de definir a função `fibonacci` de modo que menos registros de ativação na pilha sejam ativados?

9.4 O programa C/C++ a seguir resolve o problema das Torres de Hanói para três discos:

```
void moveTower (int disks, char start, char end,
                char temp) {
   if (disks == 1)
      cout << "Move a disk from " << start <<
              "to" << end << endl;
   else {
      moveTower (disks-1, start, temp, end);
      // move um disco do início para o fim
      cout << "Move a disk from " << start <<
              "to" << end << endl;
      moveTower (disks-1, temp, end, start);
   }
}
int main () {
   int totalDisks;
   totalDisks = 3;
   moveTower(totalDisks, 'A', 'B', 'C');
}
```

Usando seu compilador C++ favorito, rastreie os valores dos parâmetros `disks`, `start`, `end` e `temp` em todos os registros de ativação da pilha que sejam empilhados quando o programa é executado.

9.5 Modifique o programa da Figura 9.1 de modo que a função `B` não tenha efeitos colaterais na variável global `i`. Em vez disso, `B` deve retornar um valor `int`, que é o cálculo de `2*w`. Além disso, a chamada para `B` de `A` deve ser modificada de modo que esse resultado seja atribuído à variável global `i`. A declaração da variável global `i` dentro de `A` deve ser removida. Execute esse programa modificado usando seu compilador C++ favorito e responda às seguintes questões:

(a) Descreva graficamente (desenhe) a subárvore abstrata que representa sua chamada modificada para `B` de `A`.
(b) Descreva o registro de ativação que ocorre quando essa chamada é interpretada.
(c) Esse programa modificado possui o mesmo efeito global do programa original da Figura 9.1. O que teria acontecido se você não tivesse removido a declaração da variável local `i` de `A`?

9.6 A estratégia da pilha de tempo de execução discutida neste capítulo é especialmente robusta para suportar funções recursivas. Execute o interpretador Clite usando o programa `gcd.cpp`, que contém as seguintes funções recursivas para calcular o maior divisor comum de dois inteiros `x` e `y`:

```
int rem(int x, int y) {
return x - x/y*y;
}
int gcd(int x, int y) {
if (y==0) return x;
else return gcd(y, rem(x, y));
}
```

Quais registros de ativação são gerados pela chamada à função `gdc(24,10)`, e subseqüentes chamadas recursivas à `gdc`? Inclua em cada registro de ativação da pilha o valor de retorno, assim como os valores dos argumentos usados na respectiva chamada.

9.7 Analise o programa Ada na Figura 9.9.

(a) Mostre o conteúdo do novo *frame* de pilha quando a chamada `Quicksort(a)` é iniciada, para a matriz `a` = {4,2,5,1}.
(b) Mostre o conteúdo de cada novo *frame* de pilha adicionado quando cada uma das próximas quatro chamadas for iniciada (por exemplo, para `Sort`, para `Swap` e duas vezes para `Sort` novamente).

9.8 Como o procedimento de Ada da Figura 9.9 pode ser reescrito em C, que não permite aninhamento de funções? Que mecanismo de passagem de parâmetros é necessário na versão em C para o parâmetro `List` da função `Quicksort`?

10 Implementação de Funções

"Na teoria, não há diferença entre teoria e prática, mas não na prática."

Anônimo

VISÃO GERAL DO CAPÍTULO

As funções são elementos-chave da abstração procedimental em programação. Assim, projetistas e implementadores de linguagens devem tratá-las como uma preocupação central. Os projetistas de linguagem devem levar em consideração o projeto de funções, de modo que os programadores possam usá-las de forma efetiva, e também a implementação de funções, de modo que quem escreva compiladores possa implementá-las efetivamente e eficientemente.

Para ilustrar o projeto e a implementação de funções em profundidade, este capítulo desenvolve uma definição e uma implementação completas da sintaxe, sistema de tipos e semântica de uma extensão de Clite que permite que os programas tenham variáveis globais, declarações e chamadas de funções.

Como um estudo opcional, este capítulo também fornece uma definição formal para o sistema de tipos e a semântica de funções em Clite. Esse tratamento fornece aos leitores uma perspectiva do rigor matemático necessário para uma especificação completa de uma linguagem de programação moderna.

10.1 DECLARAÇÃO E CHAMADA DE FUNÇÕES EM CLITE

Nesta seção, enfocamos a sintaxe concreta e abstrata, sistema de tipos e semântica que incorporam o princípio da declaração, chamada, escopo e visibilidade de funções discutidos nos capítulos anteriores. A sintaxe de Clite fornece um ponto de partida para esta discussão.

Lembre-se do exemplo do programa em Clite da Figura 10.1, que foi primeiramente introduzido no Capítulo 4. Nesse programa, `main` é inicialmente chamado, e as variáveis globais `h` e `i` ficam acessíveis a ele, junto às variáveis locais `a` e `b`.

Após os três primeiros comandos de atribuição em `main` terem sido executados, as variáveis `h`, `a` e `b` possuem os valores 5, 3 e 2, respectivamente. Depois, `main` chama A e A chama B, cada chamada associando a si própria um registro ativador que contém um conjunto diferente de locais e parâmetros junto às globais `h` e `i`. Parte do nosso propósito neste capítulo é explicar como essas características semânticas podem ser implementadas no estabelecimento de uma pilha de tempo de execução.

10.1.1 Sintaxe Concreta

A sintaxe concreta dessa extensão de Clite permite que um *Programa* declare funções e variáveis globais além de uma função `main`. A metade superior da Figura 10.2 mostra essas regras de gramática que, quando adicionadas à gramática da Figura 2.7, definem a

```
int h, i;
void B(int w) {
     int j, k;
     i = 2*w;
     w = w+1;
}
void A(int x, int y) {
     bool i, j;
     B(h);
}
int main() {
     int a, b;
     h = 5; a = 3; b = 2;
     A(a, b);
}
```

| Figura 10.1 Exemplo do Programa em Clite

Programa → { *Identificador de Tipo FunçãoOuGlobal* } *FunçãoPrincipal*
Tipo → `int` | `boleano` | `float` | `char` | `void`
FunçãoOuGlobal → (*Parâmetros*) { *Declarações Comandos* } | *Global*
Parâmetros → [*Parâmetro* { , *Parâmetro* }]
Parâmetro → *Identificador de Tipo*
Global → { , *Identificador* };
FunçãoPrincipal → `int main ( )` { *Declarações Comandos* }

Comando → ; | *Bloco* | *Atribuição* | *ComandoSe*
ComandoEnquanto | *ComandoChamada* | *ComandoRetorno*
ComandoChamada → *Chamada*;
ComandoRetorno → return *Expressão*;
Fator → *Identificador* | *Literal* | (Expressão) | *Chamada*
Chamada → *Identificador* (*Argumentos*)
Argumentos → [*Expressão* { , *Expressão* }]

Figura 10.2 Regras de Gramática para a Extensão de Clite

sintaxe das funções e globais. Os sublinhados indicam adições às regras originais de Clite dadas no Capítulo 2.

Nessa definição, a *FunçãoPrincipal* é idêntica à noção original de qualquer número de declarações de função e/ou variável global para formar um *Programa* completo. O novo *Tipo* `void` designa uma função que não retorna resultado.[1]

O programa da Figura 10.1 segue essa nova sintaxe. Ele tem duas variáveis globais `h` e `i` e duas funções `A` e `B`, além da sua *FunçãoPrincipal* (*main*). As funções `A` e `B` têm ambas *Tipo* `void`, indicando que não retornam um resultado e são chamadas por comandos de *Chamada* independentes. A função `A` possui dois parâmetros `int` e `B` possui um.

A sintaxe de Clite permite que uma chamada de função apareça como um *ComandoChamada* separado ou uma *Chamada* alternativa dentro de um *Fator*. Neste último caso, o corpo da função chamada, que é um *Bloco*, deve conter um *ComandoRetorno*. A parte inferior da Figura 10.2 mostra as regras da gramática que realizam esses objetivos.

Por exemplo, o programa da Figura 10.1 possui dois *ComandosChamada*, um de `main` para `A` e um de `A` para `B`. O primeiro possui dois *Argumentos* `a` e `b` e o segundo possui um único *Argumento* `h`. Esse programa não possui *Chamadas* como instâncias de *Fator*, já que tanto `A` quanto `B` são funções void. Da mesma forma que em C++, um *ComandoRetorno* só pode aparecer em funções que não tenham *Tipo* void.

1. Pode ser intrigante ver que a função `main` não é `void`, já que ela parece não retornar resultados. Isso é consistente com C, no qual a função `main` retorna um inteiro para informar ao sistema operacional se o programa foi executado de forma correta. Em C, o valor padrão 0 (ausência de um comando de retorno) significa que ele executou corretamente, enquanto um valor diferente de zero normalmente é um código de erro definido pelo programador. Para simplificar, a função `main` em Clite não terá um comando `return`, embora seu tipo de retorno seja diferente de `void`.

Já que uma função chamada pode ou não ser do tipo `void`, é importante assegurar que um comando `return` apareça em cada função não-`void`. Esse requisito é imposto pelo sistema de tipos de Clite, que é apresentado posteriormente.

10.1.2 Sintaxe Abstrata

O próximo passo neste estudo é examinar como a sintaxe abstrata de Clite é estendida para abordar essas novas idéias. A visão adicional de sintaxe abstrata fornece uma base para estender a idéia de validação de tipos a programas com globais, declarações e chamadas de funções. Também fornece a base para a definição de um interpretador que transforma a pilha de tempo de execução quando cada chamada de função é iniciada ou terminada.

Refinamentos na sintaxe abstrata de Clite apresentados no Capítulo 2, que suportarão globais, declarações e chamadas de funções, estão resumidos na Figura 10.3. As partes sublinhadas indicam adições à sintaxe abstrata original. Essas adições espelham aquelas que ocorrem dentro da sintaxe concreta apresentada anteriormente.

Observe as duas formas diferentes nas quais uma *Chamada* aparece em um programa – como um comando separado ou dentro de uma expressão. No primeiro caso, a *Chamada* é usada para chamar uma função cujo tipo `t` é `void`. No segundo caso, a função chamada deve retornar um resultado. Assim, um comando *Return* deve aparecer no corpo de uma função que não seja void, mas não no corpo de uma função void. Essa distinção importante será imposta nas regras de verificação de tipos e usada na definição do interpretador para Clite.

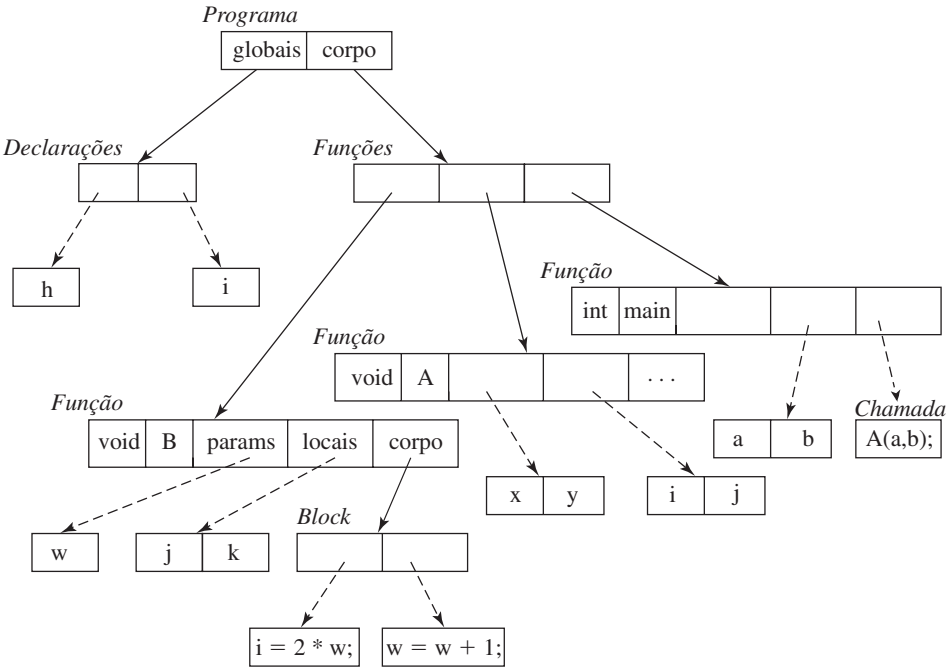
A Figura 10.4 fornece um esboço da sintaxe abstrata do programa mostrado na Figura 10.1. Aqui, as setas pontilhadas indicam abreviações de partes da árvore de sintaxe abstrata. Apenas os elementos-chave de funções, locais, parâmetros e chamadas são mostrados em detalhes aqui.

Para trabalhar com essas definições, implementamos um analisador descendente recursivo para essa extensão de Clite. Esse analisador pode ser baixado do site do livro e executado com o programa da Figura 10.1 como entrada.

A inclusão de chamadas e retornos de funções em uma linguagem de programação carrega um conjunto distinto de requisitos de verificação de tipo e semântica. Nas seções a seguir, desenvolvemos regras de verificação de tipos e semântica para programas em Clite que tenham escopo estático.

Programa = *Declarações* globais; *Funções* funções
Funções = *Função**
Função = *Tipo* t; *String* id; *Declarações* params, locais; *Bloco* corpo
Tipo = int | boleano | float | char | void
Comando = *Salto* | *Bloco* | *Atribuição* | *Condição* | *Laço* | *Chamada* | *Retorno*
Chamada = *String* nome; *Expressões* args
Expressões = *Expressão**
Retorno = *Variável* alvo; *Expressão* resultado
Expressão = *Variável* | *Valor* | *Binário* | *Unário* | *Chamada*

| Figura 10.3 Sintaxe Abstrata de Funções, Globais e Chamadas em Clite



| Figura 10.4 Esboço de Sintaxe Abstrata para um Programa em Clite

Esta discussão deixa claro como a pilha de tempo de execução é distribuída para suportar escopo, funções, chamadas e ligação de parâmetros-argumentos.

10.2 COMPLETANDO O SISTEMA DE TIPOS DE CLITE

O sistema de tipos de Clite possui regras que evitam que erros de tipo ocorram durante uma chamada a uma função. Essas regras complementam as regras originais de verificação de tipos de Clite (veja o Capítulo 6) para assegurar que qualquer programa Clite sintaticamente correto seja seguro quanto a tipos com essa nova consideração.

> **Regra de Tipo 10.1** *Cada função e* `id` *global devem ser únicos.*

Analise o programa da Figura 10.1. Essa regra evita que um programa tenha duas funções diferentes chamadas `A`, por exemplo. A sintaxe concreta de Clite não pode por si só assegurar essa unicidade mais do que pode assegurar que todas as variáveis em uma declaração tenham nomes únicos.

> **Regra de Tipo 10.2** *Os* `params` *e* `locais` *de cada função devem ter* `id`*s únicos entre si.*

Por exemplo, essa regra assegura que `id`s da lista de parâmetros e variáveis locais na função `A` — `x`, `y`, `i` e `j` — não estejam duplicados.

> **Regra de Tipo 10.3** *Cada Comando no* `corpo` *de cada função deve ser válido quanto às variáveis locais, aos parâmetros e às globais visíveis da função.*

```
int fibonacci (int n) {
     int fib0, fib1, temp, k;
     fib0 = 0; fib1 = 1; k = n;
     while (k > 0) {
       temp = fib0;
       fib0 = fib1;
       fib1 = fib0 + temp;
       k = k - 1;
     }
     return fib0;
}
int main () {
    int answer;
    answer = fibonacci(8);
}
```

| Figura 10.5 Uma Função Clite para Calcular um Número de Fibonacci

A lista de locais, parâmetros e globais que estão dentro do escopo do corpo de A contém `x`, `y`, `i`, `j` e `h`, mas não a global `i`. Cada comando no corpo de `A` pode referenciar apenas esses nomes e nenhum outro. Por exemplo, uma referência cruzada de um comando em `A` para a variável local `k` em `B` seria um erro, por essa regra.

> **Regra de Tipo 10.4** *Um comando de Retorno deve aparecer no corpo de cada função que não seja void, exceto* `main`*, e sua Expressão deve ter o mesmo Tipo dessa função.*

Por exemplo, analise o programa em Clite da Figura 10.5. Esse programa calcula o oitavo número de Fibonacci chamando a função `fibonacci`. Aqui, a função `fibonacci` é chamada pelo programa principal com o argumento 8. Já que essa função contém um comando de retorno e o tipo dessa expressão (`int`) é o mesmo do tipo da função, essa regra é satisfeita.

> **Regra de Tipo 10.5** *Nenhum comando Return pode aparecer em uma função* `void`.

Por outro lado, o `corpo` de cada função `void` `A` e `B` na Figura 10.1 não contém, apropriadamente, nenhum comando de *Retorno*.

> **Regra de Tipo 10.6** *Todo Comando de Chamada deve identificar uma Função void, e cada Expressão Chamada deve identificar uma Função não-void.*[2]

Os dois comandos de *Chamada* no programa da Figura 10.1 identificam a função void `A` e a função void `B`. A *Chamada* no programa da Figura 10.5 é uma expressão que identifica a função `fibonacci` não-void. Embora a sintaxe abstrata desse programa não seja mostrada aqui, os leitores podem desejar executar o interpretador Clite para ver que a expressão `fibonacci(8)` é analisada para uma *Chamada* a uma função com tipo de retorno `int`.

> **Regra de Tipo 10.7** *Toda Chamada deve ter o mesmo número de* `args` *e* `params` *na Função identificada, e cada um desses* `args` *deve ter o mesmo tipo do seu* `param` *correspondente, lendo-se da esquerda para a direita.*

2. Essa regra é um afastamento de C/C++.

Examinaremos a *Chamada* `A(a,b)` na Figura 10.1. Os dois argumentos `a` e `b` são do tipo `int`, o que concorda com os tipos dos dois parâmetros `x` e `y`, respectivamente. Se essa *Chamada* tivesse um número diferente de argumentos, ou se um dos argumentos fosse de um tipo `float` (suponhamos), essa regra teria pego um erro.

> **Regra de Tipo 10.8** *O tipo de uma Chamada é o tipo da função identificada, e a expressão na qual ela aparece deve ser válida de acordo com as Regras de Tipo 6.5 e 6.6.*

Por exemplo, a expressão `fibonacci(8)` que aparece na Figura 10.5 possui o tipo `int`, que é válido no contexto da *Atribuição* na qual ele aparece.

No conjunto, essas regras de tipo se combinam com as regras iniciais do Capítulo 6 para assegurar que Clite seja seguro quanto a tipos. Se essas regras forem impostas por um verificador de tipos, nenhum programa deve falhar por causa de um erro de tipo. É claro que, para ser completamente rigoroso, qualquer alegação de segurança de tipos para um projeto de linguagem deve ser provada formalmente.

10.3 SEMÂNTICA DE CHAMADA E RETORNO DE FUNÇÕES

A semântica de Clite pode ser estendida para definir uma coleção de regras significativas que estendam aquelas do Capítulo 7. Os programas das Figuras 10.1 e 10.5 são usados para ilustrar as características-chave dessas extensões.

Sempre que uma *Chamada* for executada, um novo registro de ativação será colocado na pilha e preenchido com os parâmetros e locais da função chamada e uma variável para armazenar o seu resultado de retorno. Cada parâmetro é atribuído a uma cópia do valor do respectivo argumento, supondo que argumentos sejam passados por valor. A regra a seguir define o significado de uma *Chamada*:

> **Regra de Significado 10.1** *O significado de uma Chamada c para uma Função f tem as seguintes etapas:*
>
> **1** *Criar um novo registro de ativação, adicionar os* `parâmetros` *e* `locais` *de f a ele.*
>
> **2** *Avaliar cada um dos* `args` *de c e atribuir seu Valor aos* `params` *correspondentes de f no registro de ativação.*
>
> **3** *Se a função for não-void, adicionar ao registro de ativação uma variável de resultado idêntica ao nome e ao tipo da função.*
>
> **4** *Empilhar esse registro de ativação na pilha de tempo de execução.*
>
> **5** *Interpretar os comandos no* `corpo` *de f.*
>
> **6** *Desempilhar o registro de ativação da pilha de tempo de execução.*
>
> **7** *Se a função for não-void, retornar o valor da variável de resultado para a chamada.*

Para ilustrar essa regra em ação, analisaremos novamente o programa em Clite da Figura 10.1. Quando o programa começa a ser executado, a pilha possui apenas as variáveis globais `h` e `i`. A primeira chamada vem do sistema para a função `main`, em cujo caso a Regra de Significado 10.1 é chamada. Os passos 1 e 2 se combinam para empilhar a primeira ativação, conforme mostrado à esquerda da Figura 9.8, e então o corpo de `main` é interpretado.

As variáveis que forem visíveis no tempo de execução dessa chamada, as quais chamamos de *estado visível*, são mostradas na primeira linha da Figura 10.6. Estas incluem os parâmetros e locais, junto a quaisquer variáveis globais cujos nomes sejam distintos daqueles dos parâmetros e locais.

As primeiras três *Atribuições* em `main` são interpretadas como o eram em Clite, usando as regras de semântica do Capítulo 7. Esses comandos alteram o valor da variável global `h` assim como o das variáveis locais `a` e `b`.

Chamando	Retornando	Estado Visível
main		⟨*h, indefinido*⟩, ⟨*i, indefinido*⟩, ⟨*a, indefinido*⟩, ⟨*b, indefinido*⟩
A		⟨*h,* 5⟩, ⟨*x,* 3⟩, ⟨ *y,* 2⟩, ⟨*i, indefinido*⟩, ⟨ *j, indefinido*⟩
B		⟨*h,* 5⟩, ⟨*i, indefinido*⟩, ⟨*w,* 5⟩, ⟨ *j, indefinido*⟩, ⟨*k*, indefinido⟩
	B	⟨*h,* 5⟩, ⟨*i,* 10⟩, ⟨w, 6⟩, ⟨ *j*, indefinido⟩, ⟨*k, indefinido*⟩
	A	⟨*h,* 5⟩, ⟨*x,* 3⟩, ⟨ *y,* 2⟩, ⟨*i, indefinido*⟩, ⟨ *j, indefinido*⟩
	main	⟨*h,* 5⟩, ⟨*i,* 10⟩, ⟨*a,* 3⟩, ⟨*b,* 2⟩

| Figura 10.6 Rastreamento de um Programa com Chamadas de Função

O quarto comando em `main` é uma *Chamada*, e assim a Regra de Significado 10.1 é chamada novamente. Isso cria um segundo registro de ativação, conforme mostrado no meio da Figura 9.8, e então o corpo de `A` começa a ser executado.

Nesse ponto, a interpretação do corpo de `main` é suspensa até que todos os comandos no corpo de `A` sejam completados. O estado visível para essa chamada é mostrado na segunda linha da Figura 10.6; observe que a variável global *i* se torna escondida, já que outra variável com esse nome é redeclarada localmente dentro de `A`.

Já que o corpo de `A` possui uma chamada para `B`, a Regra de Significado 10.1 se aplica novamente, adicionando um terceiro registro de ativação à pilha, conforme mostrado na Figura 9.8. O estado visível para essa chamada é mostrado na terceira linha da Figura 10.6; observe que a variável global *i* agora está visível dentro de `B`.

Quando o `corpo` de `B` é totalmente interpretado, seu registro de ativação é o primeiro a ser desempilhado e a chamada de `A` para `B` é dessa forma completada. De forma semelhante, quando o `corpo` de `A` é interpretado integralmente, uma Chamada de `main` para `A` é completada e esse registro de ativação é desempilhado. Finalmente, o registro de ativação de main é desempilhado e o controle retorna para o sistema. O estado visível antes que cada uma dessas chamadas seja completada é mostrado nas últimas três linhas da Figura 10.6.

10.3.1 Funções não-void

O significado de uma *Chamada* para uma *Função* não-void é semelhante, exceto o fato de o *valor* retornado ser obtido por meio da execução de um comando *Return*. Esse resultado é definido nos passos 3 e 7 da Regra de Significado 10.1. O sistema de tipos garante que uma função que não seja void contenha um comando *Return*, o qual, quando encontrado, atue como uma atribuição. O alvo dessa atribuição é a variável de resultado identificada pelo nome da função no registro de ativação.

> **Regra de Significado 10.2** *O significado de um Return é calculado pela substituição do* `valor` *da Variável resultado* (*o nome da função chamada*) *no registro de ativação pelo* `valor` *da Expressão resultado.*

Já que o comando *Return* pode ocorrer em qualquer local dentro do corpo da função, a Regra de Significado 8.6 para um *Bloco* também deve ser modificada, de modo que termine assim que o comando *Return* for encontrado.

> **Regra de Significado 10.3** *O significado de um Bloco é o significado agregado desses comandos quando aplicados ao estado corrente, até e incluindo o ponto onde o primeiro Return for encontrado. Se houver Blocos aninhados, o primeiro Return encontrado sinaliza o término do Bloco mais externo, ou seja, o primeiro Return encontrado completa o corpo da função.*

Por exemplo, suponha que chamemos a função `fibonacci` que retorna um valor `int` conforme mostrado na Figura 10.5. Essa função é chamada a partir de `main` como a *Expressão* fonte dentro da seguinte atribuição:

```
answer = fibonacci(8);
```

Assim, o resultado da *Chamada* a `fibonacci(8)` é retornado como um *Valor* que se tornará parte da *Variável* `answer` quando a *Atribuição* for interpretada.

O registro de ativação para essa *Chamada* contém seis entradas: quatro para as variáveis locais da função, uma para o parâmetro `n` e uma para a variável de resultado `fibonacci`. No passo 2 da Regra de Significado 10.1., `n` recebe o valor 8. Já que `fib0` é a *Expressão* no comando *Return*, o significado dessa *Chamada* é o valor de `fib0` quando esse comando *Return* é interpretado.

O corpo da função é um bloco com cinco declarações: três *Atribuições*, um *While* e um *Return*. Assim, os quatro primeiros comandos no *Bloco* são interpretados como seriam em Clite. Quando *Return* é interpretado, o valor de `fib0` é retornado e o registro de ativação da *Chamada* é desempilhado.

Para exercitar esse programa, podemos executar o interpretador Clite novamente e obter resultados como aqueles mostrados na Figura 10.7. Observe que a variável de resultado `fibonacci` aparece no registro de ativação para armazenar o resultado retornado na *Chamada*.

Finalmente, observamos que a passagem de argumentos por referência poderia ser adicionada à definição da Regra de Significado 10.1, modificando-se o passo 2 para solicitar que cada `arg` passado por referência tenha seu endereço atribuído ao parâmetro correspondente. Também teríamos de definir a desreferenciação de um parâmetro por referência em uma *Atribuição* e uma *Expressão*. Finalmente, precisaríamos modificar a sintaxe concreta e abstrata de Clite para incorporar esse recurso adicional de forma apropriada.

10.3.2 Efeitos Colaterais Revisitados

Os leitores podem ter percebido que as definições anteriores não colocam restrições sobre como e quando as funções com efeitos colaterais podem ocorrer em Clite. Em particular, qualquer função que altere o valor de uma variável global durante a vida da sua chamada possui um efeito colateral. Esse é o caso, por exemplo, da função `B` do programa da Figura 10.1, que altera o valor da variável global `i` quando é chamada.

Além disso, se `B` tivesse sido declarada como uma função não-void, esse efeito colateral teria impacto sutil no resultado retornado por uma expressão como `B(h) + i`, na qual `i` denota a mesma variável global mencionada anteriormente.

Chamando	Retornando	Estado Visível
main		⟨*answer, indefinido*⟩
fibonacci		⟨*n,* 8⟩, ⟨*fib*0*, indefinido*⟩, ⟨*fib*1*, indefinido*⟩, ⟨*temp, indefinido*⟩, ⟨*k, indefinido*⟩, ⟨*fibonacci, indefinido*⟩
	fibonacci	⟨*n,* 8⟩, ⟨*fib*0*,* 21⟩, ⟨*fib*1*,* 34⟩, ⟨*temp,* 13⟩, ⟨*k,* 0⟩, ⟨*fibonacci,* 21⟩
	main	⟨*answer,* 21⟩

| Figura 10.7 Rastreamento do Programa para uma Chamada de Função Não-void

Lembre-se de que, no Capítulo 8, fomos cuidadosos em definir a ordem de avaliação de operandos em uma operação `Binária` (como esta) como da esquerda para a direita, de modo que o significado de uma expressão que contenha uma chamada com um efeito colateral seja bem definido. Essa definição também garante que o resultado retornado por `B(h) + i` será diferente daquele retornado por `i + B(h)`.

Um problema importante com efeitos colaterais é que eles criam mais distância entre a semântica de expressões matemáticas quando aparecem em um programa do que quando aparecem em matemática. Por exemplo, em Clite, a adição é claramente não comutativa, enquanto em matemática normalmente é.

Observamos finalmente que linguagens de programação sem efeitos colaterais existem, especialmente no paradigma de programação funcional. Por exemplo, nosso tratamento da linguagem Haskell no Capítulo 14 mostra que princípios matemáticos e de programação não precisam ser incompatíveis.

10.4 π TRATAMENTO FORMAL DE TIPOS E SEMÂNTICAS

Esta seção repete as duas seções anteriores usando um estilo matemático formal. Seu propósito é ilustrar o poder da semântica denotacional para permitir especificações completas do sistema de tipos e semântica de uma linguagem de programação. Já que é independente de máquina, essa notação pode ser usada por projetistas de compiladores e linguagem quando definem e implementam novas linguagens de programação.

10.4.1 Mapas de Tipos para Clite

A noção de "mapa de tipos" para um programa em Clite com funções é mais complexa do que foi representado no Capítulo 6. Por exemplo, o escopo estático permite que ocorram múltiplas definições do mesmo *Identificador*, mas que apenas um esteja acessível por algum determinado comando no programa.

Assim, embora no Capítulo 6 tenhamos discutido um único mapa de tipos, um tratamento apropriado de escopo e visibilidade requer que um mapa de tipos diferente seja definido para cada função diferente no programa.

> **Definição:** O *mapa de tipos* para uma função *f* é definido como um conjunto de pares e trios, cada um representando uma variável global, uma função global, um parâmetro ou uma variável local.

Suponha que tm_G, tm_F, *f.locais* e *f.params* denotem o mapa de tipos das variáveis globais, os protótipos de todas as funções, os mapas de tipos de variáveis locais e os parâmetros, respectivamente, para alguma função *f* específica. Então podemos definir o mapa de tipos tm_f para *f* como a união prevalecente $tm_G \cup tm_F$ e o conjunto de todos os pares das variáveis locais e dos parâmetros de *f* com seus respectivos tipos, ou seja:

$$\begin{aligned} tm_f &= (tm_G \cup tm_F) \,\overline{\cup}\, (f.locais \cup f.params) \\ &= \{\langle v.id_1, v.t_1\rangle, \langle v.id_2, v.t_2\rangle, \ldots, \langle v.id_n, v.t_n\rangle, \\ &\quad \langle f.id_1, f.t_1, \{\langle p_{11}, t_{11}\rangle, \ldots, \langle p_{1n_1}, t_{1n_1}\rangle\}\rangle, \end{aligned}$$

$$\langle f.id_2, f.t_2, \{\langle p_{21}, t_{21}\rangle, \ldots, \langle p_{2n_2}, t_{2n_2}\rangle\}\rangle,$$
$$\ldots$$
$$\langle f.id_m, f.t_m, \{\langle p_{m1}, t_{m1}\rangle, \ldots, \langle p_{mn_m}, t_{mn_m}\rangle\}\rangle\} \overline{\cup} (f.locais \cup f.params)$$

Cada par é um identificador de variável *v.id* e tipo *v.t* e cada trio é um identificador de função *f.id*, tipo *f.t* e uma seqüência de parâmetros com seus tipos.

Analise o programa exemplo da Figura 10.1. Lá, tm_G e tm_F teriam os seguintes valores:

$$tm_G = \{\langle h, int\rangle, \langle i, int\rangle\}$$
$$tm_F = \{\langle B, void, \{\langle w, int\rangle\}\rangle, \langle A, void, \{\langle x, int\rangle, \langle y, int\rangle\}\rangle, \langle main, int, \{\}\rangle\}$$

Essa definição assegura que a visibilidade de variáveis e funções definidas globalmente sejam limitadas apropriadamente em casos em que uma nova variável local ou um parâmetro com o mesmo nome seja definido dentro de uma função.

Por exemplo, cada função no nosso programa exemplo possui um mapa de tipos distinto, conforme definido a seguir:

$$\begin{aligned}
tm_B &= (tm_G \cup tm_F) \overline{\cup} \{\langle w, int\rangle, \langle j, int\rangle, \langle k, int\rangle\} \\
&= \{\langle h, int\rangle, \langle i, int\rangle, \langle w, int\rangle, \langle j, int\rangle, \langle k, int\rangle, \\
&\quad \langle B, void, \{\langle w, int\rangle\}\rangle, \langle A, void, \{\langle x, int\rangle, \langle y, int\rangle\}\rangle, \langle main, int, \{\}\rangle\} \\
tm_A &= (tm_G \cup tm_F) \overline{\cup} \{\langle x, int\rangle, \langle y, int\rangle, \langle i, boolean\rangle, \langle j, boolean\rangle\} \\
&= \{\langle h, int\rangle, \langle x, int\rangle, \langle y, int\rangle, \langle i, boolean\rangle, \langle j, boolean\rangle, \\
&\quad \langle B, void, \{\langle w, int\rangle\}\rangle, \langle A, void, \{\langle x, int\rangle, \langle y, int\rangle\}\rangle, \langle main, int, \{\}\rangle\} \\
tm_{main} &= (tm_G \cup tm_F) \overline{\cup} \{\langle a, int\rangle, \langle b, int\rangle\} \\
&= \{\langle h, int\rangle, \langle i, int\rangle, \langle a, int\rangle, \langle b, int\rangle\} \\
&= \{\langle h, int\rangle, \langle i, int\rangle, \langle a, int\rangle, \langle b, int\rangle, \\
&\quad \langle B, void, \{\langle w, int\rangle\}\rangle, \langle A, void, \{\langle x, int\rangle, \langle y, int\rangle\}\rangle, \langle main, int, \{\}\rangle\}
\end{aligned}$$

Com essa base, podemos estender a função *tipagem* dada no Capítulo 8 de modo que gere mapas de tipos como os mostrados anteriormente para qualquer programa em Clite com globais *G*, funções *F* e função individual *f*:

$$tipagem : Declarações \times Funções \times Função \rightarrow MapaDeTipo$$
$$\begin{aligned}
tipagem(G, F, f) &= (tm_G \cup tm_F) \overline{\cup} tm_F, \text{ onde} \\
tm_G &= \cup_{i \in 1,\ldots,n} \langle G_i.id, G_i.t\rangle \\
tm_F &= \cup_{i \in 1,\ldots,m} \langle F_i.id, F_i.t, F_i.params\rangle \\
tm_f &= f.params \cup f.locais \\
f.params &= \cup_{i \in 1,\ldots,p} \langle param_i.id, param_i.t\rangle \\
f.locais &= \cup_{i \in 1,\ldots,l} \langle local_i.id, local_i.t\rangle
\end{aligned}$$

ou seja, tm_G é a união de todos os pares de variáveis globais e seus tipos, tm_F é a união de todas as declarações de funções e seus tipos, e tm_f contém os próprios parâmetros da função e das variáveis locais e seus tipos.

A união de substituição $\overline{\cup}$ que aparece na expressão $(tm_G \cup tm_F) \overline{\cup} tm_F$ realiza a idéia de que os parâmetros e os locais de uma função podem limitar o escopo de quaisquer globais e outras funções de mesmo nome pela duração da chamada. Por exemplo, perceba que,

em tm_A, o escopo da variável global *i* exclui o corpo da função `A` porque `A` possui uma variável local declarada com o mesmo nome.

10.4.2 Formalizando as Regras de Tipo de Clite

Com uma definição de mapas de tipos à mão, podemos agora formalizar a idéia de validação de tipos para qualquer programa em Clite. Muitas das funções definidas a seguir espelham idéias expressas nas regras de tipos apresentadas na Seção 10.2.

Trabalhando de cima para baixo, aqui está uma definição formal de validação de tipos para um programa em Clite:

$$V : Programa \rightarrow \mathbf{B}$$
$$V(p) = V(tm_{p.globais} \cup tm_{p.corpo}) \wedge \forall f \in p.corpo : V(f, tipagem(G, F, f))$$

Ou seja, um programa é *válido* se sua variável global e declarações de funções forem válidas e cada uma das suas funções (incluindo `main`) for válida com respeito a seu mapa de tipos.

Isso significa que o programa exemplo da Figura 10.1 é válido se as seguintes condições forem satisfeitas simultaneamente:

$V(tm_G \cup tm_F)$
$V(A, tipagem(G, F, A))$
$V(B, tipagem(G, F, B))$
$V(\text{main}, tipagem(G, F, main))$, em que
$tm_G = \{\langle h, int\rangle, \langle i, int\rangle\}$
$tm_F = \{\langle B, void, \{\langle w, int\rangle,\}\rangle, \langle A, void, \langle x, int\rangle, \langle y, int\rangle\}\rangle, \langle main, int, \{\}\rangle\}$

A validade de um mapa de tipos requer que todos os identificadores de funções e as variáveis globais sejam únicos entre si e que nenhuma variável tenha tipo `void`. Finalmente, deve haver exatamente uma função com o nome `main`. As seguintes definições ilustram esses conceitos.

$$V : Declarações \rightarrow Boleano$$
$$V(d) = \forall i, j \in \{1, \ldots, n_d\} : i \neq j \Rightarrow (d_i.id \neq d_j.id \wedge d_i.t \neq void)$$

$$V : Funções \rightarrow Boleano$$
$$V(f) = \forall i, j \in \{1, \ldots, n_f\} : i \neq j \Rightarrow f_i.id \neq f_j.id \wedge$$
$$\exists i \in \{1, \ldots, n_f\} : f_i.id = \text{main}$$

Validade de uma Função A validade de uma função individual em um programa carrega diversos requisitos.

1. Os parâmetros e locais de uma função devem ter nomes únicos entre si.
2. As declarações no corpo da função devem ser válidas com respeito ao seu mapa de tipos.
3. Uma função não-void (exceto `main`) deve conter pelo menos um comando Return, e uma função void (ou `main`) não.

Aqui está uma formalização desses requisitos, em que G e F denotam as variáveis globais e funções, respectivamente:

$$V: \textit{Função} \times \textit{MapaDeTipos} \rightarrow \mathbf{B}$$
$$V(f, tm) = V(f.params \cup f.locais) \wedge V(f.corpo, tm) \wedge$$
$$(f.t \neq void \wedge f.id \neq main \wedge \exists s \in f.corpo : s \text{ é um } Return$$
$$\wedge tipoDe(s.result) = f.t) \vee$$
$$((f.t = void \vee f.id = main) \wedge \nexists s \in m.corpo : s \text{ é um } Return), \text{ em que}$$
$$tm = tipagem(G, F, f)$$

Por exemplo, no programa anterior, necessitamos que $V(A, tm_A)$, $V(B, tm_B)$ e $V(main, tm_{main})$.

Validade de Chamada e Retorno Para completar os requisitos de verificação de tipos para Clite precisamos especificar a validade de tipos para os novos comandos *Chamada* e *Retorno*.

Cada *Chamada* deve ter nome idêntico ao id de alguma função void no mapa de tipos *tm* da função *f* na qual a *Chamada* aparece. Além disso, essa função, digamos *g*, deve ter o número de parâmetros (*np*) igual ao de argumentos da *Chamada* (*na*). Finalmente, para cada par parâmetro-argumento entre essa função e a *Chamada*, ambos os membros do par devem ser do mesmo tipo.

$$V: Chamada \times MapadeTipos \rightarrow \mathbf{B}$$
$$V(c, tm) = \exists g \in tm : (c.nome = g.id \wedge np = na \wedge$$
$$\forall i \in \{1,..., np\} : tipoDe(c.args_i) = tipoDe(g.params_i)), \text{ em que}$$
$$tm = tipagem(G, F, f)$$

Para um comando *Return*, o tipo da expressão resultante deve corresponder ao tipo da função no qual o comando aparece.

$$V: Return \times MapadeTipos \rightarrow \mathbf{B}$$
$$V(r, tm) = tipode(r.resultado) = f.t, \text{ em que } tm = tipagem(G, F, f)$$

As funções de verificação de tipos V restantes para Clite são aquelas definidas no Capítulo 6 para as várias outras declarações e expressões consideradas no subconjunto Clite original. Uma integração completa dessas especificações de tipos com as funções V originais é deixada como exercício.

10.4.3 Formalizando a Semântica de Clite

Para formalizar a semântica integral de Clite, as noções de memória estática, registro de ativação e pilha de tempo de execução precisam ser definidas formalmente para caracterizar apropriadamente o comportamento dinâmico da pilha quando as chamadas de funções forem iniciadas e terminadas.

Definição: O *ambiente* γ_f de uma função ativa *f* é um conjunto de pares que une variáveis cujo escopo inclua *f* com *endereços de memória* específicos.

Definição: A *memória* μ é um conjunto de pares que relacionam endereços a valores.

Definição: *Endereços de memória* são geralmente um subconjunto contíguo $\{0, \ldots, n\}$ dos inteiros e, muitas vezes, são denominados *espaço de endereço* do programa.

Por convenção, cada endereço de memória para o qual nenhuma variável tiver sido alocada ainda é marcado como *não usado* e cada endereço para o qual uma variável tiver sido alocada, mas nenhum valor tiver sido atribuído, é marcado como *indefinido*.

Por exemplo, suponha que temos um programa cujas variáveis *i* e *j* têm valores 13 e −1 em algum momento durante a execução da função *f*, que *k* ainda não tem um valor atribuído e que essas variáveis recebem locais de memória 154, 155 e 156. O estado de *f* representado por essa configuração pode ser expresso da seguinte forma:

$$\gamma_f = \{\langle i, 154\rangle, \langle j, 155\rangle, \langle k, 156\rangle\}$$
$$\mu = \{\langle 0, \textit{indefinido}\rangle \ldots \langle 154, 13\rangle, \langle 155, -1\rangle, \langle 156, \textit{indefinido}\rangle \ldots \langle a-1, \textit{indefinido}\rangle, \langle a, \textit{não usado}\rangle \ldots \langle n, \textit{não usado}\rangle\}$$

Aqui, $a - 1$ é o endereço do topo da pilha de tempo de execução, que contém o registro de ativação de *f*, entre outros. As variáveis *i, j* e *k* são variáveis locais ou parâmetros de *f* (se os endereços 154–156 estiverem dentro do registro de ativação de *f*) ou variáveis globais (se não estiverem).

O *endereço* de uma variável *v* no ambiente de *f* é dado pela função $\gamma_f(v) = \max x : \langle v.\text{id}, x\rangle \in \gamma_f$.[3] O *valor* dessa variável é dado por $\mu(\gamma_f(v))$, que é denotado de forma equivalente por $\sigma_f(v)$. Para este exemplo, temos

$$\gamma_f(\text{i}) = 154 \quad \text{e}$$
$$\sigma_f(i) = \mu(\gamma_f(i)) = 13$$

Agora a *atribuição* de um valor a uma variável pode ser definida se usarmos o operador de união de substituição introduzido no Capítulo 7. Por exemplo, suponha que queiramos atribuir o valor 20 para a variável *i* no exemplo anterior. A mudança na memória resultante dessa atribuição pode ser descrita por:

$$\mu' = \mu \overline{\cup} \{\langle \gamma_f(i), 20\rangle\}$$
$$= \mu \overline{\cup} \langle 154, 20\rangle$$
$$= \{\langle 0, \textit{indefinido}\rangle \ldots \langle 154, 20\rangle, \langle 155, -1\rangle, \langle 156, \textit{indefinido}\rangle \ldots \langle a-1, \textit{indefinido}\rangle, \langle a, \textit{não usado}\rangle \ldots \langle n, \textit{não usado}\rangle\}$$

O *estado* σ_f de uma função ativa *f* em um programa pode ser visto como o produto do ambiente γ_f de *f*, a memória μ e um inteiro *a* que marca o topo da pilha de tempo de execução, ou seja, $\sigma_f = \gamma_f \times \mu \times a$.

A alocação de variáveis para endereços na área estática e na pilha de tempo de execução é serial, começando no endereço 0. A qualquer momento durante a execução de um programa, a faixa de endereços ativos nessas duas áreas é $\{0, \ldots, a-1\}$, na qual *a* é o número de variáveis ativas. Neste modelo, supomos que o valor de cada variável recebe exatamente uma unidade de memória endereçável.[4]

3. O motivo para o uso de "max" para encontrar o valor da variável *v* é que pode haver mais de uma variável com o mesmo *id* alocado ao mesmo tempo. A natureza do nosso esquema de alocação de pilha nos garante que a função "max" recupere essa variável, cujo escopo inclui as declarações de *f* e não alguma outra variável na pilha com o mesmo *id*.

4. No Capítulo 5, vimos que esse não é o caso em sistemas reais; um valor char, por exemplo, requer apenas um byte, enquanto um ponto flutuante requer quatro. Fazemos essa suposição aqui para simplificar a discussão.

A função *alocar* define a mudança no estado quando um grupo de novas *Declarações* de variáveis $d_1, d_2, \ldots, d_k$ é adicionado à área estática ou à pilha de tempo de execução:

$$\begin{aligned} alocar(d_1, d_2, \ldots, d_k, \sigma) &= \gamma' \times \mu' \times a' \\ \text{em que } \gamma' &= \gamma \cup \{\{\langle d_1.\mathbf{v}.id, a\rangle, \langle d_2.\mathbf{v}.id, a+1\rangle, \ldots, \\ &\qquad \langle d_k.\mathbf{v}.id, a+k-1\rangle\} \\ \mu' &= \mu \,\overline{\cup}\, \{\langle a, \textit{indefinido}\rangle, \langle a+1, \textit{indefinido}\rangle, \ldots, \\ &\qquad \langle a+k-1, \textit{indefinido}\rangle\} \\ a' &= a + k \end{aligned}$$

apresentando o novo estado $\sigma' = \gamma' \times \mu' \times a'$. Observe que a criação de γ' é uma união comum de conjuntos e não uma de sobreposição, já que os *ids* das variáveis não são necessariamente únicos na pilha.

A remoção de um grupo de variáveis *k* do topo da pilha de tempo de execução reverte o processo e é definida pela função *desalocar:*

$$\begin{aligned} desalocar(d_1, d_2, \ldots, d_k, \sigma) &= \gamma' \times \mu' \times a' \\ \text{em que } \gamma' &= \gamma - \{\langle d_k.\mathbf{v}.id, a\rangle, \langle d_{k-1}.\mathbf{v}.id, a-1\rangle, \ldots, \\ &\qquad \langle d_1.\mathbf{v}.id, a-k+1\rangle\} \\ \gamma' &= \mu \,\overline{\cup}\, \{\langle a, \textit{não usado}\rangle, \langle a-1, \textit{não usado}\rangle, \ldots, \\ &\qquad \langle a-k+1, \textit{não usado}\rangle\} \\ a' &= a - k \end{aligned}$$

Assim, a criação de γ' é uma diferença comum de conjuntos, e a função *desalocar* está bem definida desde que $a - k \geq 0$, o que evita um *overflow* da pilha.

O estado resultante da execução de uma *Chamada c* para uma *Função f* em um programa Clite é definido de uma forma que espelhe a Regra de Significado 10.1.

$$\begin{aligned} &M : \textit{Chamada} \times \textit{Função} \times \textit{Estado} \rightarrow \textit{Estado} \\ &M(c, f, \sigma) = \textit{removerregistrodeativação}(\textit{ f.params, f.locais,} \\ &\qquad M(\textit{ f.corpo,} \\ &\qquad\ \textit{PorValor}(\textit{ f.params, c.args,} \\ &\qquad\ \textit{adicionarregistrodeativação}(\textit{ f.locais, f.params, } \sigma\ )))) \end{aligned}$$

As funções *removerregistrodeativação* e *adicionarregistrodeativação* efetivamente escondem as locais e os parâmetros do solicitante de acesso da função *chamada* durante a sua chamada. A função *PorValor* atribui o valor dos argumentos do solicitante aos parâmetros correspondentes antes de o corpo da função ser interpretado.

O estado inicial de σ_0 para qualquer programa Clite possui um ambiente γ_0 vazio, uma memória cujos locais estão todos marcados como *não usados*, e $a = 0$:

$$\sigma_0 = \gamma_0 \times \mu(0) \times 0 = \{\} \times \{\langle 0, \textit{não usado}\rangle, \langle 1, \textit{não usado}\rangle \ldots \langle n, \textit{não usado}\rangle\} \times 0$$

O significado de um programa completo é o seu estado global, $\sigma_G = alocar(p.globais, \sigma_0)$, que resulta quando a seguinte função é avaliada:

$$M : Programa \times Estado \rightarrow Estado$$
$$M(p, \sigma_0) = M(p.main(), alocar(p.globais, \sigma_0))$$

na qual *p.main()* denota uma chamada com 0 argumentos à função dentro de *p* cujo `id` é *main*. Isso é basicamente o que acontece quando qualquer programa C/C++ ou aplicação Java começa a ser executado.

Aqui está um resumo dos resultados da avaliação da função *M* para esse programa de exemplo que aparece na Figura 10.1. Primeiro, estabelecemos σ_G da seguinte maneira:

$$\begin{aligned}\sigma_G &= \gamma_G \times \mu \times 2 \\ &= \{\langle h, 0\rangle, \langle i, 1\rangle\} \times \{\langle 0, indefinido\rangle, \langle 1, indefinido\rangle, \langle 2, não\ usado\rangle \ldots \\ &\quad \langle n, não\ usado\rangle\} \times 2\end{aligned}$$

após a função *alocar* ser aplicada a *p.globais* e σ_0.

O estado alcançado quando a função `main` começa a ser executada é, dessa forma, definido pelo significado de uma chamada que aloca um novo registro de ativação com 0 parâmetros e duas variáveis locais *a* e *b*, ou seja:

$$\begin{aligned}\sigma_{main} &= \gamma_G \cup \{\langle a, 2\rangle, \langle b, 3\rangle\} \times \mu' \times 4 \\ &= \{\langle h, 0\rangle, \langle i, 1\rangle, \langle a, 2\rangle, \langle b, 3\rangle\}\ \mu' \times 4 \\ \text{em que } \mu' &= \{\langle 0, indefinido\rangle, \langle 1, indefinido\rangle, \langle 2, indefinido\rangle, \langle 3, indefinido\rangle, \langle 4, não\ usado\rangle \\ &\quad \ldots \langle n, não\ usado\rangle\}\end{aligned}$$

Exceto pelas ligações "slink" e "dlink" e os valores atribuídos para *a* e *b* pelas três primeiras declarações em *main*, essa estrutura corresponde ao estado inicial da pilha de tempo de execução mostrada à esquerda da Figura 9.8. De modo geral, a semântica formal desse programa espelha a imagem intuitiva de chamada e retorno de função mostrada naquela figura.

As três primeiras declarações em *main* deixam os valores de *h, a* e *b* em 5, 3 e 2, conforme mostrado no estado que ocorre imediatamente antes de *main* chamar *A*.

$$\begin{aligned}\sigma_{main} &= \gamma_G \cup \{\langle a, 2\rangle, \langle b, 3\rangle\} \times \mu \times 4 \\ &= \{\langle h, 0\rangle, \langle i, 1\rangle, \langle a, 2\rangle, \langle b, 3\rangle\} \times \mu_{main} \times 4 \\ \text{em que } \mu_{main} &= \{\langle 0, 5\rangle, \langle 1, indefinido\rangle, \langle 2, 3\rangle, \langle 3, 2\rangle, \langle 4, não\ usada\rangle \ldots \langle n, não\ usado\rangle\}\end{aligned}$$

Na entrada de `A` por intermédio da chamada `A(a, b)` a partir de *main*, ocorre a seguinte mudança:

$$\begin{aligned}\sigma_A &= \gamma_{main} - main.locais \cup \{\langle x, 4\rangle, \langle y, 5\rangle, \langle i, 6\rangle, \langle j, 7\rangle\} \times \mu_A \times 8 \\ &= \{\langle h, 0\rangle, \langle i, 1\rangle, \langle x, 4\rangle, \langle y, 5\rangle, \langle i, 6\rangle, \langle j, 7\rangle \times \mu_A \times 8 \\ \text{em que } \mu_A &= \{\langle 0, 5\rangle, \langle 1, indefinido\rangle, \langle 2, 3\rangle, \langle 3, 2\rangle, \langle 4, 3\rangle, \langle 5, 2\rangle, \langle 6, indefinido\rangle, \\ &\quad \langle 7, indefinido\rangle, \langle 8, não\ usado\rangle \ldots \langle n, não\ usado\rangle\}\end{aligned}$$

Essa configuração corresponde ao programa do meio do diagrama da pilha de tempo de execução mostrado na Figura 9.8. Observe aqui que a variável global `i` se tornou escondida do estado ativo para essa chamada, já que outra variável com o mesmo nome possui um endereço superior dentro de γ_A. As variáveis locais `a` e `b` dentro de *main* também são inacessíveis a partir do ambiente de `A` enquanto `A` está ativo.

Na entrada de `B`, outro registro de ativação é adicionado ao ambiente, deixando o seguinte estado:

$$\begin{aligned}\sigma_B &= \gamma_A - A.locais - A.params \cup \{\langle w, 8\rangle, \langle j, 9\rangle, \langle k, 10\rangle \times \mu_B \times 11\\ &= \{\langle h, 0\rangle, \langle i, 1\rangle, \langle w, 8\rangle. \langle j, 9\rangle, \langle k, 10\rangle\} \times \mu_B \times 11\end{aligned}$$

onde μ_B= {⟨0, 5⟩, ⟨1, *indefinido*⟩, ⟨2, 3⟩, ⟨3, 2⟩, ⟨4, 3⟩, ⟨5, 2⟩, ⟨6, *indefinido*⟩, ⟨7, *indefinido*⟩, ⟨8, *indefinido*⟩, ⟨9, *indefinido*⟩, ⟨10, *indefinido*⟩, ⟨11, *não usado*⟩, . . . ⟨*n*, *não usado*⟩}

Essa configuração corresponde ao diagrama do lado direito da pilha de tempo de execução mostrada na Figura 9.8.

A execução do corpo de `B` nesse estado altera o valor da variável global `i`, deixando o estado seguinte no retorno para `A`. Observe que o registro de ativação de `B` desapareceu e os estados de seus locais de memória são retornados para *não usados.*

$$\begin{aligned}\sigma_A &= \{\langle h, 0\rangle, \langle i, 1\rangle, \langle x, 4\rangle, \langle y, 5\rangle, \langle i, 6\rangle, \langle j, 7\rangle\} \times \mu_A \times 8\\ \text{onde } \mu_A &= \{\langle 0, 5\rangle, \langle 1, 10\rangle, \langle 2, 3\rangle \langle 3, 2\rangle, \langle 4, 3\rangle, \langle 5, 2\rangle, \langle 6, \textit{indefinido}\rangle, \langle 7, \textit{indefinido}\rangle,\\ &\quad \langle 8, \textit{não usado}\rangle \ \ldots \ \langle n, \textit{não usado}\rangle\}\end{aligned}$$

Finalmente, quando o controle retorna para `main`, o estado inclui o novo valor da variável global `i` e reflete o abandono do registro de ativação para `A`, que terminou, e a restauração das variáveis locais de `main`:

$$\begin{aligned}\sigma_{main} &= \{\langle h, 0\rangle, \langle i, 1\rangle, \langle a, 2\rangle, \langle b, 3\rangle\}\\ \text{e } \mu_{main} &= \{\langle 0, 5\rangle, \langle 1, 10\rangle, \langle 2, 3\rangle, \langle 3, 2\rangle, \langle 4, \textit{não usado}\rangle, ..., \langle n, \textit{não usado}\rangle\}\end{aligned}$$

O significado de uma *Chamada c* a uma *Função* não-void *f* é um *Valor*. Por exemplo, suponha que tenhamos definido a função `fibonacci` que retorna um valor `int`. Essa função é chamada no contexto de uma *Expressão* maior, como:

```
answer = fibonacci(8);
```

Nesse caso, o resultado `fibonacci(8)` é retornado como um novo *Valor* para o estado corrente, em vez de afetar uma transição de estado (como o faria um comando *Chamada* separado). Isso sugere a seguinte definição alternativa para uma chamada a uma função não-void:

$$\begin{aligned}&M : \textit{Chamada} \times \textit{Função} \times \textit{Estado} \rightarrow \textit{Valor}\\ &M(c, f, \sigma) = \sigma\,(f.id,\ \textit{removeregistrodeativação}(f.params, f.locais,\\ &\qquad M(f.corpo,\\ &\qquad\quad \textit{alocar}(f.id,\\ &\qquad\qquad \textit{PorValor}(f.params, c.args,\\ &\qquad\qquad\quad \textit{adicionarregistrodeativação}(f.locais, f.params, \sigma)))))\end{aligned}$$

Observe que introduzimos a alocação de uma variável local adicional na pilha, com *alocar*(*f.id,* σ), para acomodar o valor retornado pela chamada.

Isso permite que o significado de um comando *Return* seja definido como um tipo especial de atribuição–um, que atribui o valor do seu *resultado* à sua variável adicional *f.id*.

$$M : Return \times \Sigma \rightarrow \Sigma$$
$$M(r, \sigma) = \sigma \overline{\cup} \{\langle f.id, M(r, resultado, \sigma)\rangle\}$$

Finalmente, o significado do corpo da função, que é um *Bloco*, deve ser alterado da sua definição original no Capítulo 8 para acomodar a dinâmica de um comando *Return*. Aqui está uma redefinição do significado de um *Bloco* a partir daquele que leva *Return* em consideração:

$$\begin{aligned} M &: Bloco \times Estado \rightarrow Estado \\ M(b, \sigma) &= \sigma && \text{se } b = \emptyset \\ &= M(b_1, \sigma) && \text{se } b = b_1 b_2 \ldots b_n \wedge b_1 = Return \\ &= M((Bloco)_{b_{2 \ldots n}}, M(b_1, \sigma)) && \text{caso contrário} \end{aligned}$$

Ou seja, se o primeiro comando restante no *Bloco* de declarações for um *Return*, seu significado limita o significado do *Bloco* inteiro em que ele ocorre.

10.5 RESUMO

Este capítulo analisou a implementação de linguagens de programação com funções. Os aspectos dinâmicos da pilha de tempo de execução que suportam a chamada e o retorno de funções foram discutidos.

Um interpretador funcionando para a linguagem Clite inteira fornece uma ferramenta útil para explorar essas características com programas reais. Muitos dos exercícios a seguir requerem que os alunos continuem essa exploração em um laboratório.

Um tratamento formal do sistema de tipos e da semântica de Clite conclui o capítulo. Ele permite que os leitores interessados compreendam mais profundamente as dimensões matemáticas do projeto de linguagem.

EXERCÍCIOS

10.1 Considere as Regras de Tipo 10.4 e 10.5, nas quais a presença de um comando de retorno é fortemente associada com o fato de a função ser ou não ser `void`. Suponha que queiramos afrouxar essas regras um pouco, de modo que um comando de retorno pudesse aparecer em uma função `void` ou não-`void`. A presença de um comando retorno em uma função `void` simplesmente indicaria o retorno do controle ao solicitante, sem retornar um resultado.

(a) Que alterações na sintaxe concreta e abstrata seriam necessárias para essa mudança?
(b) Que alterações nas Regras de Tipos 10.4 e 10.5 seriam necessárias?
(c) Que alterações na Regra de Significado 10.2 seriam necessárias?

10.2 Usando as funções Clite de exemplo definidas nas Figuras 10.1 e 10.5, construa três chamadas diferentes; cada chamada deve violar uma das Regras de Tipo 10.6, 10.7 e 10.8, mas não as outras duas.

10.3 Considere o comando `k = k - 1;` dentro do laço `while` no programa da Figura 10.5. Remova esse comando do programa.

(a) Essa alteração causará um erro de tipo ou sintaxe para esse programa modificado? Explique.
(b) Agora execute o interpretador `Clite` sobre esse programa modificado. Descreva brevemente o que acontece.

(c) A Regra de Significado 10.1 pode de alguma forma ser alterada de modo que esse problema seja evitado? Em caso positivo, explique como. Se não, explique o porquê.

10.4 Associe cada uma das funções *V* de validação de tipos da Seção 10.2 com uma ou mais Regras de Tipo da Seção 10.4.2. Há alguma função *V* para a qual não exista Regra de Tipo ou vice-versa? Explique.

10.5 A Regra de Significado 10.1 pode ser revisada de modo que funções com efeitos colaterais não sejam permitidas? Explique.

10.6 Analise a Regra de Significado 10.1 da Seção 10.3 que define informalmente o significado de uma chamada de função Clite. Observe, todavia, que há duas funções diferentes *M* na Seção 10.4.3 que formalizam essa definição. Sugira como essas duas funções podem ser combinadas em uma única regra para que espelhe mais diretamente a Regra de Significado 10.1.

10.7 Complete a definição formal da verificação de tipos de Clite, integrando as regras de validação *V* definidas neste capítulo com aquelas apresentadas no Capítulo 6.

10.8 Complete a definição formal da semântica de Clite integrando as funções de semântica *M* definidas neste capítulo com aquelas apresentadas no Capítulo 8.

Gerenciamento de Memória

11

"C facilita um tiro no próprio pé; C++ dificulta, mas, quando você o faz, estoura toda a sua perna."

Atribuído a Bjarne Stroustrup

VISÃO GERAL DO CAPÍTULO

O gerenciamento em tempo de execução da memória dinâmica é uma atividade necessária para linguagens modernas de programação. Durante muito tempo, a automação dessa atividade tem sido um elemento-chave na implementação de linguagens de programação funcionais e lógicas, começando com Lisp, na década de 1960.

C e C++ permitiram que o gerenciamento em tempo de execução da memória dinâmica fosse assumido pelo programador ou pelo sistema. Embora existam diversas rotinas de gerenciamento automático de memória disponíveis para C e C++, seu uso pelo programador é inteiramente opcional. Java, por outro lado, insiste que o gerenciamento de memória é muito arriscado e propenso a erros para ser deixado para o programador; apenas o sistema pode ser responsável pelo gerenciamento da memória.

Em qualquer um dos casos, o gerenciamento de memória é uma área-chave no estudo de linguagens de programação, já que está muito relacionado ao significado dos programas. Embora a pilha em tempo de execução ajude a esclarecer nossa compreensão sobre como a memória é organizada para implementar funções, uma área diferente da memória, chamada de *heap*, nos ajuda a entender o comportamento em tempo de execução de objetos dinâmicos.

Este capítulo discute a implementação de matrizes dinâmicas e outros objetos usando técnicas convencionais para gerenciar memória *heap*. Particularmente importante nesse contexto é a noção de *overflow de heap*, junto a estratégias para lidar com isso quando ocorre em tempo de execução. Discutimos três algoritmos-chave para o gerenciamento de memória *heap* que são afetuosamente conhecidos como algoritmos de *coleta de lixo*.

11.1 A MEMÓRIA HEAP

Em linguagens como C, C++ e Java, a memória em tempo de execução pode ser visualizada como tendo três partes: a *área estática*, a *pilha de tempo de execução* e a *heap*.[1] Isso é mostrado na Figura 11.1.

A *memória estática* contém valores cujos requisitos de armazenamento são conhecidos antes do tempo de execução e permanecem constantes por toda a vida do programa em execução. Como vimos no Capítulo 10, a *pilha de tempo de execução* é o centro de controle para despachar funções ativas, variáveis declaradas localmente e a ligação parâmetro-argumento.

Em contraste, a *heap* contém valores que são alocados e estruturados dinamicamente enquanto o programa está sendo executado, como *strings*, matrizes dinâmicas, objetos e diversas estruturas dinâmicas de dados como listas encadeadas. Pela sua natureza, a *heap* se torna fragmentada à medida que é usada para a alocação e desalocação dinâmica de blocos de armazenamento de tamanhos diferentes. Por esse motivo, muitas linguagens usam algoritmos de coleta de lixo para gerenciar a memória *heap* de modo que o espaço disponível seja utilizado de forma eficiente. Abordamos a questão da coleta de lixo mais adiante neste capítulo.

Conforme a Figura 11.1 indica, o espaço de endereçamento de memória $\{0, \ldots, n\}$ é particionado de uma forma que antevê o comportamento dinâmico do programa. No início da execução, o tamanho da área estática é fixado, o topo da pilha é marcado pelo endereço $a - 1$, e o início e fim da *heap* são marcados pelos endereços fixos h e n.

Nesse modelo, a pilha cresce para cima em direção à extremidade superior do espaço de endereçamento. Em muitos sistemas reais, a pilha cresce na direção da extremidade inferior. A seguinte relação invariante deve ser mantida por toda a execução do programa, para evitar o surgimento de um erro de *overflow de pilha*.

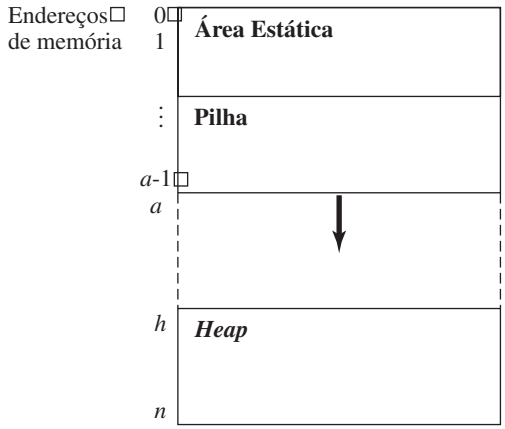
$$0 \leq a \leq h \leq n$$

Aqui, supomos que cada palavra da memória na *heap* possa ter um dentre três estados – *não usada, não definida* ou um valor elementar (como `int` ou `float`).[2] De forma intuitiva, *não usada* significa que a palavra não está alocada para o programa, enquanto *não definida* significa que está alocada para o programa, mas ainda não recebeu um valor. Uma visualização típica da *heap* é mostrada na Figura 11.2.

1. Outras linguagens tratam o gerenciamento de memória de forma diferente. Por exemplo, Fortran não se baseia na pilha de tempo de execução para controlar o programa. Programas em Fortran usam memória estática para controle de dados e programa e usam a *heap* principalmente para alocar matrizes de tamanho variável.

2. Na verdade, essa suposição é simplista. Por exemplo, uma palavra de memória pode, muitas vezes, guardar dois ou mais valores, e o valor *indefinido*, muitas vezes, não é usado como um valor inicial. Entretanto, manteremos essa suposição para que nossa discussão seja simples.

Figura 11.1
A Estrutura da Memória em Tempo de Execução



As funções *new* e *delete* de gerenciamento de memória *heap* permitem ao programa obter e liberar um bloco contíguo de palavras na memória *heap*. Blocos individuais de espaço em *heap* são endereçados usando variáveis ponteiros, cujos valores são endereços.

A função *new* retorna o endereço da primeira palavra de um bloco contíguo de *k* palavras *não usadas* e as marca como *indefinidas*.

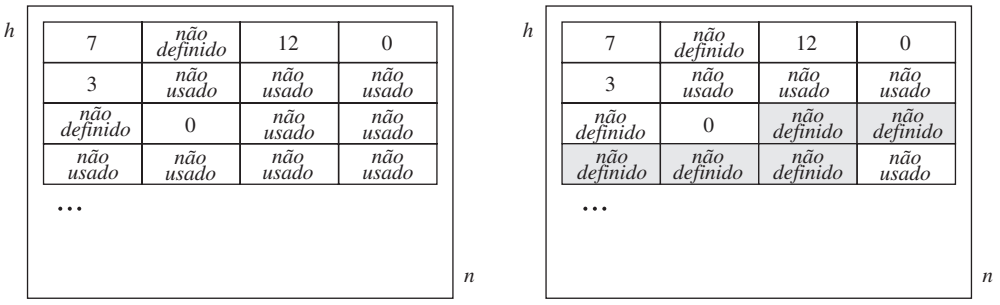
> **Definição:** *Overflow do heap* ocorre quando a função *new* é chamada e a *heap* não contém um bloco contíguo de palavras não usadas grande o suficiente para satisfazer a chamada.

Se tal bloco de *k* palavras não existir, a função *new* falha. Caso contrário, o valor retornado é um ponteiro para o início do bloco de *heap* assim alocado.

Por exemplo, o efeito da chamada à função *new*(5) é mostrada no lado direito da Figura 11.2, dado o estado *heap* arbitrário mostrado à esquerda. Os blocos sombreados à direita indicam os cinco blocos de *heap* que estão marcados como *indefinidos*. O endereço $h + 10$ é retornado por *new* neste exemplo.

A função *delete* desfaz o efeito de *new*, já que retorna para o estado *não usado* um bloco contíguo de palavras que pertençam a um determinado endereço.

Por exemplo, o efeito da chamada *delete*($h + 10$, 5) é mostrado na Figura 11.2 pela leitura da *heap* à direita como o estado inicial e da *heap* à esquerda como o estado final após a chamada.



| Figura 11.2 A Chamada à Função de Alocação de *Heap New*(5): Antes e Depois

11.2 IMPLEMENTAÇÃO DE MATRIZES DINÂMICAS

Objetos que requerem espaço em *heap* incluem matrizes de tamanho variável e estruturas de registros. É importante notar que matrizes e estruturas de tamanho variável nem sempre são alocadas na *heap*. Por exemplo, elas podem ser alocadas a um *frame* de pilha em tempo de execução. Se uma matriz ou uma pequena estrutura for passada para um parâmetro por valor, por exemplo, a alocação de pilha pode conter uma alternativa eficiente à alocação de *heap*.

Entretanto, com o interesse de introduzir o comportamento de objetos dinâmicos na *heap*, usaremos matrizes como exemplo. Esta é uma continuação da discussão sobre matrizes e estruturas iniciada no Capítulo 5, no qual primeiro fizemos referências oblíquas à *heap*. Aqui, esperamos mostrar mais integralmente o papel da *heap* quando matrizes dinâmicas são criadas e destruídas em tempo de execução.

Considere a necessidade de alocação de um bloco de memória para uma matriz cujo tamanho não seja conhecido até a execução, como na declaração a seguir:

```
int A[n];
```

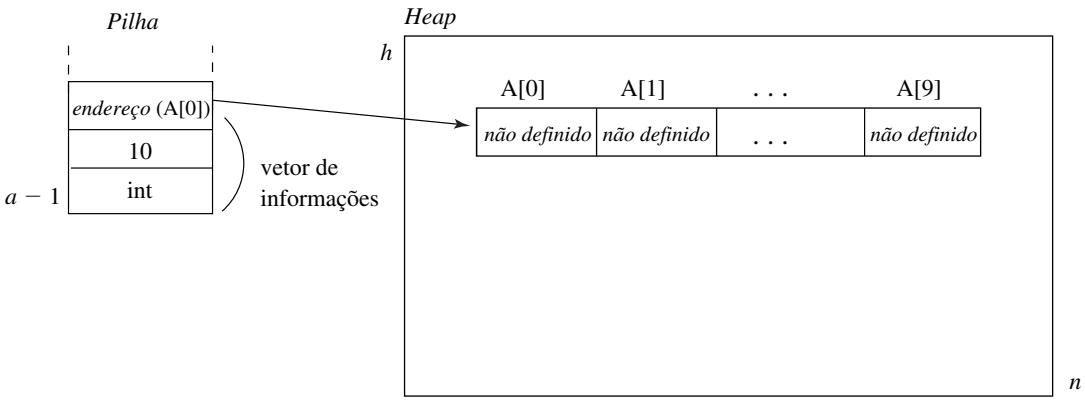
A alocação estática não é apropriada aqui. De modo geral, uma matriz requer espaço tanto na pilha quanto na *heap*. A pilha guarda uma referência ao endereço de um bloco de *heap* que guarda a primeira entrada da matriz, enquanto a *heap* guarda as entradas da matriz. Isso é ilustrado na Figura 11.3.

Já que a *heap* é usada, a função *new* entra em ação e a matriz aloca espaço conforme mostrado na Figura 11.3. Lá, vemos que o endereço de `A[0]` é armazenado na pilha junto ao resto do descritor de vetor da matriz, enquanto as entradas da matriz, `A[0]` até `A[n-1]`, são armazenadas em um bloco contíguo na *heap*.

A semântica das matrizes dinâmicas utiliza as funções *new* e *delete* discutidas anteriormente. A regra a seguir descreve a semântica de uma declaração de matriz.

Regra de Significado 11.1 *O significado de uma DeclaMatriz* `ad` *é:*

1. *Calcular o endereço* (`ad[0]`) = *new*(`ad.size`)*, no qual o valor da Expressão* `ad.size` *é calculado conforme descrito no Capítulo 5.*
2. *Empilhar o endereço* (`ad[0]`).



| Figura 11.3 Alocação de Espaço na Pilha e na *Heap* para a Declaração da Matriz A

3 *Empilhar* `ad.size`.
4 *Empilhar* `ad.type`.

Esta regra seria naturalmente incorporada ao significado de uma *Chamada* em Clite. Ou seja, onde o *frame* de pilha é criado para as declarações na função chamada. Uma *DeclaMatriz* poderia ser manipulada como um caso especial nessa configuração.

Observe que os passos 2–4 criam aqui o descritor de vetor para a *DeclaMatriz* `ad`, conforme ilustrado na Figura 11.3, para a matriz específica `A` com tamanho 10. Observe também que a função *new* usada no passo 1 pode ou não ser bem-sucedida. Se ela não o for, uma condição de exceção chamada *overflow do heap* ocorre, e a matriz não pode ser alocada imediatamente. Retornaremos a esse problema na próxima seção.

Quando uma função em um programa em Clite retorna o controle para quem a chamou, qualquer matriz alocada dinamicamente para essa função deve ser desalocada, restaurando-se todas as entradas no seu bloco de *heap* para o estado *não usado*. Assim, a semântica de uma chamada de função deve ser alterada para incorporar essa consideração. Tal mudança naturalmente utilizaria a função *delete* discutida anteriormente. Os detalhes dessa mudança são deixados como um exercício.

A semântica da avaliação de expressões em Clite pode ser estendida para cobrir referências a matrizes da seguinte forma:

Regra de Significado 11.2 *O significado de uma RefMatriz* `ar` *é*:
1 *Calcular o endereço*(`ad[ar.index]`) = *endereço*(`ad[0]`) + `ad.index` − 1.
2 *Se o endereço*(`ad[0]`) ≤ *endereço*(`ad[ar.index]`) < *endereço*(`ad[0]`) + `ad.size`, *retorne o valor no endereço*(`ad[ar.index]`).
3 *Caso contrário, sinalize um erro de índice fora de faixa.*

Observe que essa definição inclui a verificação de faixa em tempo de execução para referências de matrizes. Ela usa o vetor de informação para a matriz, conforme armazenado na pilha de tempo de execução, quando a declaração da matriz é processada pela Regra de Significado 11.1.

Por exemplo, suponha que tenhamos a *RefMatriz* A[5]. Então o valor de A[5] é recuperado do bloco de *heap* alocado para A, recuperando o endereço de A[0], adicionando 4 (denotado pelo *endereço*(`ad[ar.index]`) − 1) e recuperando o valor lá armazenado.

Finalmente, o significado de um comando de *Atribuição* `as`, cujo `alvo` é uma *RefMatriz* `ar`, pode ser definido da seguinte maneira:

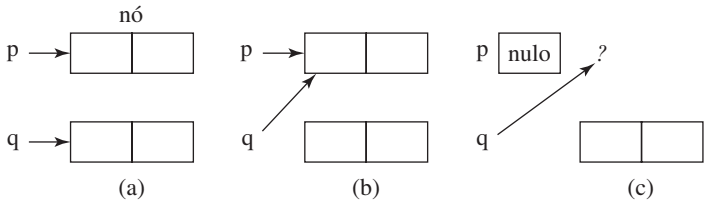
Regra de Significado 11.3 *O significado de uma Atribuição* `as` é:
1 *Calcular o endereço*(`ad[ar.index]`) = *endereço*(`ad[0]`) + `ad.index` − 1.
2 *Se o endereço*(`ad[0]`) ≤ endereço(`ad[ar.index]`) < *endereço*(`ad[0]`) + `ad.size`, *então altere o valor no endereço*(`ad[ar.index]`) *para o valor de* `as.source`.
3 *Caso contrário, sinalize um erro de índice fora de faixa.*

Por exemplo, suponha que tenhamos a atribuição `A[5] = 3`. Aqui, trocaremos o valor na *heap* endereçada por A[5] (ou seja, *endereço*(`ad[0]` + 4) pelo valor da expressão 3 (denotada por `as.source`).

11.2.1 Problemas de Gerenciamento de *Heap*: Lixo

Lixo é qualquer bloco do *heap* de memória que não possa ser acessado pelo programa, ou seja, não há um ponteiro acessível para o programa cujo valor referencie esse bloco.

Lixo pode ser criado facilmente. Por exemplo, pode ocorrer quando uma matriz alocada dinamicamente, como `A` na seção precedente, tem seu descritor de vetor removido da pilha (quando o *frame* da pilha é desempilhado) antes que seu bloco de *heap* seja retornado por meio de um *delete*.



| Figura 11.4 Criando Órfãos e Viúvas: Um Exemplo Simples

Para uma ilustração melhor, analise o seguinte exemplo simples do tipo Java, que inicia a criação de nós em uma lista encadeada:

```
class node {
   int value;
   node next;
}
...
node p, q;
p = new node();
q = new node();
q = p;
```

A última declaração cria um "vazamento de memória" conforme mostrado na Figura 11.4b. Nessa situação, o nó que havia sido referenciado por q na Figura 11.4a agora se tornou *órfão* ou lixo. Além disso, adicionar no final do programa a seguinte declaração

```
delete(p);
```

cria uma referência pendente, ou *viúva*, de q, conforme ilustrado na Figura 11.4c.

C e C++ deixam a responsabilidade da coleta de lixo para o programador, ou seja, elas fornecem ferramentas explícitas (como a função `delete` anterior) para retornar explicitamente blocos de *heap* ao sistema para reuso.[3] Java, Lisp e outras linguagens fornecem coleta automática de lixo e tiram a responsabilidade pelo gerenciamento da memória *heap* das mãos dos programadores.

11.3 COLETA DE LIXO

Conforme mencionado no início do capítulo, algoritmos de coleta de lixo têm sido usados há mais de 40 anos nas comunidades de programação funcional e lógica. Uma renovada mudança de atenção em relação à coleta de lixo ocorreu em outras comunidades com a emergência da programação orientada a objeto na década de 1990. Essa mudança fez com que se reconhecesse a necessidade de gerenciamento de *heap* efetiva, já que ela é o local onde os objetos residem em tempo de execução.

Conforme sugerido na Seção 11.2, a *heap* contém tanto objetos ativos quanto inativos – blocos que haviam sido alocados durante um estágio anterior à execução do programa e depois renegados ao *status* de "não mais necessários". De modo geral, objetos

3. Implementações recentes de C e C++ começaram a fornecer opções de coleta de lixo, isentando, assim, o programador dessa responsabilidade.

inativos são conhecidos como *lixo*, já que continuam a ocupar espaço valioso da *heap* após sua vida útil ter expirado.

Assim que a *heap* fica cheia, é importante ter uma estratégia para recuperar todos os blocos de memória que sejam lixo, de modo que esses blocos possam ser reusados por novos objetos que requeiram blocos de *heap* para si. Por exemplo, a semântica da alocação dinâmica de matrizes discutida na Seção 11.2 não funcionará normalmente quando a *heap* sofrer *overflow* (como na chamada *new*(*n*) quando um bloco contíguo de *n* palavras não estiver disponível).

O termo *coleta de lixo* refere-se a qualquer estratégia para recuperação de blocos de *heap* não usados para uso posterior pelo programa. Diversos algoritmos de coleta de lixo foram usados com mais ou menos sucesso nos últimos 40 anos. As primeiras linguagens que precisaram de coleta de lixo foram as funcionais (notadamente Lisp), já que suas atividades de tempo de execução eram dominadas pela necessidade de criar, destruir e alterar dinamicamente a estrutura de listas encadeadas. Os principais algoritmos de coleta de lixo em uso atualmente são derivados daqueles das linguagens funcionais.

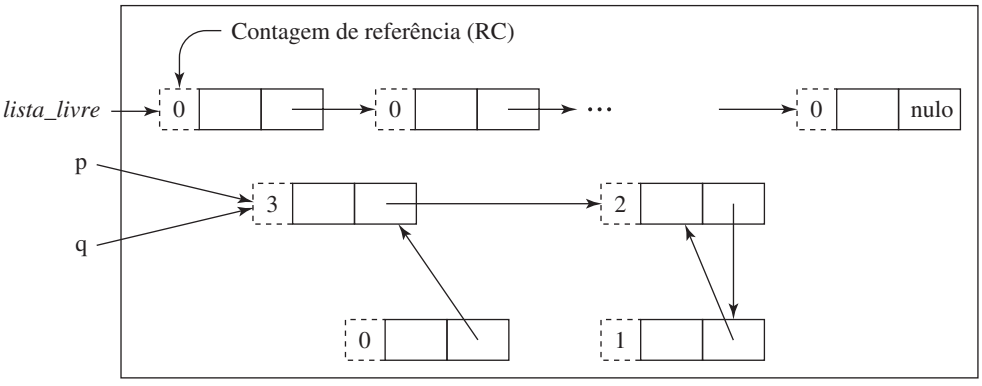
Três estratégias importantes serviram como base para os modernos algoritmos de coleta de lixo: *contagem de referências, marcar-varrer* e *coleção de cópias*. Resumimos essas três estratégias nas seções seguintes.

11.3.1 Contagem de Referência

A *contagem de referência* supõe que a *heap* inicial seja uma cadeia contínua de nós chamados de *lista_livre* (*free_list*), na qual cada nó tem um campo inteiro adicional que contém um contador do número de ponteiros referenciando esse nó (inicialmente 0). À medida que o programa é executado, os nós são tirados da lista livre e conectados entre si por meio de ponteiros, formando, assim, cadeias. A Figura 11.5 é um exemplo de *heap* na qual o contador de referência (RC) de cada nó é mostrado como um inteiro dentro de uma linha pontilhada.

Nesse momento, há quatro nós e três estão ativos, e a *lista_livre* contém todos os outros nós na *heap* que estão disponíveis para uso futuro. Assim, três nós estão diretamente ou indiretamente acessíveis a partir de p ou q (que se supõe que sejam variáveis estáticas globais ou variáveis locais alocadas dentro de um *frame* de pilha).

O algoritmo da contagem de referência é aplicado dinamicamente, ou seja, sempre que uma operação `new` ou `delete` ocorre em um ponteiro, o algoritmo é ativado e tantos órfãos quanto for possível são retornados para a *lista_livre*. Neste esquema, um órfão é



| Figura 11.5 Estrutura de Nós e *Heap* Exemplo para a Contagem de Referências

simplesmente um nó cujo contador de referências se torna zero, ou algum de seus descendentes (na cadeia) cuja contagem de referências se torna zero como resultado de uma operação `delete`.

Quando uma operação `new` ocorre no programa, o algoritmo aloca um nó de *heap* da *lista_livre* e inicializa seu contador de referências com 1 (já que é a primeira referência àquele nó). Sempre que uma atribuição de ponteiro ocorre no programa, como

```
q = p;
```

alguma contabilidade sobre a contagem de referências deve ser feita:

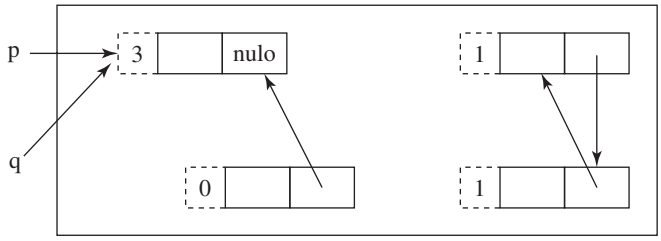
1. O contador de referências para o nó de p é incrementado em 1.
2. O contador de referências para o nó de q é decrementado em 1.
3. Se o contador de referências para o nó de q se tornar 0, os contadores de referências para cada um de seus descendentes são diminuídos em 1, o nó de q é retornado para a *lista_livre* e esse passo é repetido para cada um dos descendentes do nó de q.
4. Ao ponteiro q é atribuído o valor (referência) de p.

Quando uma operação `delete q` ocorre, o passo 3 no algoritmo anterior é aplicado e o nó de q é retornado para a *lista_livre*.

Um exame cuidadoso do algoritmo de contagem de referência revela sua fraqueza fundamental como coletor de lixo, ou seja, ele falha em retornar à *lista_livre* qualquer lixo que ocorra na forma de uma cadeia circular isolada. Por exemplo, observe o efeito da aplicação do algoritmo anterior à *heap* mostrada na Figura 11.5, quando a seguinte declaração é executada:

```
p.next = null;
```

Esse comando deixa a *heap* da Figura 11.5 ativa na seguinte condição:



Aqui, dois nós estão em uma cadeia isolada apontando um para o outro (cada uma das suas contagens de referências é 1). Entretanto, o algoritmo não pode retornar nenhum deles para a *lista_livre*, já que nenhuma contagem de referência de algum nó é 0. Um tratamento mais detalhado desse algoritmo é dado como um exercício.

De modo geral, a contagem de referência possui uma importante vantagem e três importantes desvantagens como algoritmo de coleta de lixo. A vantagem é que ele ocorre dinamicamente, sempre que uma atribuição de ponteiro ou uma ação na *heap* for disparada pelo programa. Assim, o *overhead* associado à coleta de lixo é naturalmente distribuído pelo tempo de execução do programa. Suas desvantagens são (1) sua falha em detectar cadeias circulares inacessíveis, conforme ilustrado anteriormente, (2) o *overhead* de armazenamento criado pela inserção de contador inteiro de referências a cada nó da *heap* e (3) o *overhead* de desempenho criado pelo algoritmo anterior sempre que um ponteiro for atribuído ou um bloco de *heap* for alocado ou desalocado.

11.3.2 Marcar-Varrer

Diferentemente da contagem de referências, o algoritmo *marcar-varrer* é chamado apenas quando a *heap* está cheia (por exemplo, a *lista_livre* está vazia). Assim, diversos milhares de atribuições de ponteiros e alocações e desalocações de blocos de *heap* podem ocorrer sem que aconteça *overhead* de coleta de lixo. Assim que *marcar-varrer* é chamado, entretanto, um processo mais elaborado e demorado interrompe o programa, conforme explicado a seguir.

Uma configuração inicial de exemplo para o algoritmo de *marcar-varrer* é mostrada na Figura 11.6, na qual a linha pontilhada contorna um "bit de marca" (MB) que é anexado a cada nó de *heap* e inicializado como 0.

O algoritmo de *marcar-varrer* executa duas passagens pela *heap*. Ele é chamado sempre que um novo nó é solicitado pelo programa e a *heap* está cheia. Assim, quando a atribuição à esquerda é executada, o significado à direita é deduzido:

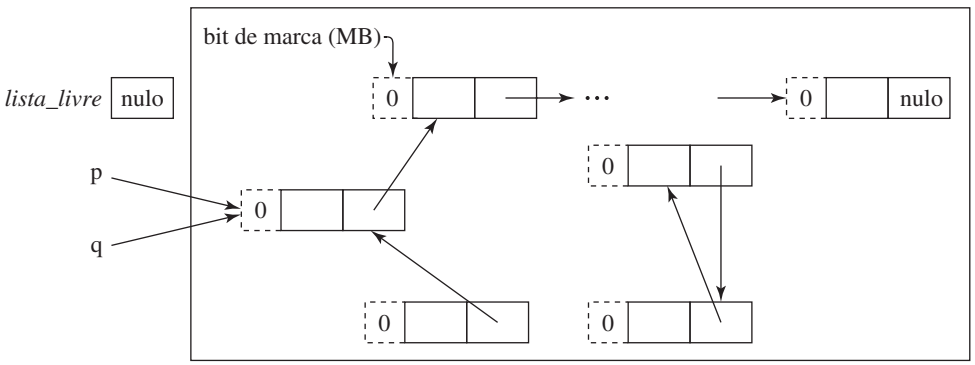
```
q = new node();      if (free_list == null)
                        mark_sweep();
                     if (free_list != null) {
                        q = free_list;
                        free_list = free_list.next;
                     }
                     else abort('heap full')
```

Aqui, a cláusula *else* final reflete o evento de que a coleta de lixo falha em recuperar alguns blocos para a `lista_livre`, o que força o término anormal do programa.

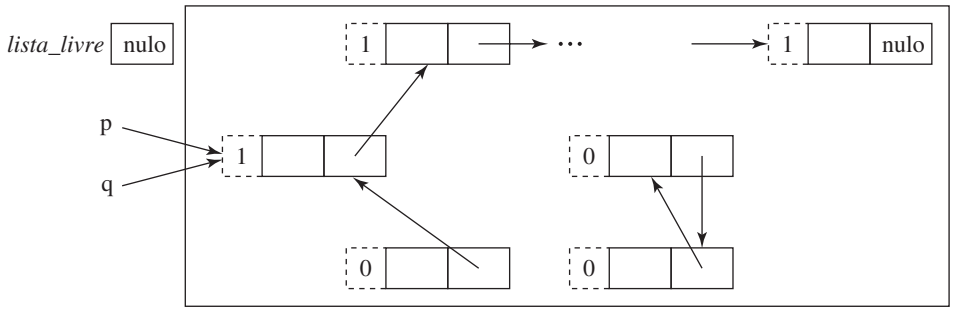
Durante a Passagem I, a passagem "marcar", cada bloco da *heap* que possa ser alcançado seguindo-se uma cadeia de ponteiros originada na pilha de tempo de execução é marcado como acessível (seu MB é configurado como 1), ou seja, todas as referências `R` na pilha de tempo de execução para endereços na *heap* fazem com que a seguinte função seja chamada:

```
Mark(R):     if (R.MB == 0)
                R.MB = 1;
             if (R.next != null)
                Mark(R.next);
```

Todos os blocos que permanecem não marcados (MB = 0) na Passagem I são então identificados como órfãos.



| Figura 11.6 Estrutura de Nós e Exemplo para o Algoritmo *Marcar-Varrer*



| Figura 11.7 ***Heap* após a Passagem I do *Marcar-Varrer***

Em nosso exemplo, suponha que `p` e `q` são as únicas variáveis da pilha que referenciam a *heap*. A Passagem I deixaria a *heap* na configuração mostrada na Figura 11.7.

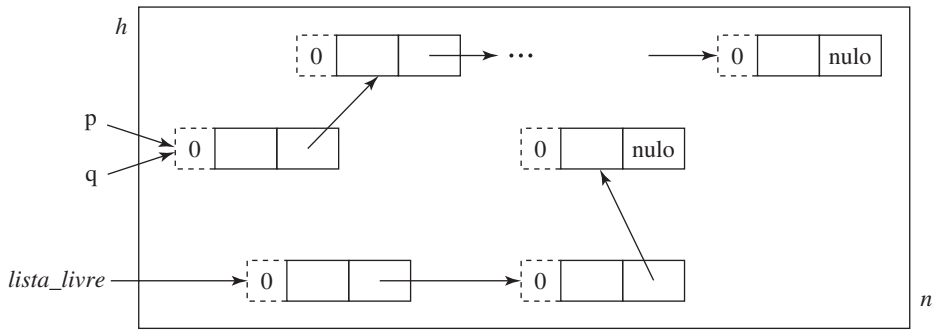
A Passagem II, passagem "varrer", atravessa a *heap* inteira, retornando todos os nós não marcados para a *lista_livre* e desmarcando todos os blocos que tenham sido marcados na Passagem I. Lembre-se de que *h* e *n* denotam as extremidades inferior e superior da *heap*, conforme definido originalmente na Figura 11.1.

```
Sweep():   i=h;
           while (i<=n) {
              if (i.MB == 0)
                 free(i);
              else i.MB = 0;
              i = i + 1;
           }
```

A função `free(N)` restaura o nó referenciado por `N` para a *lista_livre*, ou seja,

```
free(N):   N.next = free_list;
           free_list = N;
```

O resultado da Passagem II na nossa *heap* de exemplo é mostrado na Figura 11.8. Aqui supomos que o início e o final da *heap* estejam nos cantos superior esquerdo e



| Figura 11.8 ***Heap* após a Passagem II de Marcar-Varrer**

inferior direito da figura, de modo que os nós mais abaixo que aparecem na figura são os últimos a serem retornados para a *lista_livre*.

Agora que a *lista_livre* não está vazia, a atribuição original `q = new node()` que chamava o algoritmo *marcar-varrer* pode continuar e ser completada. As principais vantagens de *marcar-varrer* sobre a contagem de referências são: (1) ele recupera *todo* o lixo da *heap* e (2) só é chamado quando a *heap* está cheia.

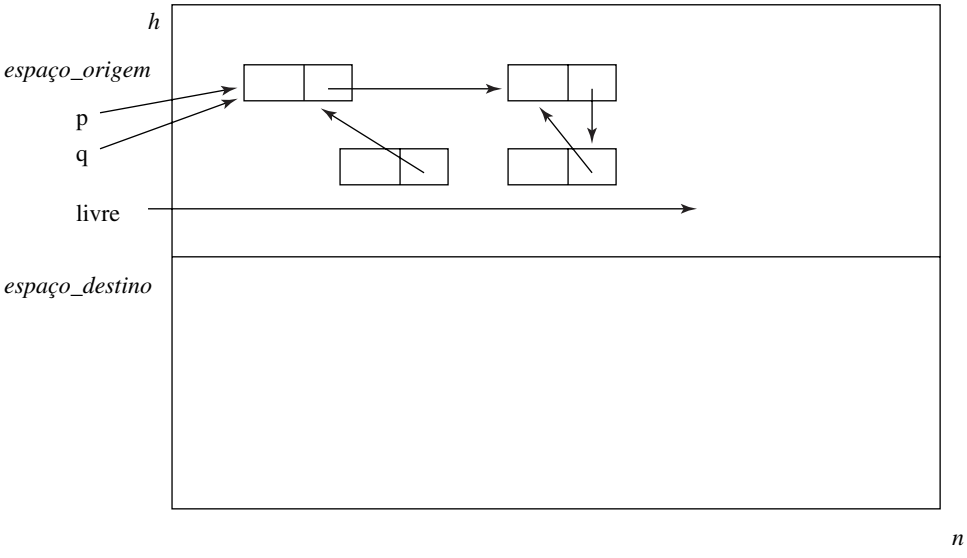
11.3.3 Coleta de Cópias

A coleta de cópias representa um tipo de comprometimento tempo–espaço quando comparada com o *marcar-varrer*. Da mesma forma que no *marcar-varrer*, a coleta de cópias é chamada apenas quando a *heap* está cheia. Diferentemente do *marcar-varrer*, a coleta de cópias só executa uma passagem sobre a *heap*, de modo que é significativamente mais rápida. Entretanto, a coleta de cópias requer significativamente mais memória porque apenas metade do espaço da *heap* está disponível ativamente para alocar novos blocos de memória para o programa de uma vez.

Para a coleta de cópias, a *heap* é dividida em dois blocos idênticos, denominados *espaço_origem* e *espaço_destino*, conforme mostrado na Figura 11.9. Aqui, nenhum campo extra é necessário em cada nó para um contador de referências ou bit de marca e nenhuma *lista_livre* é necessária. Em vez disso, um ponteiro `livre` é mantido para o final da área alocada no *espaço_origem*.

No início da execução do programa, todos os nós ativos são residentes no *espaço_origem*, e o *espaço_destino* não é usado. A inicialização da *heap* é executada pelo seguinte algoritmo:

```
espaço_origem = h;
topo_do_espaço = h + (n - h)/2;
espaço_destino = topo_do_espaço + 1;
livre = espaço_origem;
```



| Figura 11.9 Organização da *Heap* antes da Coleta de Cópias

Sempre que um novo nó é alocado durante a execução do programa, o próximo bloco referenciado como `livre` é alocado e o valor de `livre` é atualizado. Se nenhum estiver disponível, a coleta de cópias é chamada, fazendo com que os blocos alocados sejam copiados e reempacotados no *espaço_destino*, e os papéis de *espaço_origem* e de *espaço_destino* são invertidos. Ou seja, quando a declaração abaixo à esquerda do programa for executada, as ações à direita são chamadas:

```
q = new node();   if (livre + 1 > topo_do_espaço)
                     girar();
                  if (livre + 1 > topo_do_espaço)
                     abort('heap cheia')
                  q = livre;
                  livre = livre + 1;
```

As funções *girar* e *copiar* desempenham essa reversão de papéis da seguinte forma:

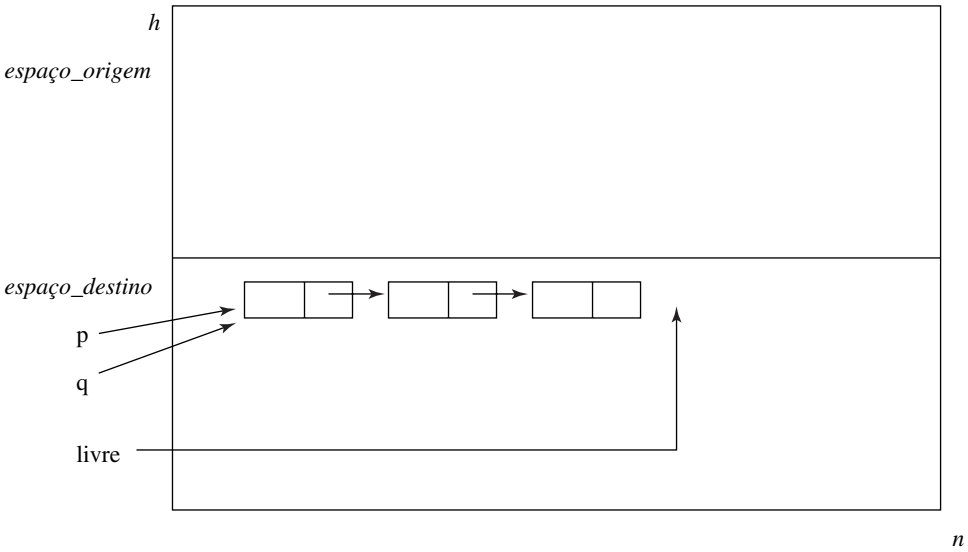
```
girar():  espaço_origem, espaço_destino = espaço_destino, espaço_origem;
          top_of_space = to_space + (n - h)/2;
          livre = espaço_destino;
          for R in raízes
              R = copiar(R);
copiar(R):    if (atômico(R) || R == null)
              return R;
          if (!enviado(R)) {
              R'= livre;
              livre = livre + 2;
              endereço_de_envio(R) = R';
              R'= copiar(R.value);
              R'+1 = copiar(R.next);
          }
          return endereço_de_envio(R);
```

`Girar` reatribui o ponteiro `topo_de_espaço`, e `copiar` reempacota os blocos ativos para o início do *espaço_destino*. Durante esse processo, um nó referenciado `R` no *espaço_origem* pode ser "enviado" (copiado para o *espaço_destino*) ou não. Se ele tiver sido enviado, cada referência sucessiva ao nó `R` recebe seu novo endereço no *espaço_destino*. Se não tiver sido enviado, `copiar(R)` o copia para o *espaço_destino* usando o próximo nó disponível na *lista_livre*. Esse é um processo recursivo, já que o nó R pode ter descendentes no *espaço_origem* que precisem ser enviados.

A coleta de cópias elimina, portanto, todos os nós inacessíveis no *espaço_origem* e reempacota os nós ativos no *espaço_destino*, conforme mostrado na Figura 11.10.

11.3.4 Comparação das Estratégias

A comparação de desempenho entre a coleta de cópias e o *marcar-varrer* tem sido cuidadosamente estudada, incluindo uma discussão detalhada sobre algoritmos de coleta de lixo e seus desempenhos (Jones e Lins, 1996). Essas comparações são resumidas da seguinte forma: se R for o número de blocos ativos da *heap* e a "residência" r for definida como a taxa de R para o tamanho da *heap* $n - h$, então a eficiência e (medida como a quantidade de memória recuperada por unidade de tempo) é maior para a coleta de cópias



| Figura 11.10 Reorganização da *Heap* após a Coleta de Cópias

quando r for muito menor do que $(n - h)/2$. Entretanto, *marcar-varrer* se torna mais eficiente do que a coleta de cópias quando r se aproxima de $(n - h)/2$.

Dúzias de algoritmos diferentes de coleta de lixo estão em uso atualmente, e a maioria deles é mais complexa do que as três estratégias simples discutidas anteriormente. Por exemplo, algumas implementações de linguagens usam uma estratégia híbrida, selecionando entre *marcar-varrer* e *coleta de cópias*, dependendo da residência r na *heap* quando a coleta de lixo é chamada.

No sistema Java da Sun, o coletor de lixo é executado como um processo de prioridade baixa chamado de *"thread"*, que é executado sempre que a demanda por tempo de processamento de outros *threads* é baixa.[4] Isso reduz o número de chamadas de coleta de lixo necessárias durante os tempos de pico de processamento. Também em Java, o programa pode chamar explicitamente o coletor de lixo usando a chamada `System.gc()`. Isso chama a coleta de lixo imediatamente, independentemente do estado da *heap* no momento da chamada.

11.4 RESUMO

Neste capítulo estudamos o importante assunto de gerenciamento de memória para objetos dinâmicos em linguagens de programação. A semântica das declarações de matrizes dinâmicas e referências são usadas para ilustrar os princípios mais importantes. A semântica de verificação de índices de matrizes em tempo de execução também é apresentada.

Este capítulo dá atenção especial a situações excepcionais que ocorrem na memória dinâmica, incluindo vazamentos de memória, viúvas e órfãos. Devido ao fato de os algoritmos de coleta de lixo terem se tornado importantes em todos os paradigmas de programação, esses algoritmos foram naturalmente incluídos nessa discussão.

4. Discutiremos *threads* Java em detalhes no Capítulo 17.

EXERCÍCIOS

11.1 C possui as funções `malloc` e `free`, que permitem ao programador alocar e desalocar dinamicamente espaço na *heap*. Pesquise essas duas funções e compare suas semelhanças e diferenças com as funções *new* e *delete* discutidas neste capítulo.

11.2 Determine o estado atual da coleta de lixo na versão corrente da linguagem Perl. A coleta de lixo é interna na linguagem? Há algoritmos de coleta de lixo separados disponíveis para programadores Perl? Como os programadores Perl normalmente gerenciam vazamentos de memória *heap*, viúvas e órfãos?

11.3 Identifique todos os erros possíveis em tempo de execução que podem ocorrer para a declaração de matriz ou referência `A[n]` quando ela é encontrada durante a interpretação de um programa.

11.4 Índices de matrizes em Clite devem ser verificados em tempo de execução, já que o tamanho de uma matriz não pode ser determinado estaticamente. Use os erros de indexação que você identificou na questão anterior, junto ao modelo baseado em *heap* para alocação de matrizes discutido neste capítulo, como uma base para suas definições de regras expandidas.

11.5 Expanda a regra de significado para uma *Chamada* em Clite para incorporar a idéia de alocação de memória *heap* e a recuperação para um parâmetro de matriz ou variável local. Essa mudança naturalmente utilizará as funções *new* e *delete* discutidas no início deste capítulo.

11.6 Matrizes dinâmicas em Clite podem ser alocadas na pilha de tempo de execução em vez de ser alocadas na *heap*. Discuta o impacto dessa escolha sobre a semântica da declaração e as referências de matrizes usando o *framework* semântico apresentado no Capítulo 10.

11.7 Quais são os limiares, em tempo e espaço, quando a alocação de matrizes dinâmicas ocorre na pilha de tempo de execução em vez de ocorrer na *heap*?

11.8 (Projeto de Equipe) Estenda a implementação de Clite de modo que suporte matrizes dinâmicas. Use a sintaxe concreta e abstrata, as regras de verificação de tipos e as regras de significados para matrizes que são definidas neste e nos capítulos anteriores. Isso requererá a extensão da definição de estado de modo que a memória inclua a *heap*, os limites *m e n* de *heap* e as funções de gerenciamento de *heap new* e *delete*.

11.9 (Projeto de Equipe) Projete um algoritmo de coleta de lixo de contagem de referências em C++ usando a discussão e a estrutura de nó elementar apresentada neste capítulo. *Dica*: suponha que a faixa de endereços da *heap* seja {*h*, ..., *n*} e que cada nó possua três unidades endereçáveis de memória: uma para o contador de referências, uma para o valor armazenado e uma para um ponteiro para outro nó na *heap*. Analise as seguintes funções auxiliares:

`free(N)`: restaura o nó referenciado por N para a *lista_livre.*

`delete(N)`: decrementa o contador de referências para o nó N; se o seu contador de referências se tornar 0, então `delete(N.next)` e `free(N)`. (Esta é uma recursiva.)

`N = M : delete(N)`, incrementa o contador de referências para M e atribui N = M.

12 Programação Imperativa

"Eu realmente odeio esta maldita máquina; eu queria que eles a vendessem. Ela não faz o que quero que faça, só o que lhe digo para fazer."

Lamento do programador (anônimo)

VISÃO GERAL DO CAPÍTULO

A programação imperativa é o paradigma de programação mais antigo e bem desenvolvido. Ela surgiu com os primeiros computadores, na década de 1940, e seus elementos espelham diretamente as características arquiteturais dos computadores modernos também.

Neste capítulo, discutimos os recursos-chave das linguagens imperativas. A seguir, enfocamos o papel que as bibliotecas de funções têm desempenhado. Concluímos o capítulo analisando três linguagens muito diferentes: C, Ada e Perl.

12.1 O QUE TORNA UMA LINGUAGEM IMPERATIVA?

Em meados da década de 1940, John von Neumann e outros reconheceram que tanto um programa quanto os seus dados poderiam residir na memória principal de um computador, uma idéia implícita no trabalho inicial de Turing (Turing, 1936). Os primeiros computadores armazenavam seus programas fora da memória, geralmente usando um painel de ligações com fios. A idéia de armazenar um programa na memória do computador levou a um aumento enorme do poder e da versatilidade potenciais de um computador.

A arquitetura do assim chamado modelo de von Neumann-Eckert (veja o Capítulo 1) é a base para o paradigma da programação imperativa. A memória da máquina contém tanto instruções de programas (o *armazenamento de programa*) quanto valores de dados (o *armazenamento de dados*). No coração dessa arquitetura está a idéia de *atribuição* – alterar o valor de um local de memória e destruir seu valor anterior.

Já que elas surgiram do modelo de von Neumann-Eckert, todas as linguagens imperativas incluem a atribuição como um elemento central. Além disso, elas suportam declarações de variáveis, expressões, comandos condicionais, laços e abstração procedural. Declarações atribuem nomes a locais de memória e associam tipos aos valores armazenados. As expressões são interpretadas por meio da recuperação dos valores correntes das variáveis com nomes a partir das suas respectivas localizações na memória e pelo cálculo de um resultado a partir desses valores. Dada uma referência a uma variável `x`, a memória retorna o valor corrente no local associado a `x`.

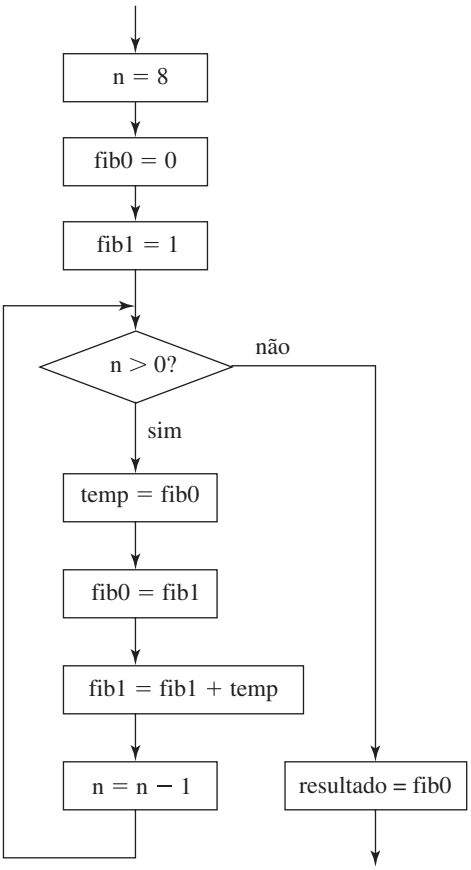
Os comandos são executados normalmente na ordem em que aparecem na memória, embora comandos de ramificações condicionais e incondicionais possam interromper esse fluxo normal de execução. Devido ao seu uso extensivo de ramificações, os primeiros programas imperativos eram, muitas vezes, modelados com o auxílio de um tipo especial de grafo conhecido como *diagrama de fluxo*. Um diagrama de fluxo de exemplo para o programa Fibonacci, discutido em capítulos anteriores, aparece na Figura 12.1.

Originalmente, os comandos de uma linguagem imperativa eram abstrações simples das instruções em máquinas de von Neumann-Eckert padrão; esses comandos englobam os comandos de atribuição, comandos condicionais e comandos de ramificação. Comandos de atribuição forneciam a capacidade de se atualizar dinamicamente o valor armazenado em um local de memória, enquanto os comandos condicionais e de ramificação podiam ser combinados para permitir que um conjunto de comandos fosse pulado ou executado repetidamente. Essas construções sozinhas tornam a linguagem *completa quanto a Turing.*[1]

Uma linguagem imperativa é dita *completa quanto a Turing* se fornecer uma base efetiva para a implementação de qualquer algoritmo que possa ser projetado. Linguagens imperativas que contêm variáveis e valores inteiros, operações aritméticas básicas, comandos de

1. A característica de ser "completa quanto a Turing" é importante porque fornece uma medida pela qual um conjunto mínimo de recursos de uma linguagem pode expressar todos os algoritmos que possam ser concebidos. Essa característica não é exclusiva das linguagens imperativas – linguagens funcionais, lógicas e orientadas a objeto também são completas quanto a Turing, no sentido de que qualquer uma dessas linguagens é igualmente capaz de expressar qualquer algoritmo.

Figura 12.1
Exemplo de Diagrama de Fluxo para Calcular Números de Fibonacci



atribuição, seqüenciamento de comandos baseados em memórias, condições e comandos de ramificação são "completas quanto a Turing".

A linguagem Clite contém todas essas características, exceto os comandos de ramificação. Entretanto, Dijkstra (1968b) mostrou que o uso excessivo de comandos de ramificação (ou "go to") era prejudicial ao processo de desenvolvimento de programas confiáveis. Não obstante, é bem sabido que qualquer seqüência de comandos em um programa imperativo que incluísse comandos de ramificação poderia ser escrita de forma equivalente, usando apenas laços condicionais e *while* no seu lugar. Assim, Clite é "completa quanto a Turing".

Uma *linguagem de programação imperativa* é, assim, a que é "completa quanto a Turing" e também suporta determinadas características comuns que surgiram com a evolução do paradigma de programação imperativa:

- Estruturas de controle.
- Entrada/saída.
- Manipulação de exceções e erros.
- Abstração procedural.
- Expressões e atribuição.
- Suporte de biblioteca para estruturas de dados.

As três primeiras dessas características foram discutidas exaustivamente em capítulos anteriores. O restante é discutido a seguir. Juntas, todas essas seis características aparecem em uma diversidade de linguagens e uma ampla gama de aplicações complexas. Este capítulo aborda o uso dessas características, por meio dos exemplos de C, Ada e Perl.

12.2 ABSTRAÇÃO PROCEDURAL

No paradigma de programação imperativa, os programas são modelados como "algoritmos mais estruturas de dados" (Wirth, 1976). Os algoritmos são desenvolvidos em programas por intermédio de duas idéias complementares: abstração procedural e refinamento gradual.

Bem no início do desenvolvimento das linguagens imperativas, o valor de bibliotecas e funções reusáveis foi reconhecido. Mesmo na década de 1950, Fortran e Algol incluíam uma quantidade de funções matemáticas padrão, inclusive `sin`, `cos`, `sqrt`, entre outras. Os programadores exploram a disponibilidade de funções predefinidas com o uso da abstração procedural.

> **Definição:** O processo de *abstração procedural* permite ao programador se preocupar principalmente com a interface entre a função e o que ela calcula, ignorando os detalhes de como o cálculo é executado.

Uma forma sistemática de produzir um programa (Wirth, 1973) é chamada de *refinamento em passos* (ou, às vezes, *decomposição funcional*), ou seja, o programador começa com uma descrição da função calculada por um programa, junto a sua entrada e a sua saída, e então divide a função em um conjunto de funções mais primitivas, por meio de seqüenciamento, iteração e seleção. Esse processo é repetido até que todas as funções a serem calculadas e seus dados possam ser executados pelos comandos e tipos de dados primitivos da própria linguagem.

> **Definição:** O processo de *refinamento gradual* utiliza abstração procedural desenvolvendo um algoritmo da sua forma mais geral para uma implementação específica.

Um exemplo familiar é o desenvolvimento de uma função de ordenação, na qual o programa precisa de um algoritmo para ordenar uma matriz de números, ignorando os detalhes sobre como essa ordenação é executada. Assim, qualquer implementação dada do algoritmo de ordenação poderia ser substituída posteriormente por uma diferente. A interface de tal rotina de ordenação seria:

```
sort(list, len);
```

na qual `list` é a matriz de números a ser ordenada e `len` contém o número de números na lista.

Uma primeira implementação desse algoritmo poderia ser expressa como:

```
foreach i in the sequence of indices of list {
    list[i] = minimum element in remaining list
}
```

O segundo comando poderia ser então refinado como:

```
foreach i in the sequence of indices of list {
    foreach j > i in the sequence of indices of list {
        list[i], list[j] = min, max of list[i], list[j]
    }
}
```

Reconhecendo que o terceiro comando pode ser executado por um if e uma troca, obtemos:

```
foreach i in the sequence of indices of list {
    foreach j > i in the sequence of indices of list {
        if list[j] < list[i] { swap list[i], list[j] }
    }
}
```

Esse algoritmo refinado agora assume uma forma que pode ser codificada razoavelmente de forma direta em uma linguagem imperativa. Uma codificação disso no estilo de C poderia se parecer com:

```
void sort(Type list, int len) {
    for (int i = 0; i < len; i++)
        for (int j = i+1; j < len; j++)
            if (list[j] < list[i]) {
                Type t = list[j];
                list[j] = list[i];
                list[i] = t;
            }
}
```

Esse é o coração do refinamento gradual em uma linguagem imperativa. Embora não mostrados aqui, níveis adicionais de abstração procedural poderiam ter sido usados durante esse processo de refinamento, por exemplo, para abstrair as últimas três linhas do código anterior em uma chamada para um procedimento `swap`.

12.3 EXPRESSÕES E ATRIBUIÇÃO

Fundamental para todas as linguagens imperativas é o comado de atribuição, que assume a forma geral:

```
alvo = expressão
```

Existe uma variedade de símbolos de operadores de atribuição; os dois mais populares são os estilo Fortran = e os estilo Algol :=.

A semântica de uma atribuição é simples. Na ausência de erros, a expressão é avaliada para um valor, que é então *copiado* para o destino, ou seja, a maioria das linguagens imperativas usa *semântica de cópia*.

Expressões são escritas com o uso de operadores aritméticos e lógicos familiares, assim como as chamadas ocasionais a funções-padrão fornecidas pela linguagem. Conforme a Tabela 2.4 sugere, uma rica variedade de operadores está disponível para uso em

Tabela 12.1 Algumas Funções Matemáticas Padrão

Função	Fortran	C/C++	Ada
Seno de x	`sin(x)`	`sin(x)`	`Sin(x)`
Cosseno de x	`cos(x)`	`cos(x)`	`Cos(x)`
Tangente de x	`tan(x)`	`tan(x)`	`Tan(x)`
Arco seno de x	`asin(x)`	`asin(x)`	`Arcsin(x)`
Arco cosseno de x	`acos(x)`	`acos(x)`	`Arccos(x)`
Arco tangente de x	`atan(x)`	`atan(x)`	`Arctan(x)`
Seno hiperbólico de x	`sinh(x)`	`sinh(x)`	`Sinh(x)`
Cosseno hiperbólico de x	`cosh(x)`	`cosh(x)`	`Cosh(x)`
Tangente hiperbólica de x	`tanh(x)`	`tanh(x)`	`Tanh(x)`
Exponencial: e^x	`exp(x)`	`exp(x)`	`Exp(x)`
Logaritmo natural: $ln(x)$	`log(x)`	`log(x)`	`Log(x)`
Logaritmo: $\log_{10}(x)$	`log10(x)`	`log10(x)`	`Log(x,10)`
x^y	`x**y`	`pow(x,y)`	`x**y`
$\sqrt{x}$	`sqrt(x)`	`sqrt(x)`	`Sqrt(x)`
$\lceil x \rceil$	`ceiling(x)`	`ceil(x)`	`Float'Ceiling(x)`
$\lfloor x \rfloor$	`floor(x)`	`floor(x)`	`Float'Floor(x)`
$\lvert x \rvert$	`abs(x)`	`fabs(x)`	`abs(x)`

linguagens imperativas como C e C++. Além disso, linguagens imperativas fornecem bibliotecas de "funções-padrão", que podem executar diversos cálculos durante a avaliação de uma expressão. Um resumo de algumas funções matemáticas de Fortran, C/C++ e Ada é apresentado na Tabela 12.1.

A lista da Tabela 12.1 fornece apenas uma amostra de todas as funções disponíveis. Por exemplo, Fortran possui muitas assim chamadas "funções intrínsecas", além daquelas listadas na Tabela 12.1. As funções-padrão de C, C++ e Ada estão organizadas em bibliotecas predefinidas. Discutimos as bibliotecas padrão C++ adiante, neste capítulo. As bibliotecas padrão C e Ada são identificadas na Tabela 12.2.

As bibliotecas IMSL (**www.vni.com/products/imsl/**) fornecem milhares de funções matemáticas e estatísticas para programadores Fortran e C, além daquelas fornecidas nas suas respectivas bibliotecas-padrão. Essas funções executam uma ampla gama de cálculos em áreas como equações diferenciais, transformações Fourier rápidas (FFTs), correlação, regressão, análise de séries temporais, otimização e muitas outras.

Tabela 12.2 Algumas das Muitas Bibliotecas de Funções em C e Ada

Biblioteca C	Unidade de Biblioteca Ada	Funções Fornecidas
math.h	Ada.Numerics	Funções matemáticas (Tabela 12.1)
ctype.h	Ada.Characters	Funções de classificação de caracteres
string.h	Ada.Strings	Funções de manipulação de strings
stdlib.h	Ada.Numerics	Funções utilitárias
time.h	Ada.Calendar	Funções de data e horário

A Tabela 2.4 revela muito sobre a evolução das linguagens de programação imperativas no decorrer das últimas quatro décadas. Os operadores Fortran são um subconjunto dos operadores C, que, por sua vez, são um subconjunto dos operadores C++, refletindo a evolução dessas três linguagens à medida que a faixa de aplicações de programação se expandiu nas últimas três décadas.

Entretanto, também é importante examinar bibliotecas dependentes de linguagens que facilitam a organização de dados flexíveis em tabelas hash, grafos, filas, pilhas, matrizes e outras formas úteis.

12.4 SUPORTE DE BIBLIOTECA PARA ESTRUTURAS DE DADOS

As estruturas de dados básicas em linguagens imperativas – matrizes e estruturas de registros – foram discutidas no Capítulo 5. Além dessas, linguagens modernas têm bibliotecas extensíveis de funções que facilitam o desenvolvimento de aplicações complexas. Na prática, os programadores se baseiam nessas bibliotecas para evitar "reinventar a roda" cada vez que uma função ou uma estrutura de dados comum é necessária para uma aplicação.

A versão-padrão ANSI/ISO corrente de C++ inclui a "Biblioteca C++ Padrão", uma ampla coleção de classes e funções projetadas para suportar o gerenciamento de estruturas de dados complexas, as funções de E/S, exceções e assim por diante. Essa biblioteca possui as seguintes partes principais:

1. Uma coleção de funções para a definição e manipulação de estruturas de dados
2. Uma coleção de funções de E/S
3. Uma classe de *strings*
4. Uma classe de números complexos
5. Um *framework* para ajustar o programa ao ambiente de execução, como os detalhes de implementação para cada tipo de dado elementar em uma determinada arquitetura
6. Funções de alocação e desalocação de memória
7. Funções de manipulação de exceções
8. Uma classe otimizada para aritmética de matrizes

Antes que o padrão C++ corrente fosse adotado, no final da década de 1990, muitos desses itens eram conhecidos como a *Standard Template Library*, ou simplesmente *STL*. Essa parte da Biblioteca C++ Padrão contém estruturas de dados e funções que manipulam essas estruturas de dados. Nesse sentido, a biblioteca é projetada para a programação imperativa em vez daquela orientada a objetos, como seria característica da biblioteca de classes de Java ou Eiffel, por exemplo. Assim, programadores que prefiram um estilo imperativo em relação a um estilo orientado a objetos podem facilmente explorar a funcionalidade fornecida pela Biblioteca C++ Padrão.

A lista a seguir resume os principais elementos fornecidos pela Biblioteca C++ Padrão:

- Iteradores
- Vetores
- Listas
- Pilhas, filas, deques e filas com prioridade
- Conjuntos e multiconjuntos
- Mapas

- Grafos
- *Strings*
- Números complexos
- Sobrecarga
- Genéricos

A lista a seguir resume algumas das principais funções que podem ser usadas para manipular essas estruturas de dados:

- Indexar um vetor
- Redimensionar vetores e listas
- Inserir e remover elementos de vetores, listas e mapas
- Pesquisa em vetores, listas, mapas, grafos e *strings*
- Ordenação de vetores e listas
- Inserção e exclusão em listas, conjuntos e mapas
- Operações de filas e pilhas (empilhar, desempilhar etc.)
- Funções de grafos (caminhos mais curtos etc.)
- Funções de *strings* (“substrings”, inserção etc.).
- Funções aritméticas, de comparação e E/S para números complexos

Para obter mais detalhes, uma referência excelente sobre a Biblioteca C++ Padrão pode ser encontrada em Josuttis, 1999.

12.5 PROGRAMAÇÃO IMPERATIVA E C

Para que se entenda o projeto de C, ajuda compreender as circunstâncias que levaram ao seu desenvolvimento. De acordo com Kernighan e Ritchie (1978):

> C foi projetado originalmente para ser implementado no sistema operacional UNIX sobre DEC PDP-11, por Dennis Ritchie. O sistema operacional, o compilador C e, essencialmente, todos os programas de aplicação UNIX (inclusive todo o software usado para preparar este livro) são escritos em C. Compiladores de produção também existem para diversas outras máquinas, inclusive o IBM System/370, o Honeywell 6000 e o Interdata 8/32. C não está associado a algum sistema ou hardware específico, entretanto, é fácil escrever programas que sejam executados sem alteração em qualquer máquina que suporte C.

Naquela época, Ken Thompson e Dennis Ritchie se envolveram com os Laboratórios Bell no desenvolvimento de um avançado sistema operacional chamado Multics, que, diferentemente dos sistemas operacionais anteriores, estava sendo escrito amplamente em PL/I em vez de em código assembly. O Bell Labs desistiu de ser um dos parceiros no projeto Multics e vetou trabalhos posteriores nele. Thompson e Ritchie propuseram então o desenvolvimento de um sistema de documentação que seria independente de máquina e seria executado em minicomputadores baratos do final da década de 1960. Assim nasceu o Unix, que era um trocadilho com a palavra Multics.

Um minicomputador de 16 bits típico como o PDP-11 poderia ter 32 KB de memória, dos quais o sistema operacional usava a metade e uma aplicação como um compilador C usava a outra metade. C era baseado inicialmente em uma linguagem chamada BCPL, que não

possuía tipo. Entretanto, os problemas de portabilidade ao se lidar com máquinas *big-endians* e *little-endians* necessitavam de algum sistema de tipos mínimo. Todavia, a falta de um tipo de dado Boleano, de aritmética de ponteiros (por exemplo, tratar os ponteiros como se fossem inteiros) e assim por diante são vestígios do projeto original sem tipos.

A outra característica de projeto importante foi que virtualmente todo o Unix e seus utilitários foram escritos em C. Isso significava que o compilador C tinha de gerar código muito bom, mas tinha uma memória mínima na qual otimizava o código gerado. A solução foi projetar uma linguagem de nível relativamente baixo, na qual foram acrescentados recursos de linguagem como os operadores `++` e `--`, que eram suportados diretamente pelo hardware do PDP-11. Assim, C conseguiu atingir seu objetivo de suportar a codificação do sistema operacional Unix.

Correntemente, a popularidade de C parece ter perdido um pouco de terreno em favor de linguagens como C++, Java, C#, Perl e Python. Todavia, C ainda tem sucesso em duas áreas. C é muitas vezes tratado como uma linguagem de máquina universal; a maioria do núcleo do Unix, por exemplo, é escrita em C. Além disso, o primeiro compilador C++, chamado de `cfront`, traduzia C++ para C, em vez de fazê-lo para código de máquina. Outro uso de C continua a ser feito no desenvolvimento de software para ambientes limitados em memória ou potência, como telefones celulares.

C teve um impacto enorme sobre o projeto de linguagens. Por exemplo, as linguagens C++, Java e C# se parecem muito com C, tanto sintaticamente quanto semanticamente. A maioria dos seus operadores é copiada diretamente de C. Até mesmo Perl usa comandos cuja sintaxe é uma variante de C.

12.5.1 Características Gerais

A sintaxe de C foi examinada nos Capítulos 2 e 3. Seus operadores e sua precedência foram apresentados na Tabela 2.4. C introduziu a operação de conversão, cuja sintaxe é um nome de tipo entre parênteses precedendo uma expressão que gera uma conversão do valor da expressão para aquele tipo. C também introduziu o uso de chaves no lugar do `begin` e `end` de Algol.

C possui comandos de atribuição; comandos de seqüência; comandos condicionais *if* e *switch*; laços *while*, *for* e *do*; além de chamadas a funções. Na área de estruturas de dados, C possui matrizes, ponteiros, estruturas (registros) e tipos de dados de união. Para continuar sendo uma linguagem de baixo nível, possui um dispositivo macro de tempo de compilação e compilação condicional.

Como uma linguagem imperativa, C não possui:

- Iteradores
- Manipulação de exceções
- Sobrecarga
- Genéricos

Algumas pessoas têm argumentado que diversos operadores de C, como o *ou inclusivo de bits* e 11 variações diferentes do operador de atribuição, têm valor questionável na programação prática. Ainda assim, um dos objetivos do projeto de C era facilitar a programação em nível de sistema operacional, no qual o uso estratégico de operações de bits que economizem nanossegundos de tempo de execução é um exercício valioso.

Em C, a atribuição é um operador, o que permite que uma atribuição ocorra em qualquer contexto no qual uma expressão possa ocorrer. Isso é bem diferente de Pascal, Ada, Algol e Fortran, em que a atribuição é um tipo de comando separado e nada mais. A implicação de tratar a atribuição como um operador é que atribuições intermediárias

podem ser inseridas dentro de uma expressão enquanto seu valor estiver sendo avaliado. Analise a seguinte implementação da função `strcpy` de C (que também aparece na Figura 5.6):

```
void strcpy (char *p, char *q) {
     while (*p++ = *q++) ;
}
```

De fato, três comandos aparecem no teste do laço *while*: a cópia de um caractere, a atribuição de pós-incremento de `p` e a atribuição de pós-incremento de `q`. O laço é muito eficiente, terminando quando um caractere `NUL` é copiado. Esse idioma é razoavelmente padrão, mas também é cheio de riscos.

Essa função revela muitos dos problemas de se programar em C. Não é feita nenhuma verificação se o espaço de memória referenciado por `p` não sofrer *overflow*. Tais problemas em C são responsáveis por muitos dos problemas de segurança de *overflow de buffer* na Internet. Além disso, o próprio código é muito crítico.

C suporta a alocação dinâmica de matrizes por intermédio da função `malloc`. Considere o caso em que um programa tenha determinado que precisa de uma matriz de inteiros de posições `size`. Isso seria executado conforme descrito na primeira edição do texto de C de Kernighan e Ritchie (que chamamos de K&R C) por meio de:

```
int *a;
...
a = malloc(sizeof(int) *size);
/* ANSI C: a = (int *) malloc(sizeof(int) *size);
   C++: a = new int[size]; */
```

Suponha que o tipo de `a` mude para algo maior do que um `int`. Devemos alterar não apenas o comando, mas também a referência ao tipo de `a` na chamada a `sizeof`. K&R C não informa um erro em tal caso. Em contraste, ANSI C (a primeira linha de comentário) detecta o erro de conversão, mas, se ele for consertado, não captura o erro de `sizeof`. C++ (a segunda linha de comentário) requer que o tipo no `new` satisfaça ao tipo declarado. Infelizmente, a maior parte de Unix é escrita em K&R C, enquanto alguns utilitários são escritos em dialeto pré-K&R C e outros são escritos em ANSI C.

Todas as três variantes de C requerem que o programador desaloque explicitamente a memória quando esta não for mais necessária. Isso foi discutido no Capítulo 11. Uma falha sutil ocorre quando o programa continua a usar armazenamento alocado para a variável `a` após ele ter sido desalocado.

As Seções 12.5.2 e 12.5.3 apresentam alguns exemplos simples de programação imperativa. O desenvolvimento de um tipo pilha em C é apresentado no Capítulo 13.

12.5.2 Exemplo: Grep

Nesta seção desenvolvemos uma versão C do utilitário Unix chamado `grep`, que é usado para pesquisar *strings* dentro de um arquivo de texto. O utilitário `grep` real possui uma miríade de opções e permite o uso de expressões regulares (veja o Capítulo 3) para pesquisa de texto em linha de comando.

Nossa versão simples, apresentada na Figura 12.2, recebe dois argumentos de linha de comando. O primeiro argumento é uma *string* que deve ser correspondida exatamente. O segundo é o nome do arquivo que deve ser pesquisado.

Nesse exemplo, cada linha que corresponde é impressa no console, precedida pelo seu número de linha. Se nenhuma linha corresponder, não há saída impressa.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SIZE 1024
FILE *in;
void find(char* string);

int main(int argc, char *argv[ ]) {
    char *string;

    if (argc < 3) {
        fprintf(stderr, "Usage: %s string file\n", *argv);
        exit(1);
    }

    string = *++argv;

    if ((in = fopen(*++argv, "r")) == NULL) {
        fprintf(stderr, "Cannot open: %s\n", *argv);
        exit(1);
    }
    find(string);
    return 0;
}

void find(char *string) {
    char line[SIZE];
    int count;
    count = 0;
    while (fgets(line, SIZE, in)) {
        count++;
        if (strstr(line, string))
            printf("%5d:\t%s\n", count, line);
    }
}
```

| Figura 12.2 Uma Versão Simples de Grep

As três primeiras linhas do programa importam arquivos de cabeçalho para três bibliotecas diferentes. Arquivos de cabeçalho contêm assinaturas de tipo para diversas funções. A linha 6 é um exemplo de uma assinatura de função.

> **Definição:** Uma *assinatura de função* é uma declaração do nome da função junto ao seu tipo de retorno e aos tipos e à ordem dos seus parâmetros. Às vezes, isso é chamado de *protótipo da função*.

O próprio programa consiste em duas funções. A função `main` é responsável pelo processamento dos argumentos da linha de comando. Erros são relatados se a linha de comando contiver menos do que dois argumentos ou se o arquivo de entrada informado na linha de comando não puder ser aberto. Em ambos os casos o programa pára.

A função `find` é responsável pela leitura do arquivo, pela pesquisa em cada linha por uma ocorrência da `string` e pela gravação da linha, se a *string* tiver sido encontrada. Aqui, a chamada `strstr(line, string)` retorna um ponteiro para a primeira ocorrência da `string` dentro da `line`. Observe também que as linhas são restritas a 1.023 caracteres, uma a menos do que o tamanho declarado para permitir um byte `NUL` no final. Felizmente, um dos argumentos de `fgets` é o tamanho do *buffer* de linha, evitando assim um *overflow* do *buffer*.

A contrapartida de `fgets` para a leitura da entrada-padrão (o console) não tem tal argumento. Seu uso, muitas vezes, leva a *overflows* no *buffer*. Felizmente, `fgets` pode ser usada com um terceiro argumento de `stdin` para ler do console. Um dos problemas é que até mesmo ANSI C não fornece uma função-padrão para determinar o comprimento de uma linha de entrada, embora seja razoavelmente fácil de se escrever um.

12.5.3 Exemplo: Média

Neste exemplo, lemos uma seqüência de números e calculamos seu mínimo, seu máximo e sua média. Esse é um programa de exemplo típico para explicar o uso de um laço `while` em um curso introdutório de programação.

O programa, apresentado na Figura 12.3, consiste basicamente de um laço que lê números um de cada vez e então conta o número, soma-o e verifica se o número é um novo mínimo ou máximo. Os números são lidos do console (`stdin`).

```
#include <stdio.h>
int main(int argc, char *argv[ ]) {
    int ct, number, min, max, sum;
    sum = ct = 0;
    printf("Enter number: ");
    while (scanf("%d", &number) != EOF) {
        if (ct == 0)
            min = max = number;
        ct++;
        sum += number;
        min = number < min ? number : min;
        max = number > max ? number : max;
        printf("Enter number: ");
    }
    printf("d% numbers read\n", ct);
    if (ct > 0) {
        printf("Average:\t%d\n", sum / ct);
        printf("Maximum:\t%d\n", max);
        printf("minimum:\t%d\n", min);
    }
}
```

| Figura 12.3 Programa para Calcular a Média

O programa usa uma declaração de atribuição múltipla para inicializar `sum` e `ct` e, posteriormente, `min` e `max`. Atribuições condicionais são usadas para atualizar `min` e `max`. A entrada e a saída que usam códigos de formatos também são utilizadas.

Um potencial problema ocorre na chamada para `scanf`. O segundo argumento deve ser o endereço de uma variável `int`. Já que C só suporta chamadas por valor, é responsabilidade do programador fornecer o endereço usando o operador de endereço (`&`). Não fazê-lo em máquinas mais antigas geralmente levava a erros sutis em tempo de execução. Na maioria dos computadores mais modernos a falha é capturada em tempo de compilação (pelo compilador gcc) ou gera uma violação de segmentação (violação na proteção de memória) em tempo de execução. Essa situação expõe uma fraqueza do sistema de tipos de C.

12.5.4 Exemplo: Diferenciação Simbólica

Este exemplo implementa diferenciação simbólica de fórmulas de cálculos simples. Algumas regras familiares para diferenciação simbólica são apresentadas na Figura 12.4.

Por exemplo, diferenciar a função $2 \cdot x + 1$ com respeito a `x` usando essas regras fornece:

$$\begin{aligned}\frac{d(2 \cdot x + 1)}{dx} &= \frac{d(2 \cdot x)}{dx} + \frac{d1}{dx} \\ &= 2 \cdot \frac{dx}{dx} + x \cdot \frac{d2}{dx} + 0 \\ &= 2 \cdot 1 + x \cdot 0 + 0\end{aligned}$$

que comumente simplificaria para 2.

Em uma aplicação como essa é conveniente projetar a representação de expressões usando sintaxe abstrata. A sintaxe abstrata de programas em Clite foi discutida na Seção 2.5.3. Aqui a sintaxe abstrata necessária, resumida na Figura 12.5, é adaptada da sintaxe abstrata de uma expressão em Clite. As diferenças principais de Clite são que existe apenas um único tipo de valor, e as classes de sintaxe abstrata `Binary`, `Variable` e `Value` são tipos de uma `Expression`.

Em uma linguagem imperativa, uma classe na sintaxe abstrata não pode ser mapeada para uma classe na implementação. Em vez disso, é usado um tipo lógico ou físico de união. Já que C não possui uniões com rótulos, a implementação aqui (veja a Figura 12.6) usa uma união lógica, que combina os três tipos de expressões em um registro, junto a um campo de rótulo (chamado `kind`).

Figura 12.4 Regras de Diferenciação Simbólica

$\frac{d}{dx}(c) = 0$	c é uma constante
$\frac{d}{dx}(x) = 1$	
$\frac{d}{dx}(u + v) = \frac{du}{dx} + \frac{dv}{dx}$	u e v são funções de x
$\frac{d}{dx}(u - v) = \frac{du}{dx} - \frac{dv}{dx}$	
$\frac{d}{dx}(uv) = u\frac{dv}{dx} + v\frac{du}{dx}$	
$\frac{d}{dx}\left(\frac{u}{v}\right) = \left(v\frac{du}{dx} - u\frac{dv}{dx}\right)\Big/ v^2$	

Figura 12.5 Sintaxe Abstrata de *Expressões*

Expressão = *Variável* | *Valor* | *Binário*
Variável = *char* id
Valor = *int* valor
Binário = *char* op; *Expressão* esquerda, direita

Como em uma linguagem orientada a objetos, uma função é necessária para criar cada um dos três tipos de nós. Já que C não possui nem construtores de objetos nem funções sobrecarregadas pelo usuário, cada nome de função deve ser único e retorna um ponteiro para o nó construído (veja a Figura 12.7).

Outras duas funções são definidas (veja a Figura 12.8). A função `diff` recebe dois argumentos: a variável na diferenciação e um ponteiro para uma árvore de expressão. Ela retorna uma árvore de expressão representando a expressão diferenciada (mas não simplificada). A função `dump` imprime uma árvore de expressão em um formato de prefixo polonês.

Observe o uso de comandos `switch` tanto na função `diff` quanto na `dump`. Essa é uma das marcas registradas da programação imperativa, a saber, comandos *switch* distribuídos, que a programação orientada a objetos põe de lado. Dos quatro operadores aritméticos, apenas + e * são apresentados; os outros são deixados como exercícios.

A primeira regra na Figura 12.4 diz que a diferenciação de uma constante com respeito a x é 0. Isso é implementado pelo caso `value` na função `diff`, que retorna um zero.

De forma semelhante, a regra de adição da Figura 12.4 (regra três) diz que, para diferenciar uma adição, você deve primeiro diferenciar cada termo e depois adicionar os resultados. O caso em que `op` é um + na função `diff` faz precisamente isso, construindo uma nova expressão (árvore de sintaxe abstrata) para guardar o resultado.

Uma aplicação do método `diff` para a expressão $2 \cdot x + 1$ resulta:
`+ + * 2 1 * x 0 0` representando a expressão $2 \cdot 1 + x \cdot 0 + 0$. Um formatador melhor, junto a um simplificador algébrico, são deixados como exercícios.

12.6 PROGRAMAÇÃO IMPERATIVA E ADA

Ada foi desenvolvida no final da década de 1970 pelo Departamento de Defesa norte-americano para grandes sistemas de comando e controle e para sistemas embarcados em tempo real.

```
enum nodekind {binary, var, value};

struct node {
     enum nodekind kind;
     char op;
     struct node *term1, *term2;
     char id;
     int val;
};
```

| Figura 12.6 Arquivo de Cabeçalho de Diferenciação Simbólica

```
#include <stdio.h>
#include <stdlib.h>
#include "node.h"

struct node *mknodebin(char op1, struct node *left,
                       struct node * right) {
  struct node *result;
  result = (struct node*) malloc(sizeof(struct node));
  result->kind = binary;
  result->op = op1;
  result->term1 = left;
  result->term2 = right;
  return result;
}

struct node *mknodevar(char v) {
  struct node *result;
  result = (struct node*) malloc(sizeof(struct node));
  result->kind = var;
  result->id = v;
  return result;
}

struct node *mknodeval(int v) {
  struct node *result;
  result = (struct node*) malloc(sizeof(struct node));
  result->kind = value;
  result->val = v;
  return result;
}
```

| Figura 12.7 Funções de Construção de Nós

De acordo com um estudo, o Departamento de Defesa estava gastando bilhões de dólares em software, um valor que tinha projeção de aumentar significativamente. E, o que era pior, o estudo descobriu que mais de 450 linguagens estavam em uso, muitas das quais altamente especializadas e idiossincráticas.

A solução foi óbvia: padronizar em uma linguagem. Mas qual delas? Assim nasceu o *Higher Order Language Working Group*, cuja tarefa era identificar uma linguagem apropriada de modo geral para os requisitos do Departamento de Defesa. Isso acabou levando ao desenvolvimento e à padronização, em 1983, da linguagem de programação Ada. O padrão Ada 95 acrescentou extensões orientadas a objetos a Ada 83.

O tempo do desenvolvimento de Ada foi infeliz por dois motivos. Ada 83 era uma linguagem muito ampla. Naquele tempo, um compilador Pascal típico escrito em Pascal variava

```
struct node *diff(char x, struct node *root) {
    struct node *result;
    switch (root->kind) {
    case value: result = mknodeval(0);
        break;
    case var:
        result = mknodeval(root->id == x?1:0);
        break;
    case binary:
        switch (root->op) {
        case '+':
            result = mknodebin(plus,
                        diff(x, root->term1),
                        diff(x, root->term2));
            break;
        case '*';
            result = mknodebin(plus,
                        mknodebin(times, root->term1,
                                diff(x, root->term2)),
                        mknodebin(times, root->term2,
                                diff(x, root->term1)));
            break;
        }
    }
    return result;
}

void dump(struct node *root) {
    if (! root) return;
    switch (root->kind) {
    case value: printf("%d", root->val);
        break;
    case var: printf("%c", root->id);
        break;
    case binary: printf("%c", oper[root->op]);
        dump(root->term1);
        dump(root->term2);
        break;
    }
}
```

| Figura 12.8 Implementação de Diferenciação Simbólica

de 8K a 12K linhas de código, enquanto um compilador Modula típico variava de 15K a 25K linhas. Um compilador Ada 83 tinha no mínimo 250K linhas de código Ada.

Devido ao fato de o estudo da linguagem ter sido feito em meados da década de 1970, o Departamento de Defesa baniu subconjuntos Ada. Entretanto, na década de 1980, os computadores pessoais estavam começando a substituir minicomputadores e mainframes. Computadores pessoais da década de 1980 não tinham memória necessária para hospedar um compilador Ada.

A segunda razão para o tempo infeliz é que linguagens orientadas a objetos estavam para revolucionar o projeto de linguagens. Se o projeto de Ada tivesse iniciado 5 ou 10 anos mais tarde, ela poderia ter sido uma linguagem muito mais simples.

Ada enfrentou dificuldades durante a década de 1990, já que o Departamento de Defesa eliminou o requisito de usar Ada em favor de linguagens e software comercial prontos. Todavia, agora parece haver um ressurgimento do interesse em Ada:

- Softwares prontos provaram ser problemáticos no ambiente de comando e controle.
- O desenvolvimento de Spark Ada e suas ferramentas de tempo de compilação associadas (veja o Capítulo 18) demonstrou convincentemente que software pode ser desenvolvido com alta confiabilidade e custos mais baixos do que software usando linguagens prontas como C/C++.
- O desenvolvimento do compilador NYU GNAT (Ada), agora parte do conjunto de compiladores GNU, disponibilizou um compilador de alta qualidade para educadores e alunos, sem custo.

Nas seções a seguir, usamos o nome Ada para nos referir ao subconjunto imperativo de Ada.

12.6.1 Características Gerais

As linguagens com a maior influência sobre Ada foram Algol e Pascal. Exemplos dessa influência abundam: desde o uso do símbolo := para atribuição até o uso de `begin-end` para blocos. Sintaticamente, as duas maiores diferenças dessas linguagens predecessoras foram o uso de símbolos de término únicos para estruturas compostas e o uso do ponto-e-vírgula como um símbolo de término de comando em vez de um separador de comandos. Essas duas foram concessões ao pragmatismo, já que se descobriu que as outras convenções eram problemáticas.

Mesmo sintaticamente Ada é uma linguagem ampla. A gramática EBNF possui quase 200 produções. Ada não diferencia maiúsculas de minúsculas quanto à escrita de palavras reservadas e identificadores.

Os tipos básicos de dados incluem: caractere, ponto flutuante, ponto fixo, boleano e enumeração. Tipos de dados agregados incluem: matrizes (unidimensionais e multidimensionais), *strings*, registros, registros de tamanho variável e ponteiros. Ada suporta subtipos e tipos derivados.

Diferentemente de C, todos os erros de indexação de matrizes são capturados em Ada. O uso de subfaixas permite que muitas instâncias de erros de indexação sejam capturadas em tempo de compilação.

De forma semelhante, conforme vimos na Seção 5.4.6, embora Ada suporte o tipo de registro de tamanho variável com rótulos (o que geralmente é um furo no sistema de tipos), a sintaxe é dada de forma a permitir que tanto a variante quanto o rótulo sejam atribuídos em uma declaração; qualquer tentativa de atribuição à variante ou ao rótulo separadamente resulta em um erro de tempo de compilação.

Cada referência a uma variante é verificada para assegurar que seja consistente com o rótulo:

```
type union(b : boolean) is
     record
        case b is
           when true =>
                i : integer;
           when false =>
                r : float;
        end case;
     end record;
tagged : union;
begin
     tagged := (b => false, r => 3.375);
     put(tagged.i);
```

Nesse caso, a referência `tagged.i` gera uma exceção em tempo de execução. Uma prática comum em Ada é embutir tais referências em um comando *case* (ou *switch*) no qual acessos dentro do *case* sejam consistentes com o rótulo do *case*.

Os comandos imperativos comuns estão incluídos em Ada: atribuição, if, case (switch), laços (mas não iteradores), blocos, exit, return e goto.

Tanto as funções que retornam valores quanto os procedimentos que não retornam valores são suportados. Contudo, a passagem de parâmetros em Ada é incomum em diversos aspectos. Ada especifica como o parâmetro deve ser usado (entrada, saída, entrada-saída) e deixa para o compilador decidir qual mecanismo usar, por exemplo, valor, referência, resultado e assim por diante. Outro aspecto incomum dos parâmetros é que os parâmetros formais podem especificar valores-padrão se eles forem omitidos. Além disso, uma chamada de função pode usar os nomes dos parâmetros formais e seus valores, em cujo caso a ordem dos parâmetros é irrelevante. Por exemplo:

```
sort(list => student_array, length => n);
```

Esse uso de nomes de parâmetros formais facilita a compreensão dos comandos de chamada.

Ada suporta a manipulação de exceções, as exceções definidas pelo usuário, a sobrecarga e os genéricos.

Uma especificação de uma rotina de ordenação genérica é apresentada na Figura 12.9. Para instanciar essa rotina, o cliente deve fornecer um tipo `element` e uma operação `>` (maior que) para esse tipo. Assim, a rotina de ordenação pode ordenar uma lista arbitrária de tais elementos.

A Figura 12.10 apresenta a implementação da ordenação genérica. Diferentemente de arquivos de cabeçalho C, tanto o cliente quanto a implementação devem referenciar a **mesma** especificação de pacote. As referências `a'first` e `a'last` são instâncias de atributos e denotam os valores inferior e superior da faixa declarada de índices. Assim, o compilador pode garantir em tempo de compilação que nenhum erro de *subscript* possa ocorrer.

Tasking é um recurso interno de Ada. A comunicação entre tarefas é feita por intermédio de um mecanismo chamado *rendezvous* (Hoare, 1978; Hoare, 1985).

```
generic
  type element is private;
  type list is array(natural range <>) of element;
  function ">"(a, b : element) return boolean;
package sort_pck is
  procedure sort (in out a : list);
end sort_pck;
```

| Figura 12.9 Ordenação Geral em Ada

Ada 95 acrescentou novas bibliotecas de funções e suporte à programação orientada a objetos, tornando Ada, dessa forma, uma linguagem multiparadigma. Dado que sua principal área de aplicação é o controle e o comando em tempo real, os programas em Ada precisam ser altamente confiáveis, diferentemente da maioria do software comercial. Os desenvolvedores de uma implementação chamada Spark Ada relatam taxas de erros de 0,04 erro por mil linhas de código (KLOC) contra taxas-padrão de 1–7 erros por KLOC, com uma taxa de produtividade de três vezes a norma da indústria (Croxford, 2005). Em comparação, há aproximadamente 15 anos, a falta de um comando `break` em um programa em C com diversos milhões de linhas causou uma falha em todo o sistema AT&T, resultando em um prejuízo em torno de 1 bilhão de dólares.

12.6.2 Exemplo: Média

Neste exemplo lemos uma seqüência de números e calculamos seu mínimo, máximo e sua média. Esse é um típico programa de exemplo para explicar o uso de um laço `while`. O leitor interessado pode desejar comparar a versão em C do programa da Figura 12.3 com a versão em Ada.

```
package body sort_pck is
procedure sort (in out a : list) is
begin
  for i in a'first .. a'last - 1 loop
    for j in i+1 .. a'last loop
      if a(i) > a(j) then
         declare t : element;
         begin
            t := a(i);
            a(i) := a(j);
            a(j) := t;
         end;
      end if;
    end loop;
  end loop;
end sort;
end sort_pck;
```

| Figura 12.10 Implementação de Ordenação por Seleção Genérica em Ada

O programa, apresentado na Figura 12.11, consiste basicamente em um laço que lê números um de cada vez e então conta o número, soma-o e verifica se o número é um novo mínimo ou máximo. Os números são lidos do console.

O programa é notadamente uma cópia da versão em C com algumas melhorias. Primeiro, o laço é basicamente infinito (já que não há teste), baseando-se no fato de que uma tentativa de leitura que passe do final do arquivo gera uma exceção `End_Error`.

Em segundo lugar, o laço contém um bloco interno que captura as exceções geradas pelo `Get`. Os dois primeiros geram uma mensagem de erro e o laço é repetido, iniciando pela solicitação de outro número. No caso de um final de arquivo ser detectado, o laço é abandonado.

A outra característica de Ada a ser observada, além da sua robustez, é a sua prolixidade em relação a C. Por exemplo, escrever a média, o mínimo e o máximo demanda três comandos cada um em Ada, contra um em C.

12.6.3 Exemplo: Multiplicação de Matrizes

Ada suporta a redefinição de qualquer operador aritmético sempre que seus operandos não forem tipos aritméticos simples. Usamos essa estratégia aqui para sobrecarregar o operador de multiplicação para implementar a multiplicação de matrizes.

A Figura 12.12 mostra uma implementação em Ada da multiplicação de matrizes como uma sobrecarga do operador "*" quando seus operandos forem do tipo `Matrix`. Quando o número de colunas em A não for o mesmo de linhas em B, esse algoritmo gera a exceção `Bounds_Error`.

As expressões `A'First(2)` e `A'Last(2)` denotam os limites inferior e superior da faixa de índices na segunda dimensão da matriz A. A expressão `A'Range(2)` denota a faixa de valores de índice para a segunda dimensão de A.

Lembre-se de que o produto de uma matriz $m \times n$ e de uma matriz $n \times p$ é uma matriz $m \times p$. Assim, o resultado `C` deve ser declarado como tendo `A'Range(1)` para sua primeira dimensão e `B'Range(2)` para sua segunda dimensão.

O comando `if` verifica se a faixa da segunda dimensão de `A` é idêntica à primeira dimensão de `B`. Caso contrário, as matrizes não podem ser multiplicadas. Nesse caso, o programa em Ada gera uma exceção `Bounds_Error`.

Da mesma forma que Pascal, os laços com contador em Ada são restritos a contar crescente ou decrescentemente por um, com o número de vezes em que o laço é executado sendo determinado no momento do início do laço. Assim, quaisquer mudanças nos valores inicial e final da variável contadora dentro do laço não afetam o número de vezes em que o laço é repetido. As variáveis `i` e `j` são declaradas implicitamente pelo cabeçalho do laço `for`; seu escopo é estritamente o do próprio laço `for`. Diferentemente de C, um índice ilegal não pode ocorrer no laço `for`, que pode ser verificado em tempo de compilação.

Finalmente, Ada usa `end`'s explícitos para seus comandos `if` e `for`, assim como a própria função. Isso evita o problema de sintaxe do *else pendente* discutido no Capítulo 2.

12.7 PROGRAMAÇÃO IMPERATIVA E PERL

Perl é uma linguagem de *scripting* amplamente usada, que é uma linguagem de alto nível interpretada em tempo de execução em vez de compilada em linguagem de máquina. As primeiras linguagens de *scripting* eram usadas para automatizar o controle de tarefas que um usuário poderia executar no teclado. Exemplos incluem arquivos de lote MS-DOS, *scripts* de *shell* Unix Bourne e *scripts* HyperCard Apple.

```
with Ada.Text_IO; with Ada.Integer_Text_IO;
procedure Average is
   Ct, Number, Min, Max : Integer;
begin
   Sum := 0;
   Ct := 0;
   Ada.Text_IO.Put("Enter number: ");
   loop
      begin
         Ada.Integer_Text_IO.Get(Number);
         if Ct = 0 then
            Min := Number;
            Max := Number;
         end if;
         Count := Count + 1;
         Sum := Sum + Number;
         if Number < Min then
            Min := Number;
         elsif Number > Max then
            Max := Number;
         end if;
      exception
         when Constraint_Error =>
            Ada.Text_IO.Put("Value out of range. ");
         when Ada.Text_IO.Data_Error =>
            Ada.Text_IO.Put("Value not an integer. ");
         when Ada.Text_IO.End_Error =>
            exit;
      end;
      Ada.Text_IO.Put("Enter number: ");
   end loop;
   Ada.Integer_Text_IO.Put(Ct, 5);
   Ada.Text_IO.Put(" numbers read");
   Ada.Text_IO.New_Line;
   if Ct > 0 then
      Ada.Text_IO.Put("Average: ");
      Ada.Integer_Text_IO.Put(Sum / Ct);
      Ada.Text_IO.New_Line;
      Ada.Text_IO.Put("Maximum: ");
      Ada.Integer_Text_IO.Put(Maximum);
      Ada.Text_IO.New_Line;
      Ada.Text_IO.Put("Minimum: ");
      Ada.Integer_Text_IO.Put(Minimum);
      Ada.Text_IO.New_Line;
   end if;
end Average;
```

| Figura 12.11 Programa em Ada para o Cálculo de Média

```
type Matrix is
   array (Positive range <> of Float,
          Positive range <> of Float);
function "*" (A, B: Matrix) return Matrix is
   C: Matrix (A'Ranged(1), B'Range(2));
   Sum: Float;
begin
   if A'First(2) /= B'First(1) or
      A'Last(2) /= B'Last(1) then
      raise Bounds_Error;
   end if;
   for i in C'Ranged(1) loop
      for j in C'Range(2) loop
          Sum := 0.0;
          for k in A'Range(2) loop
              Sum := Sum + A(i,k) * B(k,j);
          end loop;
          Result(i,j) := Sum;
      end loop;
   end loop;
   return C;
end "*";
```

| Figura 12.12 Sobrecarga do Operador "*" para Multiplicação de Matrizes

Na maioria dos computadores, um programa em Perl é compilado dinamicamente em *byte code*, que é então interpretado. Todavia, existem compiladores Perl que produzem um programa executável (semelhantes aos compiladores C).

De acordo com Schwartz (1993):

> Larry Wall [...] criou Perl quando estava tentando produzir alguns relatórios a partir de uma hierarquia de arquivos do tipo *news* Usenet para um sistema de relatórios de erros, e *awk* começou a dar resultados piores. Larry, sendo o programador preguiçoso que era, decidiu acabar com o problema com uma ferramenta de propósito geral que ele pudesse usar em pelo menos outro lugar. O resultado foi a primeira versão de Perl.

Embora Perl tenha sua raiz como uma linguagem de *scripting* Unix, agora está amplamente disponível para a maioria dos principais sistemas operacionais, incluindo o Linux, Mac OS X e MS Windows.

As linguagens de *scripting* permitem que as aplicações sejam conectadas, no sentido de que *scripts* são amplamente usados para receber dados de saída de uma aplicação e reformatá-los, deixando-os em um formato apropriado para a entrada de uma aplicação diferente. Embora exista uma penalidade de desempenho para realizar o *scripting* de uma aplicação *conectada*, a maior parte do tempo é gasta nas próprias aplicações, comparado com o *script*, e, à medida que os computadores ficam mais rápidos, os *scripts* que conectam as aplicações têm se provado "rápidos o suficiente".

As linguagens geralmente incluídas na categoria de *scripting* incluem o *shell* Bourne Unix para o controle de tarefas, Javascript para páginas web e PHP para aplicações web no lado servidor. Linguagens de *scripting* de propósito geral comparáveis a Perl incluem Python, Ruby e Tcl.

12.7.1 Características Gerais

Da mesma forma que a maioria das linguagens de *scripting*, Perl é tipada dinamicamente e suporta números, tanto inteiros quanto de ponto flutuante, e expressões regulares. As estruturas de dados incluem matrizes dinâmicas (com índices inteiros) e matrizes associativas (com índices de *strings*).

Perl e Python (veja a Seção 13.5) usam abordagens opostas quanto a tipos básicos e estruturas de dados. Perl executa conversões implícitas entre tipos de dados básicos conforme for necessário. O resultado é que Perl possui um operador distinto para a maioria dos operadores. Assim, Perl usa um ponto (com espaços em branco necessários em volta) para a concatenação de *strings*. Dessa forma,

```
"abc" . "def"
```

produz uma *string* "`abcdef`", enquanto:

```
123 . 4.56
```

produz a *string* "`1234.56`"; observe os dois usos do ponto no último exemplo. Como um operador binário, o ponto é interpretado como uma concatenação de *strings* e os operandos são convertidos em *strings*.

A política de ter operadores únicos para *strings* é levada para os relacionais. Os operadores relacionais comuns são reservados para números; os relacionais de *strings* são `eq`, `ne`, `lt`, `le`, `gt` e `ge`. A seguir estão vários exemplos com o resultado mencionado como um comentário (seguindo o símbolo #):

```
10 < 2      # false
10 < "2"    # false
"10" lt "2" # true
10 lt "2"   # true
```

No segundo exemplo, a *string* é convertida no número 2, enquanto no último exemplo o 10 é convertido em uma *string*. A verdade das duas últimas expressões segue do fato de que o caractere 1 é menor do que o caractere 2, de modo que o 0 no 10 é irrelevante; isso é análogo a comparar a *string* `ax` com a *string* `b`.

As variáveis são escalares, as quais devem ser prefixadas com um sinal de cifrão (`$`), matrizes (prefixadas por `@`) ou matrizes associativas (prefixadas por `%`). Uma ocorrência indexada de uma matriz ou uma matriz associativa é geralmente um valor escalar, e é, dessa forma, prefixado com um `$`. Devido a reclamações dentro da comunidade Perl, Perl 6 rompe essa tradição.

Perl é igualmente permissivo com matrizes (que usam indexação com zero como padrão). Se declararmos a matriz:

```
@a = (2, 3, 5, 7);
```

então o tamanho de `a` é 4 e o valor de `a[3]` é 7. Entretanto, se depois executarmos:

```
$a[7] = 17;
```

a atribuição é legal e o tamanho de `a` se torna 8. Todavia, os índices 4, 5 e 6 de `a` têm o valor especial `undef`, que é interpretado como falso (como um Boleano), a *string* vazia (como uma *string*) e 0 (como um número).

O desenvolvedor de Perl, Larry Wall, acha que as três grandes virtudes de um programador são a *preguiça,* a *impaciência* e a *insolência*. Como lingüista, Wall é amplamente a favor de que se permitam muitas formas diferentes de "dizer a mesma coisa". Assim, muito da sintaxe de Perl é opcional, o que dificulta sua compreensão por um iniciante. O erro de sintaxe mais comum de Perl é:

Erro de sintaxe próximo da linha *x*

em que *x* é algum número de linha. Isso lembra a citação de Niklaus Wirth no início do Capítulo 3: se o analisador não conseguir apresentar uma mensagem de erro e localização precisa, talvez a falha seja da sintaxe da linguagem.

Como uma linguagem tipada dinamicamente, Perl perde tanto em sobrecarga quanto em genéricos. Perde também em manipulação de exceções e em iteradores definidos pelo usuário. Perl adicionou suporte a classes na versão 5, tornando-a uma linguagem multiparadigma.

Um dos pontos mais fortes de Perl é o seu suporte a expressões regulares, que são usadas tanto para a correspondência simples de padrões quanto para a substituição de uma *string* por outra. De fato, muitos programas em Perl exploram essa característica.

Todavia, Perl é irregular a esse respeito; expressões regulares não são objetos de primeira classe. Elas geralmente não podem ser atribuídas a variáveis ou passadas como parâmetros. Essa é apenas uma das muitas irregularidades da linguagem. Como o exemplo mostra, Perl é uma linguagem ampla com muitas sintaxes opcionais ou alternativas e muitas irregularidades.

12.7.2 Exemplo: Grep

Nesta seção desenvolvemos uma versão em Perl do utilitário Unix denominado `grep`, que foi desenvolvido em C em uma versão anterior. Essa versão, apresentada na Figura 12.13, recebe dois argumentos de linha de comando, uma *string* a ser encontrada de forma exata e o nome do arquivo a ser pesquisado.

```
#! /usr/bin/perl

die "Usage mygrep string \n" if @ARGV < 1;
use strict;
my $string = shift;
my $ct = 0;

while (<>) {
    $ct++;
    print "$ct:\t$_" if /$string/;
}
exit;
```

| Figura 12.13 Uma Versão Simples de Grep

Nesse exemplo, cada linha que corresponder é impressa no console, precedida pelo seu número de linha. Se nenhuma linha corresponder, não há saída.

Os comentários começam com o símbolo # e continuam até o final da linha. A primeira linha do programa é um comentário especial (o `!` após o símbolo de comentário # é importante) que apresenta a localização do interpretador Perl. Em Unix, se tal arquivo estiver marcado como executável, pode ser chamado diretamente como um comando.

O segundo comando é um `if` para trás que termina o programa se não houver pelo menos um argumento de linha de comando para o comando. Esse argumento é deslocado da frente da matriz `@ARGV` e armazenado na variável escalar `$string`. Referências a matrizes são prefixadas pelo símbolo (@), enquanto referências escalares são prefixadas com o sinal de cifrão ($).

A diretiva `use strict` requer que cada variável seja declarada por meio de uma diretiva `my`. Embora desnecessária em *scripts* pequenos como esse, é indispensável na captura de erros na escrita de nomes de variáveis em *scripts* maiores.

A sintaxe do laço `while` é convencional, exceto pelo fato de Perl requerer o uso de chaves tanto para laços quanto para comandos `if`. Observe os parênteses necessários em torno do teste do laço *while*, da mesma forma que em C.

O operador de ângulo (<>) representa uma linha lida em um manipulador de arquivo de entrada. Se o arquivo estiver no final, um `undef` é retornado, o que é representado como falso, terminando o laço. Caso contrário, a linha lida é retornada, incluindo quaisquer caracteres de final de linha dependente de sistema operacional. Assim, a linha nunca está vazia, e é interpretada como verdadeira.

Mas qual arquivo está sendo lido? E a linha lida é simplesmente jogada fora? A resposta é que o teste é interpretado da seguinte maneira:

```
while ($_ = <>) {
    ...
}
```

De acordo com o "Camel book" (Wall *et al.*, 1996, p. 53), a referência mais oficial a Perl é:

> O operador de entrada mais usado é o de entrada de linha, também conhecido como o operador de ângulo. [...] Normalmente, você atribuiria o valor de entrada a uma variável, mas há uma situação em que ocorre uma atribuição automática. Se, e somente se, o operador de entrada de linha for a única coisa dentro da condição de um laço *while*, o valor é atribuído automaticamente à variável `$_`.

Infelizmente, Perl é permeado por tais regras, tornando-a uma linguagem difícil de se conhecer bem.

O arquivo que foi lido é (Wall *et al.*, 1996, p. 15):

> todos os arquivos na linha de comando, ou `STDIN`, se nenhum for especificado.

Supondo que o comando se chame `mygrep`, qualquer uma das chamadas a seguir funcionaria em Unix

```
mygrep  xxx  aFile
cat aFile | mygrep xxx
mygrep xxx < aFile
```

```
#! /usr/bin/perl

if (@ARGV < 1) {
    die "Usage mygrep string \n" ;
}
use strict;
my $string = shift(@ARGV);
my $ct = 0;
my $line;

while ($line = <>) {
    $ct++;
    if ($line =~m/$string/) {
        print STDOUT $ct, ":\t", $line;
    }
}
exit;
```

| Figura 12.14 Versão Alternativa do Programa grep

para uma *string* arbitrária `xxx`. A primeira chamada de comando é comparável à versão em C. A segunda possui algum comando (ou comandos) gerando entrada que é enviada para `mygrep` por meio de um *pipe* Unix. A terceira usa redirecionamento de entrada para fazer `mygrep` ler o arquivo de `stdin`. Apenas a primeira dessas chamadas funciona para a versão em C da Seção 12.5.2.

Ao se reescrever o teste do laço *while*, observe o sujeito deduzido chamado `$_`. Muitos comandos fazem essa interpretação se nenhum sujeito for fornecido. Por exemplo,

```
print; #interpreted as: print $_;
```

a impressão simples, sem objeto, é interpretada como imprimir o escalar `$_`. De forma semelhante, o comando `if` para trás dentro do laço *while* é interpretado da seguinte forma:

```
print "$ct:t$_" if $_ =~m/$string/;
```

Ou seja, a linha é impressa se a *string* corresponder a parte da linha. O escalar `$_` é o sujeito (=~)[2] de uma correspondência de padrão simples (`m`), na qual o padrão é convencionalmente inserido em símbolos `/`. Assim, alguns caracteres especiais de correspondência de padrões (como o caractere de máscara simples) funcionarão, enquanto outros não.

Observe também que referências escalares podem ser inseridas em uma *string* entre aspas duplas, mas não em uma *string* entre aspas simples.

Uma versão mais convencional do programa grep é apresentada na Figura 12.14. Nessa versão são usados apenas comandos `if` convencionais, e a versão tenta tornar tudo explícito.

2. Muitos programadores acham esse símbolo confuso, já que se parece com um operador de atribuição. A linguagem *awk* usa apenas o til (~).

Ambas as versões têm um erro sutil. Experimente executar o seguinte comando:

```
altgrep string mygrep.c mygrep.c
```

São deixados como exercício a descrição e o conserto desse erro.

Outra irregularidade em Perl é que a variável `$_` possui escopo global. O uso excessivo desse sujeito pode fazer uma sub-rotina alterar acidentalmente seu valor, levando a erros sutis.

Finalmente, o comando `exit` em cada *script* é desnecessário.

12.7.3 Exemplo: Enviando Notas

Um *script* de conexão típico move dados de uma aplicação para outra. Notas de alunos são muitas vezes mantidas em uma planilha. Após cada projeto ou teste, o instrutor quer enviar por e-mail uma cópia das notas e sua média para cada aluno. É claro que, por motivos de segurança, cada aluno deve receber uma cópia apenas das suas próprias notas. Além disso, é útil incluir a média da turma em cada projeto e teste, assim como a média geral da turma.

Ao escrever tal aplicação, é útil saber que diferentes instrutores podem usar aplicações diferentes de planilhas ou versões diferentes da mesma aplicação de planilha. Tais aplicações de planilha geralmente podem exportar seus dados para arquivos comuns de texto em uma forma chamada *valores separados por vírgulas* (CSV, da sigla em inglês). Linhas de planilhas se tornam linhas de texto, com cada coluna com um separador (por padrão, uma vírgula). Já que as vírgulas podem ocorrer naturalmente em alguns campos, escolhas melhores (dependendo da planilha usada) incluem o caractere de tabulação, os dois-pontos e o ponto-e-vírgula. Algumas aplicações de planilhas podem colocar entre aspas simples ou duplas alguns ou todos os valores, as quais precisam ser apagadas. Assim, cada aplicação de planilha pode ter uma definição ligeiramente diferente do formato CSV.

Um exemplo de como esse formato é usado nessa aplicação aparece na Figura 12.15. O separador usado nesse exemplo são dois-pontos, já que a vírgula não pode ser usada (ela aparece dentro do campo nome) e os nomes dos alunos são armazenados como *Último nome, primeiro nome*.

Nessa planilha, as duas primeiras colunas contêm nomes de alunos e endereços de e-mail, enquanto as duas últimas contêm os pontos totais e as médias. As colunas intermediárias contêm as próprias notas; algumas colunas estão vazias porque os projetos ou os testes ainda não foram atribuídos. A primeira linha contém o nome do projeto ou do teste, enquanto a segunda contém o número total de pontos atribuídos ao projeto. A última linha contém a coluna das médias. As linhas intermediárias contêm o número de alunos individuais; no exemplo da Figura 12.15, há dois alunos. Valores calculados podem aparecer como inteiros ou ponto flutuante; esses valores podem conter um número excessivo de dígitos decimais.

Figura 12.15
Notas de Alunos em Formato CSV

```
::Proj1:Test1:::::Total:Average
::50:100::::::150:
Tucker:atuck@college.edu:48:97:::::145:96.66666666
Noonan:rnoon@college.edu:40:85:::::125:83.33333333
Average::88:91:::::135:90
```

```
#! /usr/bin/perl

use strict;
my $class = shift;
my $suf = ".csv";
open(IN, *<$class$suf")  || die "Cannot read: $class$suf\n";
my $sep = ":";
my $tab = 8;
my $q = ' " '

# read header lines: titles, max grades
my @hdr = &readSplit();
my @max = &readSplit();
push(@max, '100%');

# read students
my @student;
while (<IN>) {
    chomp;
    tr /"//d; # "
    push(@student, $_);
}
my @ave = split(/$sep/, pop(@student));

# gen mail for each student
my $ct = 0;
foreach (@student) {
    my @p = split(/$sep/);
    $ct += &sendMail(@p);
}
$ave[1] = $ENV{"USER"};
&sendMail(@ave);
print "Emails sent: $ct\n";
exit;
```

| Figura 12.16 Exemplo: Programa Principal de `mailgrades`

O programa é apresentado em duas figuras. O programa principal aparece na Figura 12.16. Ele está dividido em quatro partes, cada uma separada por uma linha em branco.

A primeira parte recupera a designação da classe da linha de comando e abre o arquivo CSV para entrada. Ela também é responsável por "declarar" algumas constantes globais que estão sujeitas a alterações.

A segunda parte lê e processa as duas primeiras linhas de cabeçalho. Estas são armazenadas como matrizes, um elemento de matriz por coluna.

A terceira parte do programa lê todos os dados dos alunos usando o idioma de laço *while* de Perl para ler e processar linhas de um arquivo. Cada linha é armazenada em uma matriz usando a função `push` para inseri-la na matriz `@student`. É claro que a última linha não é um aluno, mas, sim, a linha das médias das colunas, de modo que é retirada da matriz, dividida em colunas e armazenada na matriz `@ave`. Assim, o arquivo inteiro deve ser lido para enviar um e-mail com as médias das colunas a cada aluno.

A última parte itera a lista de alunos, gerando um e-mail para cada aluno, usando a rotina `sendMail`. O laço `foreach` é como o iterador Java, que processa um valor de matriz por iteração. Já que nenhuma variável explícita de iteração é dada, a variável implícita (e global) `$_` é usada e referenciada implicitamente no operador `split`. Este último usa o separador (os dois-pontos) para dividir a linha em uma matriz, uma coluna por elemento da matriz. A trilha após o laço `foreach` envia o e-mail com as médias das colunas para a pessoa executando o *script*.

O resto do programa é apresentado na Figura 12.17 e consiste em duas rotinas auxiliares. Observe que nenhuma rotina declara parâmetros formais; a rotina `readSplit` não possui nenhum, mas a `sendMail` possui dois. Na ausência de comentários, é necessário que se leia o código para descobrir esses fatos.

A rotina `readSplit` consiste em cinco linhas de código. O primeiro comando lê a linha do arquivo de entrada e o segundo apaga o caractere ou os caracteres de final de arquivo, que é dependente do sistema operacional. A variável `$_` deve ser explícita no primeiro comando, mas é opcional no segundo.

Outra irregularidade de Perl é demonstrada pela reescrita do segundo comando (com ou sem parênteses após o `chomp`) como:

```
$_ = chomp $_;
```

O operador `chomp` modifica seu argumento e retorna os caracteres apagados como o valor da função, de modo que essa versão modificada é totalmente incorreta.

O terceiro comando em `readSplit` usa o operador `tr` para apagar todas as aspas duplas da linha. Já que nenhum objeto ou assunto foi declarado, supõe-se que a variável `$_` é assumida.

As duas últimas linhas da rotina trazem outra característica, que às vezes é confusa. Muitos operadores podem ser avaliados em contexto de lista ou escalar. O operador `split` varre uma *string* procurando delimitadores conforme apresentado pelo padrão (nesse caso, dois-pontos) e divide a *string* em uma matriz de "substrings". No contexto de lista essa matriz de substrings é o resultado, enquanto no escalar o tamanho da matriz de substrings é o resultado. Como exercício, experimente diversas modificações nessas duas últimas linhas.

Os detalhes da geração de uma mensagem de e-mail para o aluno estão escondidos dentro da rotina `sendMail`. Ela retorna 1 se a mensagem for enviada e 0, caso contrário. Os primeiros dois comandos deslocam os argumentos para a rotina desde a matriz `@_`; isso fornece efetivamente uma chamada por valor. Uma alternativa é usar os nomes `$_[0]` para `$name` e `$[1]` para `$email`; isso fornece efetivamente uma chamada por referência. Para essa rotina, chamada por valor é suficiente e significativamente mais legível. Aproximadamente 40% dos comandos nesse programa estão nessa rotina, a maioria dos quais está associada à geração e à formatação da mensagem de e-mail.

O terceiro comando retorna se o campo do endereço de e-mail estiver vazio. Esse mecanismo suporta a manutenção do aluno na planilha mesmo se ele sair do curso.

```
sub readSplit {
    $_ = <IN>;
    chomp;
    tr /$q//d;
    my @r = split(/$sep/);
    return @r;
}

sub sendMail {
    my $name = shift;
    my $email = shift;
    return 0 unless $email;
    open(MAIL, "| mail -s '$class Grades' $email")
        || die "Cannot fork mail: $!\n";
    print MAIL "GRADE\t\tYOUR\tMAX\tCLASS\n",
               "NAME\t\tSCORE\tSCORE\tAVE\n\n";

    my $ct = 1:
    foreach (@_) {
        $ct++;
        next unless $hdr[$ct];
        print MAIL "$hdr[$ct]\t";
        print MAIL "\t" if length($hdr[$ct]) < $tab;
        if (/^\d/) { print MAIL int($_ + 0.5); }
        else { print MAIL $_; }
        print MAIL "\t$max[$ct]\t";
        if ($ave[$ct] =~/^\d/) {
            print MAIL int($ave[$ct] + 0.5);
        } else { print MAIL $ave[$ct];}
        print MAIL "\n";
    }
    return 1;
}
```

| Figura 12.17 Rotinas de Apoio para Enviar Notas

O próximo comando abre um arquivo de saída como um *pipe* para o comando de mail Unix/Linux (Berkeley). Quando o arquivo é fechado, o que foi escrito nele se torna o corpo da mensagem de e-mail. Uma implementação melhor usaria um módulo Perl para interfacear com um servidor de pop mail. Após imprimir alguns cabeçalhos de colunas, o laço `foreach` a seguir processa as colunas restantes, formatando a mensagem de mail.

Isso conclui nosso breve exame de Perl. Apesar da nossa crítica ao projeto da linguagem, os autores escreveram cinco programas "rápidos e simples" para suportar a escrita deste livro; todos os cinco foram escritos em Perl.

12.8 RESUMO

A programação imperativa é o mais antigo e mais bem estabelecido dos paradigmas de linguagens, imitando o projeto dos primeiros computadores. Muitas das características-chave das linguagens imperativas foram discutidas em capítulos anteriores.

Neste capítulo, deslocamos nossa atenção para um exame do desenvolvimento de programas no paradigma imperativo. A seguir, analisamos o comando de atribuição e a importância de funções e operadores internos.

Finalmente, examinamos brevemente as linguagens C, Ada e Perl. C, a mais antiga destas, é fracamente tipada, enquanto Ada é fortemente tipada. Perl é uma linguagem de *scripting* que é muito útil para aplicações "rápidas e simples", apesar das suas muitas idiossincrasias.

EXERCÍCIOS

12.1 Discuta as vantagens e as desvantagens de se ter identificadores que diferenciem letras maiúsculas e minúsculas em uma linguagem de programação, no que diz respeito à confiabilidade do programa, à verificação de tipos e à complexidade em tempo de compilação.

12.2 Começando na Tabela 2.4, mostre os operadores e suas precedências em C, Ada e Perl.

12.3 Prepare uma tabela que contenha as bibliotecas de *strings* padrão de C, Ada e Perl. Ignore expressões regulares em Perl.

12.4 Escreva um artigo que compare e diferencie matrizes (com subscripts inteiros) em C, Ada e Perl.

12.5 Usando programas para os quais código-fonte está disponível, discuta, por exemplo, as principais diferenças entre C pré-K&R, a primeira edição de C K&R e ANSI C.

12.6 Usando programas para os quais o código-fonte esteja disponível, discuta, por exemplo, como os programas utilitários de *strings* como `grep` evitam o problema do *overflow* do *buffer* em C.

12.7 Que outras funções-padrão ANSI C requerem o endereço de uma variável como `scanf`?

12.8 Para a linguagem C, use uma rotina de ordenação simples para demonstrar que é possível fazer um cliente e uma implementação usarem arquivos de cabeçalho distintos com conseqüências desastrosas.

12.9 Escreva o programa das oito rainhas da Seção 13.4.2 em C.

12.10 Compile e execute o programa `average` em Ada. O que acontece se você digitar valores não numéricos, valores inteiros maiores do que $2^{31} - 1$ e valores cuja soma exceda 2^{32} ?

12.11 Escreva um *test driver* para a rotina de multiplicação de matrizes em Ada.

12.12 Estenda a rotina de multiplicação de matrizes Ada em um pacote de matrizes que suporte, além disso, adição e subtração de matrizes.

12.13 Estenda o pacote de matrizes Ada para suportar a criação de matrizes.

12.14 Reimplemente o programa das oito rainhas da Seção 13.4.2 em Ada.

12.15 Usando o programa `mygrep` em Perl, teste quais caracteres na *string* são interpretados como seqüências de expressões regulares, em vez deles mesmos.

12.16 Demonstre a falha descrita na Seção 12.7.2. A seguir, conserte-a.

12.17 Construa um programa em Perl mostrando o escopo global da variável escalar `$_`.

12.18 Para a linguagem Perl, modifique o programa `mailgrades` para demonstrar, usando a função `split`, a diferença entre o contexto de lista e o escalar.

12.19 Reimplemente o programa das oito rainhas da Seção 13.4.2 em Perl.

13 Programação Orientada a Objetos

"Surpreende-me ver que escritores antigos e modernos não tenham atribuído maior importância às leis da herança..."

Alexis de Tocqueville, *Democracy in America*, 1840

VISÃO GERAL DO CAPÍTULO

Neste capítulo, apresentamos o paradigma da programação orientada a objetos (POO), incluindo sua história e seu impacto, seus problemas de projeto e exemplos de sua utilização. Primeiro, descrevemos a idéia dos tipos de dados abstratos, uma idéia que antecedeu e influenciou amplamente a idéia da programação orientada a objetos. Em seguida, descrevemos o paradigma orientado a objetos como uma extensão da abstração de dados. Essa discussão engloba a história pertinente e as controvérsias em torno do projeto do núcleo que envolvem as linguagens de programação orientada a objetos, como a herança

e o polimorfismo. Depois, apresentamos alguns exemplos interessantes de programação orientada a objetos em Smalltalk, Java e Python.

13.1 PRELÚDIO: TIPOS DE DADOS ABSTRATOS

Um importante desenvolvimento em programação imperativa foi um trabalho que começou com Simula 67 para estender a abstração procedural, conforme descrevemos na Seção 12.2, para incluir *abstração de dados* ou *tipos de dados abstratos*. Existem bons exemplos de abstração de dados entre os tipos de dados primitivos.[1] Por exemplo, o tipo de dado `float` tem as operações aritméticas padrão (soma, subtração etc.) e as operações relacionais (menor que, maior que etc.) encapsuladas com ele. Ele é abstrato porque o uso do tipo `float` nos programas não depende da representação concreta de seus valores ou dos detalhes de como suas operações são computadas.

A abstração de dados estende a noção de tipo, proporcionando ao programador um mecanismo de *encapsulamento* para definir novos tipos de dados que estejam ligados à aplicação que se tem em mãos.

> **Definição**: *Encapsulamento* é um mecanismo que permite que constantes logicamente relacionadas, tipos, variáveis, métodos, entre outros, sejam agrupados em uma nova entidade. Exemplos incluem *procedimentos*, *pacotes* e classes.

Utilizando esse mecanismo, o programa pode limitar o escopo e a visibilidade dos valores de dados e das funções encapsuladas para esse tipo de dado recém-definido. Assim, temos a noção de um *tipo de dado abstrato*.

Considere, por exemplo, a definição de uma pilha. Uma abstração dessa noção deverá encapsular uma representação da pilha, juntamente às suas operações fundamentais: `push`, `pop`, `top` e `empty`. Podem ser desejáveis também outras operações como a habilidade para inicializar ou mostrar o conteúdo da pilha. Considere a implementação para uma pilha de inteiros na linguagem C dada na Figura 13.1.

Em C, um arquivo de cabeçalho separado fornece a especificação da *interface* para a abstração da pilha. Uma interface contém somente a *assinatura* para cada função, que inclui seu nome, seu tipo de retorno e os tipos de cada um de seus parâmetros. A omissão de uma função ou de uma variável de dado global no arquivo de cabeçalho efetivamente oculta os detalhes de sua implementação do uso por outras partes do programa.[2]

O uso de compilação independente para proporcionar um mecanismo de compilação para uma pilha não nos fornece, por si só, um verdadeiro tipo de dado abstrato. A dificuldade resulta do fato de que, em C, não podemos declarar variáveis como sendo do tipo pilha. O código apresentado na Figura 13.1 define somente uma única instância de uma pilha, não um *tipo* pilha. Além disso, o código é restrito a uma pilha de inteiros; se quisermos uma pilha de `float`s, provavelmente teremos de copiar e editar o código.

Por outro lado, a Figura 13.2 define um tipo pilha que permite ao programador declarar variáveis desse tipo. Esse arquivo de cabeçalho fornece uma especificação do tipo `STACK`; normalmente ele seria incluído em dois lugares: o arquivo de implementação (Figura 13.3) e qualquer programa cliente que use esse tipo. Nesse caso, a pilha é, na realidade, um ponteiro

1. Lembre-se do que foi dito no Capítulo 5, que um *tipo* é um conjunto de valores de dados junto a um conjunto de operações sobre aqueles dados.

2. Note que C permite que tanto funções globais quanto dados sejam declarados como `static`, o que reforça esse ocultamento. No entanto, o uso da palavra-chave `static` para esse papel pode ser apontado como confuso. Esse uso da palavra-chave `static` para reforçar o ocultamento de informações em C não parece ser uma prática muito difundida.

```
#include "stack.h"

struct Node {
    int val;
    struct Node* next;
};
typedef struct Node* STACK;

STACK theStack = NULL;

int empty( ) {
    return theStack == NULL;
}

int pop( ) {
    STACK temp;
    int result = theStack->val;
    temp = theStack;
    theStack = theStack->next;
    free(temp);
    return result;
}

void push(int newval) {
    STACK temp = (STACK)malloc(sizeof(struct Node));
    temp->val = newval;
    temp->next = theStack;
    theStack = temp;
}

int top( ) {
    return theStack->val;
}
```

| Figura 13.1 Uma Pilha (Stack) Simples em C

para um registro que implementa uma lista vinculada. As operações dadas aqui restringem a visualização da lista vinculada de forma a ser aquela de uma pilha. Além disso, essa implementação requer que a pilha seja inicializada explicitamente (operação `newstack`), e a própria pilha deve ser passada como um argumento explícito para cada função.

O uso de compilação e ponteiros independentes em C permite a construção de um tipo pilha, que encapsula tanto os dados quanto as operações (ou funções) naqueles dados. No entanto, C sofre pelo fato de que ela permite acesso por parte de clientes da pilha (chamado de *acesso público*) à representação subjacente da pilha como uma lista vinculada. Isso pode ser um problema particular, se o cliente precisar estender o *tipo de dado abstrato*, mas não tiver acesso ao código-fonte; nesse caso, a tentação de acessar diretamente a representação subjacente pode ser irresistível.

```
struct Node {
    int val;
    struct Node* next;
};
typedef struct Node* STACK;

int empty(STACK theStack);
STACK newstack( );
int pop(STACK* theStack);
void push(STACK* theStack, int newval);
int top(STACK theStack);
```

| Figura 13.2 Um Tipo STACK (pilha) em C

```
#include "stack.h"

int empty(STACK theStack){
    return theStack == NULL;
}

STACK newstack( ) {
    return (STACK) NULL;
}
int pop(STACK* theStack) {
    STACK temp;
    int result = (*theStack)->val;
    temp = *theStack;
    *theStack = (*theStack)->next;
    free(temp);
    return result;
}

void push(STACK* theStack, int newval) {
    STACK temp;
    temp = (STACK)malloc(sizeof(struct Node));
    temp->val = newval;
    temp->next = *theStack;
    *theStack = temp;
}

int top(STACK theStack) {
    return theStack->val;
}
```

| Figura 13.3 Implementação do Tipo STACK (pilha) em C

O objetivo principal da abstração de dados é empacotar os tipos de dados e suas funções juntos em um único módulo, de forma que as funções proporcionem uma interface pública para o tipo. O suporte direto para esse conceito pode ser encontrado em "módulos" em Modula e em "packages" (pacotes) em Ada. O uso de módulos/pacotes permite a separação da interface lógica (freqüentemente chamada de "especificação") de sua implementação detalhada. Ou seja, os módulos/pacotes permitem ao programador restringir o acesso à representação concreta subjacente do tipo de dado abstrato e os detalhes de implementação de suas funções. Isso é chamado de *ocultação de informação*.

Ada usa pacotes para encapsular dados e operações nos dados, em suma, para encapsular um tipo de dado abstrato. Ada divide um pacote em uma parte de definição ou especificação e uma parte de implementação ou corpo. Diferentemente de C e C++, esses recursos de compilação separados asseguram verificação total de tipo dos pacotes. Isso tem a vantagem adicional de eliminar a necessidade de recompilar implementações de tipo de dado abstrato para cada novo cliente.

Nas Figuras 13.4 e 13.5, aparecem uma definição e uma implementação Ada package do tipo `STACK`, comparável àquela dada em C. Há uma grande diferença, no entanto, no suporte Ada aos genéricos. Em lugar de uma pilha de inteiros, a implementação Ada suporta pilhas de qualquer tipo.

Vamos ignorar o tipo genérico `element`, por enquanto, na Figura 13.4. Exceto quanto à sintaxe, as declarações de procedimento se parecem com um arquivo de cabeçalho. O tipo `stack` é público (ele aparece na lista de parâmetros de todos os procedimentos), mas é declarado como sendo `private`, e, sem dúvida, a parte privada da especificação declara uma pilha como sendo um ponteiro (tipo `access`) para um `node`. Este último é um registro que contém um valor do tipo `element` e um ponteiro `next`. Pelo fato de estes serem declarados como privados, o programador cliente não pode acessar partes do `node` nem acessar os dados apontados pelo ponteiro pilha. Somente o compilador pode usar esse conhecimento, que é um compromisso aqui entre um desejo de ocultar completamente as informações e o requisito de que a linguagem seja implementável.

```
generic
   type element is private;

package stack_pck is
   type stack is private;
   procedure push (in out s : stack; i : element);
   function pop (in out s : stack) return element;
   function empty(in s : stack) return boolean;
   function top(in s : stack) return element;

private
   type node;
   type stack is access node;
   type node is record
      val : element;
      next : stack;
   end record;
end stack_pck;
```

| Figura 13.4 Especificação Genérica de Stack (Pilha) em Ada

```
package body stack_pck is
   procedure push (in out s : stack; i : element) is
      temp : stack;
   begin
      temp := new node;
      temp.all := (val => i, next => s);
      s := temp;
   end push;

   function pop (in out s : stack) return element is
      temp : stack;
      elem : element;
   begin
      elem := s.all.val;
      temp := s;
      s := temp.all.next;
      dispose(temp);
      return elem;
   end pop;

   function empty(in s : stack) return boolean is
   begin
      return s = null;
   end empty;

   function top(in s : stack) return element is
   begin
      return s.all.val;
   end top;
end stack_pck;
```

| Figura 13.5 Implementação Genérica de Stack (Pilha) em Ada

A Figura 13.5 dá a implementação de pilha. A não ser pela sintaxe, o código é muito similar à implementação C. Examinando o código para um `push`, vemos que à variável `temp` é atribuído o ponteiro retornado pela expressão `new node`; recorde-se que o tipo `stack` é `access node`, isto é, um ponteiro para um nó. Em lugar do acesso de C via `->`, Ada usa o especificador `all`.

Similarmente, o código `pop` deverá ser familiar, exceto pela sintaxe. Primeiro, é feita uma cópia do valor a ser retornado. Em seguida, é definido um (valor) temporário para apontar para o início atual da lista, de forma que o ponteiro `s` da pilha pode ser avançado um nó. A operação `dispose` libera o nó originalmente na frente da lista e é retornada a cópia do valor originalmente na frente da lista.

Observe que nem `pop` nem `top` verificam se a pilha está vazia; isso é responsabilidade do cliente. Qualquer uma dessas operações sobre uma pilha vazia gerará uma exceção, que pode ser capturada pelo cliente. Isso, novamente, enfatiza a habilidade Ada para se escrever software robusto e confiável.

Para realmente usar esse código, a pilha genérica deve ser instanciada para algum tipo. Se quisermos imitar o código C, que usa pilha de inteiros, podemos instanciar o pacote via:

```
package int_stack is new stack_pck(element => integer);
```

Assim, podemos criar uma pilha de qualquer tipo individual, isto é, a pilha deve ser homogênea.

13.2 O MODELO OBJETO

Mesmo com a evolução dos módulos e dos pacotes nos anos 80, os problemas continuavam. Um desses problemas era que não havia nenhum mecanismo para inicialização e finalização automática de um valor do tipo fornecido. As inicializações normalmente necessárias incluem abertura de um arquivo, alocação de memória e inicialização de variáveis locais ao módulo. As finalizações importantes incluem fechamento de arquivos e liberação de memória. Outro problema com os módulos era que eles não proporcionavam uma maneira simples de estender uma abstração de dados adicionando novas operações. Ambos os problemas foram resolvidos pela introdução da idéia de uma "classe", o elemento fundamental para as linguagens orientadas a objetos.

No entanto, apareceu um problema mais importante nos anos 80, quando programadores e projetistas de linguagem começaram a perceber que uma gama significativa de aplicações não era bem servida pelo paradigma imperativo da decomposição funcional e abstração de dados. Era preciso mais. Por exemplo, a programação de computadores embarcados (*embedded*) e Interfaces Gráficas de Usuário (graphical user interfaces – GUIs) podiam ser mais bem modeladas como uma coleção de objetos, comunicando-se uns com os outros, enviando e recebendo mensagens. Uma aplicação GUI, por exemplo, é modelada mais naturalmente como uma coleção de diferentes tipos de objetos – botões, áreas de texto, imagens, videoclipes e menus desdobráveis –, cada um comunicando-se com o programa e com o usuário, mandando e recebendo mensagens. Para grandes programas, em geral, o paradigma orientado a objetos é uma maneira fundamentalmente diferente de pensar sobre programas. Para um tratamento completo de projeto e programação orientada a objetos, veja Budd, 2000, e Horstmann, 2004.

Assim, a *programação orientada a objetos* surgiu como um estilo popular de programação[3] no qual a decomposição de objeto se tornou uma preocupação central, em lugar da decomposição funcional e da abstração de dados. Nas próximas seções, usamos Java para explicar as principais características de uma linguagem orientada a objetos. Nosso propósito não é explicar Java, mas, sim, identificar as principais idéias em linguagens orientadas a objetos, como encapsulamento, visibilidade e herança.

13.2.1 Classes

Uma linguagem orientada a objetos suporta o encapsulamento dos tipos de dados com suas funções e as características de ocultação de informação da abstração de dados.

> **Definição:** Uma *classe* é uma declaração de tipo que encapsula constantes, variáveis e funções para manipulação dessas variáveis.

3. Embora essas idéias já estivessem por aí há muito tempo – desde o projeto da Simula nos anos 60 e o desenvolvimento da Smalltalk nos anos 70 –, só mais tarde apareceu uma orientação prática clara para o uso do paradigma orientado a objetos.

Uma classe é, por si mesma, um tipo de dado abstrato e um mecanismo para definir um tipo de dado abstrato em um programa. No idioma próprio da programação orientada a objetos, as variáveis locais de uma classe são chamadas *variáveis de instância,* suas inicializações são obtidas por funções especiais chamadas *construtores,* suas finalizações, por *destruidores*, e outras funções são implementadas por *métodos*. Cada instância de uma classe é um *objeto*.

Uma implementação do tipo de dado abstrato `stack`, como uma classe Java `MyStack`, é dada na Figura 13.6.

Java também suporta a idéia de uma *classe interna*.

> **Definição:** Uma *classe interna* é uma definição de classe que está totalmente encaixada dentro de uma outra classe.

A classe `Node` é um exemplo de uma classe interna, mas, como ela não tem outros métodos a não ser construtores, ela é mais uma C `struct` do que uma verdadeira classe.

A classe interna `Node` tem duas variáveis de instância (`val` e `next`) e um construtor denominado `Node`, sem nenhum tipo de retorno explícito. Semelhantemente, a classe `MyStack` tem uma variável de instância (`theStack`), um construtor (`MyStack`) e métodos (`empty`, `top`, `push` e `pop`). Não há destruidores para qualquer uma das classes, já que Java executa a coleta automática de lixo da memória e, infelizmente, não suporta destruidores.

Uma vantagem da abordagem orientada a objetos para a abstração de dados é que ela proporciona um *construtor*, no qual o espaço *heap* é alocado para o objeto e as variáveis de instância do objeto podem ser inicializadas. No construtor para `Node`, ambas as variáveis

```
class MyStack {
    class Node {
        Object val;
        Node next;
        Node(Object v, Node n) { val = v; next = n; }
    }
    Node theStack;

    MyStack( ) { theStack = null; }

    boolean empty( ) { return theStack == null; }

    Object pop( ) {
        Object result = theStack.val;
        theStack = theStack.next;
        return result;
    }

    Object top( ) { return theStack.val; }

    void push(Object v) {
        theStack = new Node(v, theStack);
    }
}
```

| Figura 13.6 Uma Classe Stack Simples em Java

de instância são valores dados. Sem dúvida, um uso primário de um construtor é garantir que todas as variáveis de instância sejam corretamente inicializadas.

> **Definição:** Um *cliente* de uma classe C é qualquer outra classe ou um método que declara ou usa um objeto da classe C.

A declaração e a inicialização de uma variável dessas em Java criam um *objeto* no *heap*, pois Java usa semântica de referência para objetos. Note que a classe `MyStack` é um cliente da classe `Node`, já que a variável de instância `theStack` é um `Node`.

Considere o seguinte código cliente:

```
MyStack s = new MyStack();
```

Note que a classe `MyStack` tem um construtor com argumento zero explicitamente definido.

Após essa declaração, um cliente pode agora especificar qualquer número de operações `push` e `pop` sobre esse objeto (usando a notação de ponto em Java). Por exemplo, veja aqui uma seqüência de instruções que deixa `MyStack s` com os valores 1 e 5 (5 no topo):

```
s.push(1);
s.push(2);
s.pop();
s.push(5);
```

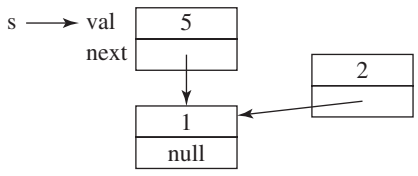
O efeito dessa declaração e das quatro instruções é mostrado na Figura 13.7. Note aqui que existem agora três nós no *heap*, um dos quais se tornou "lixo" em resultado da chamada `s.pop()`. O `Node` colocado no topo da pilha mais recentemente é referenciado por `s`.

Em linguagens que suportam destruidores, o destruidor de um objeto é chamado automaticamente quando o objeto é desalocado. Para objetos de classes como `MyStack`, um dos usos de um destruidor é para desalocar memória para objetos que não podem mais ser referenciados. Linguagens com coleta automática de lixo, como Java, não precisam de um destruidor nesse caso. Outro uso de um destruidor seria para liberar recursos e fechar arquivos.

Construtores são exemplos de *métodos de classe*, às vezes também conhecidos (infelizmente) como *métodos estáticos*. Construtores usualmente são invocados por meio da operação `new`. Outros métodos de classe podem ser invocados por intermédio do nome da classe (por exemplo, `Math.max`), com o endereço computado efetivamente em tempo de compilação.

O oposto dos métodos estáticos ou de classe são os *métodos de instância*, que devem ser invocados por meio de um objeto. Os usos dos métodos de instância são discutidos na Seção 13.2.3, sobre herança, e suas implementações, na Seção 13.2.9. No restante da Seção 13.2, o leitor deve admitir que todos os métodos, exceto os construtores, são métodos de instância, a menos que seja dito o contrário.

Figura 13.7 Criação de uma Stack (Pilha) com Dois Valores



Uma maneira importante de ver um programa orientado a objetos é que ele é uma coleção de objetos em interação que se comunicam passando mensagens. Com efeito, cada objeto pode ser visto como uma máquina separada que possui dados e operações sobre aqueles dados. Particularmente, vários objetos podem ser instanciações diferentes da mesma classe, caso em que as operações são as mesmas, enquanto os dados são distintos. Em nosso exemplo de pilha, o programa poderia declarar vários objetos do tipo `MyStack`, cada um com sua própria coleção de dados.

Os objetos se comunicam passando mensagens entre eles. Quando muito, uma máquina (objeto) pode estar executando de cada vez. Quando uma máquina passa uma mensagem para outra máquina, ela invoca um método particular (função) da máquina receptora, passa os parâmetros (conteúdo da mensagem) apropriados para o método e, então, espera até que a máquina receptora retorne uma resposta (valor de retorno).

O conjunto de comportamentos que permite a um programa criar qualquer número de instâncias de um tipo de dado abstrato geralmente é caracterizado como uma *linguagem baseada em objetos*. Essa também é uma nova maneira de pensar sobre programação, pois os dados não vão mais sofrer passivamente o efeito das funções, mas tornam-se ativos.

As classes desempenham dois papéis complementares importantes em linguagens baseadas em objeto. Primeiro, elas determinam o tipo de um objeto, de forma que definem que mensagens (chamadas de método) são legais e precisamente que método está sendo chamado. Segundo, elas permitem verificação total de tipo. Em uma linguagem estaticamente tipada como Java, essas verificações são feitas em tempo de compilação, enquanto em uma linguagem dinamicamente tipada, como a Smalltalk, as verificações são feitas em tempo de execução.

13.2.2 Visibilidade e Ocultamento de Informação

O ocultamento de informação é tornado explícito requerendo que todo método e toda variável de instância em uma classe tenham um nível particular de visibilidade – `public`, `protected` ou `private` – com relação às suas subclasses e às classes cliente.

Uma variável ou um método `public` é visível a qualquer cliente e subclasse de uma classe. Os métodos públicos de uma classe definem a interface da classe com o mundo externo. A maioria dos métodos de uma classe são públicos.

Uma variável ou um método `protected` é visível somente para uma subclasse da classe.[4] Isso proporciona às subclasses o acesso direto às variáveis de instância protegidas e aos métodos da classe. Ilustraremos a utilidade desse nível de acesso quando discutirmos a *herança* na Seção 13.2.3.

Uma variável ou um método `private` é visível à classe corrente, mas não às suas subclasses ou aos seus clientes. Seguindo a boa prática no projeto de software, é comum tornar privadas as variáveis de instância e permitir o acesso de subclasse somente por meio do uso de métodos públicos ou protegidos. Isso minimiza as alterações que teriam de ser feitas se a representação concreta do tipo de dado abstrato tivesse de ser alterada.

Para ilustrar essas idéias mais concretamente, é apresentada, na Figura 13.8, uma reimplementação da classe `MyStack` (Figura 13.7) com modificadores de visibilidade. Na classe `MyStack`, a variável de instância `theStack` é privada e, portanto, acessível somente dentro da própria classe. O construtor e os métodos são públicos, acessíveis às subclasses e aos clientes.

4. Em uma escolha infeliz de projeto, Java torna os objetos e os métodos protegidos acessíveis a todas as classes definidas dentro do mesmo pacote. Por padrão, essa escolha rompe o encapsulamento, já que qualquer classe pode ser declarada como sendo parte de um pacote. Para uma discussão mais completa desse assunto, veja Niemeyer e Knudsen, 2002, Cap. 6.

```
public class MyStack {
    protected class Node {
        public Object val;
        public Node next;
        public Node(Object v, Node n) {
            val = v; next = n;
        }
    }
    private Node theStack;

    public MyStack( ) { theStack = null; }

    public boolean empty( ) { return theStack == null; }

    public Object top( ) { return theStack.val; }

    public Object pop( ) {
        Object result = theStack.val;
        theStack = theStack.next;
        return result;
    }

    public void push(Object v) {
        theStack = new Node(v, theStack);
    }
}
```

| Figura 13.8 Uma Classe Stack com Modificadores de Visibilidade

A classe encaixada ou interna `Node` é protegida, o que significa que ela é acessível a subclasses, mas não a clientes. Porém, suas variáveis de instância e seu construtor são `public`. Se eles precisam ser acessíveis a subclasses, não podem ser `protected` ou `private`.

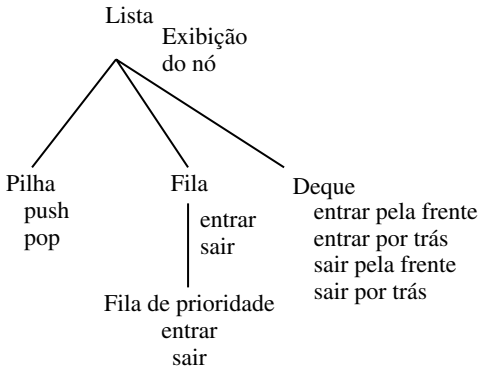
13.2.3 Herança

O paradigma orientado a objetos suporta reutilização de código por intermédio da *herança.* As classes existem em uma linguagem orientada a objetos em uma hierarquia de classes. Uma classe pode ser declarada como uma *subclasse* de outra classe, que é chamada de *classe-mãe* ou *superclasse*. Dentro dessa hierarquia, cada subclasse é capaz de herdar variáveis e métodos de sua classe-mãe simplesmente em virtude do fato de que ela é uma subclasse.

Em projeto orientado a objetos, a relação entre uma subclasse e uma superclasse é estabelecida principalmente pelo fato de existir ou não entre elas uma relação do tipo *é-uma* (*ou é-um*). Por exemplo, uma pilha *é um* tipo de lista como também é uma fila, uma *deque* e uma fila de prioridades. Além disso, uma fila de prioridades *é um* tipo de fila.

Um engano comum cometido pelos principiantes em linguagem orientada a objetos é confundir a relação *é-uma* (*é-um*) com a relação *tem-uma* (*tem-um*). Essa última não identifica uma relação subclasse–superclasse; em vez disso, ela identifica uma classe como um cliente de outra classe, usualmente chamada *agregação*.

Figura 13.9 Uma Hierarquia de Classe Simples



> **Definição:** Uma classe C_1 é uma agregação de uma classe C_2, se C_1 contém objetos do tipo C_2.

Por exemplo, um programa que declara uma variável da classe `Stack` é um cliente, não uma subclasse, da classe `Stack`.

Assim, esperamos que um projeto orientado a objetos possa implementar essas várias classes em uma hierarquia como aquela mostrada na Figura 13.9.

A herança aparece em duas variedades: *herança simples* e *herança múltipla*. Essas duas variedades são diferenciadas uma da outra conforme uma classe possa ter ou não apenas uma ou mais de uma classe-mãe, respectivamente. C++, Python e Eiffel suportam herança múltipla, enquanto Java e Smalltalk suportam somente herança simples.

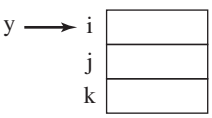
Na herança simples, a hierarquia de classe forma uma árvore com sua raiz na classe mais geral, por exemplo, *Object*. Uma classe D é uma subclasse de outra classe C quando ela estende ou especializa o significado da classe C e acrescenta novas variáveis de instância ou métodos, ou modifica as definições dos métodos público e protegido de C. Uma definição Java de classe usa a palavra-chave `extends` para identificar sua superclasse; se essa palavra-chave for omitida, a classe é simplesmente uma subclasse da classe mais geral *Object.*

> **Definição:** Supondo-se que uma subclasse simples D estende uma classe C, dizemos que a subclasse D *herda* todas as variáveis de instância e os métodos de C, incluindo construtores e destruidores.

Naturalmente, se algumas dessas variáveis ou desses métodos herdados forem declarados como privados, a referência direta pela subclasse D não será possível. No entanto, o acesso a objetos e a métodos privados pode ser possível por meio de métodos protegidos.

Por exemplo, suponha que a classe C tenha variáveis de instância `i` e `j` e a subclasse D tenha uma variável de instância `k`. Um objeto cliente `y` da classe `D` cria um objeto com as variáveis de instância mostradas na Figura 13.10. Isso é verdade independentemente da visibilidade de `i` e `j`, desde que as variáveis não sejam declaradas estáticas (ou variáveis de classe).

Figura 13.10 Inicialização de um Objeto com Características Herdadas



Para um exemplo mais concreto, considere as definições parciais de classe para `List`, `Stack`, `Queue`, `Deque` e `PriorityQueue` dadas na Figura 13.11. Essa hierarquia posiciona a definição de classe interna `Node` de forma que ela pode ser utilizada por todas as classes mostradas na Figura 13.9. Isto é, todas as subclasses da classe `List` herdam a classe `Node` e suas características.

Repare que as únicas características dessas classes que os clientes podem ver e utilizar são os métodos públicos `pop`, `push`, `enter`, `leave` e assim por diante. Note também nesse exemplo que `enter` e `leave` são redefinidas na classe `PriorityQueue`, já que suas semânticas são diferentes daquelas da classe `Queue`. Ou seja, é necessário um parâmetro extra para o método `enter`, e a escolha de qual nó selecionar em um método `leave` é diferente para um método `PriorityQueue` e para um simples método `Queue`. Finalmente, observe que as classes `Queue` e `PriorityQueue` requerem as variáveis `front` e `rear`, que `PriorityQueue` pode herdar da classe `Queue`. `Stack` não requer essas variáveis, ao contrário, sua variável local `theStack` é definida e usada privativamente por seus métodos. Cada subclasse tem, portanto, uma configuração diferente de métodos públicos e variáveis de instância, identificando as maneiras única e sobreposta pelas quais ela acessa os elementos na estrutura básica de dados `List`. *Essa é a essência da idéia da reutilização de código via herança em Programação Orientada a Objetos.*

A propósito, vale destacar que a classe interna `Stack` em Java é uma subclasse de sua classe `Vector`. Dessa forma, a classe `Stack` pode não apenas chamar os métodos usuais `push` e `pop`, mas também pode chamar métodos como `insertElementAt` e `elementAt`, que estão publicamente disponíveis e são herdados da classe-mãe `Vector`. Esses últimos métodos permitem a inserção e o acesso de objetos em um ponto arbitrário na pilha. Devido ao fato de que essa inserção e o apagamento arbitrário não são comportamentos "tipo pilha", a classe `Stack` de Java tem sido criticada por violar a relação *é-uma* (*é-um*). Em vez disso, a classe `Stack` de Java deveria ter usado *agregação*, na qual a pilha tem uma variável de instância do tipo `Vector`.

Embora estejamos criticando hierarquias de classe, aquela apresentada na Figura 13.11 usa herança onde ela usaria *agregação*, isto é, declara uma variável de instância de uma implementação de lista geral tal como `ArrayList` ou `LinkedList`. Em um caso desses, virtualmente, todos os métodos de cada classe podem ser implementados em uma única instrução; por exemplo:

```
public class LinkedStack( ) {
    private LinkedList stack = new LinkedList( );
    public void push(Object val) { stack.addFirst(val); }
    public Object pop( ) { return stack.removeFirst( ); }
    ...
}
```

Dessa discussão não se deve concluir que é preferível a agregação e não a herança. Cada uma tem seu lugar. Um bom programador reconhece qual é apropriada em cada contexto.

13.2.4 Herança Múltipla

Em contraste com as linguagens de herança simples, algumas linguagens orientadas a objetos suportam *herança múltipla*, que permite que uma classe seja uma subclasse de uma *ou mais* superclasses.

Sob a herança múltipla, uma classe pode herdar variáveis de instância e métodos de várias classes distintas. Há muitas situações práticas nas quais essa é uma ferramenta útil de modelagem. Considere, por exemplo, a hierarquia na Figura 13.12. Nela, nós vemos

```
public class List extends Object {
    protected class Node {
        public Object val;
        public int priority;
        public Node prev, next;
        public Node (Object v, Node p) {
            val = v; prev = p; next = null; priority = 0; }
        public Node (Object v, Node p, Node n ) {
            val = v; prev = p; next = n; priority = 0; }
        public Node (Object v, int pr, Node p, Node n) {
            val = v; priority = pr; prev = p; next = n; }
    }
}
public class Stack extends List {
    private Node theStack;
    public MyStack( ) { theStack = null; }
    public Object pop( ) {
        Object result = theStack.val;
        theStack = theStack.next;
        return result;
    }
    public void push(Object v) {
        theStack = new Node(v, theStack);
    }
    ...
}
public class Queue extends List {
    protected Node front = null, rear = null ;
    public void enter(Object v) { ... }
    public Object leave( ) { ... }
    ...
}
public class PriorityQueue extends Queue {
    public void enter(Object v, int pr) { ... }
    public Object leave( ) { ... }
    ...
}
public class Deque extends List {
    public void enterfront(Object v) { ... }
    public void enterrear(Object v) { ... }
    public Object leavefront( ) { ... }
    public Object leaverear( ) { ... }
    ...
}
```

| Figura 13.11 Implementação Java Parcial da Hierarquia na Figura 13.9

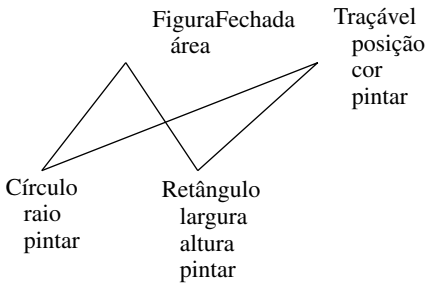


Figura 13.12
Um Exemplo de Herança Múltipla

a modelagem de círculos e retângulos como tipos particulares de figuras fechadas. Simultaneamente, se acrescentarmos características da classe `Drawable`, círculos e retângulos ganharão características (posição e cor) que permitem que eles sejam desenhados em uma tela gráfica. Assim, por exemplo, uma classe `Circle` herda as características área, posição, cor e preenchimento de duas superclasses diferentes simultaneamente.

A herança múltipla tem vantagens e desvantagens. Conforme mostra esse exemplo simples, ela facilita a reutilização de código permitindo a uma subclasse herdar características de várias classes-mãe.

Uma desvantagem da herança múltipla é que sua semântica se torna muito complicada em certas circunstâncias. Por exemplo, se uma classe E tem subclasses B e C, e um método M é definido diferentemente em B e C, que implementação de M deveria ser herdada por E: aquela em B, aquela em C ou ambas?

Além disso, suponha que o método M seja virtual. Pode a classe E anular a versão de B do método M, mas não a versão de C? Pode a classe E anular ambas as versões do método M? Se assim for, como as chamadas para as duas versões podem ser diferenciadas?

Embora esteja claro que esses problemas podem ser resolvidos, fazer isso complica a semântica de uma linguagem com herança múltipla. Também complica a implementação dessa linguagem. Uma complicação final é que a experiência no projeto de hierarquias de classe com herança múltipla não é tão extensa quanto a experiência para herança simples. Por exemplo, no livro *Design Patterns* (Gamma et al., 1995) é mencionada a herança múltipla em conjunto com apenas dois de seus muitos padrões.

Neste ponto, resumimos todas as características essenciais de uma linguagem orientada a objetos.

> **Definição:** Uma linguagem é *orientada a objetos* se ela suporta um mecanismo de encapsulamento com ocultamento de informação para definir tipos de dados abstratos, métodos virtuais e herança.

A maior parte das demais subseções desta seção discute acréscimos úteis ao modelo objeto, particularmente para linguagens estaticamente tipadas.

13.2.5 Polimorfismo

De acordo com o dicionário, o termo *polimórfico* significa "ter muitas formas".

> **Definição:** Em linguagens orientadas a objetos, *polimorfismo* refere-se à ligação tardia de uma chamada a uma ou várias diferentes implementações de um método em uma hierarquia de herança.

Esses métodos devem ser métodos de instância. Conforme veremos no Capítulo 14, esse uso do termo *polimorfismo* é diferente de seu uso em programação funcional. Veja também a discussão na Seção 4.8.

Como exemplo, suponha que temos um objeto `obj` de um tipo declarado (ou aparente) `T` cujo tipo real pode variar em tempo de execução. Considere a seguinte chamada de método:

```
obj.m();
```

O tipo `T` de `obj` garante que todos os subtipos de `T` implementem um método `m` apropriado; uma linguagem estaticamente tipada verifica isso em tempo de compilação. A verdadeira instância do método `m` que é chamado variará dependendo do tipo *real* do objeto `obj` que ocorre em tempo de execução. Portanto, o método `m` é polimorfo.

Por exemplo, considere a hierarquia de classe mostrada na Figura 13.13, junto com a declaração Java:

```
Expression e;
```

Durante o decurso de análise de um programa Clite, o analisador pode atribuir a `e` um valor em uma das várias diferentes subclasses: `Variable`, `Binary`, `Unary` e assim por diante. Eventualmente, `e` deve ser exibido por intermédio da chamada:

```
e.display( );
```

Uma de várias implementações diferentes de `display`, que corresponde à subclasse de `Expression`, à qual pertence o valor de `e`, determina a ação correspondente a essa chamada. O método de instância `display`, nesse caso, é polimórfico.

Para usar outro exemplo, suponha que temos uma aplicação que implementa um programa de desenho ou um sistema de algoritmo de animação. Em um programa desses, diferentes tipos de objetos gráficos (círculos, retângulos etc.) devem poder ser desenhados em uma janela. Suponha que cada objeto gráfico da classe `Drawable` tem um método `paint` (seguindo a convenção Java AWT). Em algum lugar, o sistema mantém uma lista (`myList`) desses objetos e, periodicamente, pede a cada objeto que se redesenhe:

```
for (Drawable obj : myList)
   obj.paint();
```

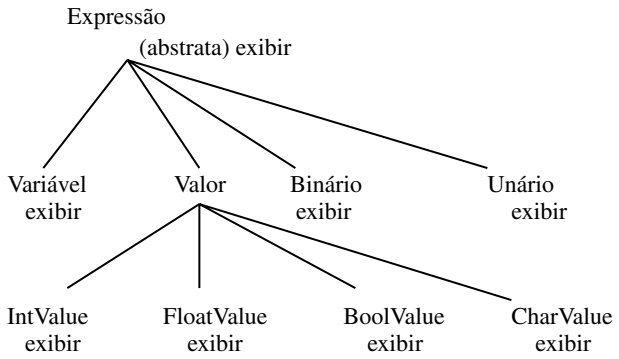


Figura 13.13
Implementação da Expressão Clite como uma Classe Abstrata

Quando esse código é executado, cada objeto gráfico desenha a si próprio de acordo com sua própria lógica; quadrados são desenhados diferentemente dos círculos, por exemplo. Assim, o verdadeiro método `paint` que está sendo chamado varia de acordo com o tipo atual (dinâmico) do objeto gráfico que está sendo desenhado. Ainda, abstratamente, cada um é meramente solicitado a se redesenhar. *Essa é a essência do polimorfismo em Programação Orientada a Objetos.*

Para ser verdadeiramente polimórficas, as várias instâncias do método `paint` devem suportar o princípio da *substituição* (Liskov e Guttag, 2001). Isto é, cada instância deve executar a mesma função abstrata, com apenas o código sendo particularizado pelo objeto gráfico que está sendo desenhado.

> **Definição:** Um método de subclasse é *substituível* por um método de classe-mãe se o método da subclasse executa a mesma função geral.

Assim, para cada tipo diferente de objeto gráfico, o comportamento lógico da função pintar deve ser transparente para o chamador. Sem o princípio da substituição, o código dado aqui não funcionaria.

Em Java, por exemplo, o tipo declarado de objeto que está envolvido na chamada de método polimórfica pode ser um tipo concreto, um tipo abstrato ou um tipo interface. O compilador verifica em tempo de compilação que um método com a assinatura apropriada existe para o objeto especificado, usando seu tipo declarado e a hierarquia de classe dentro da qual o tipo existe. No entanto, um compilador não pode garantir que a função abstrata executada seja logicamente equivalente; isto é, ele não pode verificar que os métodos, do modo como são implementados, sustentam o princípio da substituição. Isso é responsabilidade do programador.

13.2.6 Modelos

Uma característica conveniente de uma linguagem orientada a objetos estaticamente tipada é o suporte aos *modelos*, também conhecidos como *genéricos*. Conforme ficou demonstrado por Java antes da versão 1.5, a falta dos modelos em uma linguagem orientada a objetos cria uma *overhead* de programação significativa em termos da necessidade de conversão de tipo de dados. Os modelos são bem suportados em Java (denominados genéricos na versão 1.5), Ada (também denominados genéricos) e C++. A discussão a seguir usa Java 1.5 para ilustrar as idéias básicas.

> **Definição:** Um *modelo* define uma família de classes parametrizadas por um ou mais tipos.

Classes coleção são, muitas vezes, usadas para ilustrar esse conceito, em que uma *classe coleção* é uma classe usada para armazenar valores de objetos para uma recuperação (acesso) eficiente. Os exemplos de coleções incluem listas, pilhas, filas, árvores e tabelas *hash*.

Em Java, antes da versão 1.5, os programadores tinham de monitorar explicitamente a classe de cada elemento armazenado em uma coleção, tanto no instante de sua entrada quanto no instante de seu exame ou de sua remoção. Por exemplo, vamos ampliar o exemplo gráfico em `Drawable` da Seção 13.2.5:

```
1 ArrayList myList = new ArrayList( );
2 ...
3 for (Iterator i = myList.iterator( ); i.hasNext( ); ) {
4    Drawable obj = (Drawable) i.next( );
5    obj.paint( );
6 }
```

A variável `myList` se destina a conter objetos `Drawable`, isto é, quadrados, círculos e semelhantes. No entanto, as assinaturas de tipos dos vários métodos que acrescentam objetos a uma `ArrayList` admitem qualquer subclasse do tipo `Object`, isto é, absolutamente qualquer objeto. Se acaso o programador inserir um objeto que não seja `Drawable` ou uma de suas subclasses, a conversão na linha 4 falhará e irá gerar uma exceção em tempo de execução. O programador deve então encontrar o ponto no programa em que um objeto do tipo errado foi inserido em `myList`.

Em linguagens estaticamente tipadas, podem ser usados modelos para identificar o tipo de objeto armazenado em uma coleção, por exemplo:

```
ArrayList<Drawable> myList = new arrayList<Drawable>( );
```

Agora, qualquer tentativa de acrescentar um objeto não-`Drawable` a `myList` gera um erro em tempo de compilação. Conversão explícita não é mais necessária para acessar um valor de `myList`.

Em linguagens de herança simples e múltipla que suportam modelos, podem ser escritas novas classes parametrizadas. Considere o seguinte:

```
import java.util.ArrayList;
public class MyList<T> {
    private ArrayList<T> list;
    public MyList( ) { list = new ArrayList<T>( ); }
    public void add(T obj) { list.add(obj); }
    ...
}
```

O nome `T` é arbitrário e se refere meramente ao tipo ou à classe de objeto que cada instância de `MyList` irá conter. Observe o uso do nome `T` no método `add`.

13.2.7 Classes Abstratas

Abstração é uma característica fundamental da programação e do projeto orientados a objetos. Essa característica permite que grandes sistemas sejam especificados em um nível muito geral, muito antes de ocorrer a implementação dos métodos individuais.

Muitas linguagens orientadas a objetos permitem que uma classe seja declarada *abstrata*, uma condição que automaticamente ocorre sempre que um ou mais de seus métodos é `abstract`.

> **Definição:** Uma *classe abstrata* é uma classe declarada como abstrata ou que tem um ou mais métodos abstratos.
>
> **Definição:** Um *método abstrato* é um método que não contém código além de sua assinatura.

Qualquer subclasse de uma classe abstrata que não fornece uma implementação de um método abstrato herdado é por si só abstrata. Devido ao fato de as classes abstratas terem métodos que não podem ser executados, programas cliente não podem inicializar um objeto que é um membro de uma classe abstrata. Essa restrição assegura que uma chamada não será feita a um método abstrato (não-implementado).

Uma classe pode ser explicitamente declarada como abstrata mesmo que ela não tenha métodos abstratos. Ocorre um caso desses quando a classe representa um conceito abstrato que unifica várias classes concretas.

Nossa implementação de uma *Expressão* Clite (Clite *Expression*) da Figura 2.14 fornece um exemplo de uma classe abstrata cujas subclasses são *Binary*, *Unary*, *Variable* e *Value*.

Graficamente, essa estrutura está resumida na Figura 13.13. Aqui, a presença de uma assinatura do método abstrato `display` torna a própria `Expression` uma classe abstrata e requer que todas as subclasses de `Expression` implementem um método com a mesma assinatura. Isso é apropriado para nossa classe `Expression`, pois cada uma de suas subclasses precisa mostrar uma configuração diferente de dados. Observe, no entanto, que a subclasse `Value` de `Expression` também é declarada como abstrata. Mesmo sem o método `display`, tanto `Expression` quanto `Value` devem permanecer abstratas, pois cada uma representa uma abstração de várias classes concretas.

Nesse exemplo, um cliente não pode criar um objeto concreto `Expression`. Qualquer tentativa de fazer isso seria identificada como um erro no momento da compilação. O esqueleto de código Java a seguir expõe a estrutura de classe na Figura 13.13.

```
abstract class Expression { ... }
   class Variable extends Expression { ... }
   abstract class Value extends Expression { ... }
      class IntValue extends Value { ... }
      class BoolValue extends Value { ... }
      class FloatValue extends Value { ... }
      class CharValue extends Value { ... }
   class Binary extends Expression { ... }
   class Unary extends Expression { ... }
```

13.2.8 Interfaces

Além das classes abstratas, Java também suporta uma estrutura relacionada denominada interface. Logicamente, uma interface pode ser considerada uma extensão do recurso `typedef` de C.

> **Definição:** Uma *interface* encapsula uma coleção de constantes e assinaturas de métodos abstratos. Uma interface não pode incluir variáveis, construtores ou métodos não-abstratos.

Sintaticamente, uma declaração de interface se parece com uma definição de classe abstrata, exceto pelo fato de a palavra `interface` substituir a palavra `class`. Como exemplo, considere a definição da interface Java `Map` e alguns de seus principais métodos:

```
public interface Map {
    public abstract boolean containsKey(Object key);
    public abstract boolean containsValue(Object value);
    public abstract boolean equals(Object o);
    public abstract Object get(Object key);
    public abstract Object remove(Object key);
    ...
}
```

A principal diferença entre uma interface e uma classe abstrata é que *todos* os métodos em uma interface devem ser abstratos; somente constantes (variáveis `final`) podem ser declaradas concretamente lá. No entanto, em uma classe abstrata, objetos podem ser declarados e alguns métodos podem ser totalmente implementados. Devido ao fato de não ser uma classe, uma interface não tem um construtor, mas tem uma classe abstrata.

Algumas pessoas gostam de pensar em uma interface como uma alternativa à herança múltipla. Rigorosamente falando, no entanto, uma interface não é exatamente a mesma coisa, já que ela não proporciona um meio de reutilização de código; isto é, todos os seus métodos devem ser abstratos. Uma interface é similar à herança múltipla no sentido de que uma interface é um tipo. Uma classe que implementa múltiplas interfaces parece ser muitos tipos diferentes, um para cada interface.

A classe Java `HashMap` é uma subclasse da classe abstrata `AbstractMap` e também desfruta características de três interfaces distintas. Assim, `HashMap` pode ser vista como usando herança múltipla para obter várias assinaturas de métodos de diferentes fontes. Os métodos `entrySet` e `keySet` são obtidos da superclasse `AbstractMap`, enquanto os métodos `containsKey` e `containsValue` são obtidos da interface `Map`. Um cliente da classe `HashMap`, portanto, tem acesso a uma combinação de características vindas de quatro diferentes classes. Um objeto da classe `HashMap` pode ser visto como sendo qualquer um dos seguintes tipos: `AbstractMap`, `HashMap`, `Map`, `Cloneable`, `Serializable`.

Portanto, um uso comum das interfaces em Java é simular o efeito da herança múltipla ao mesmo tempo em que evita a armadilha da herança múltipla mencionada acima. Isso significa que a interface assegura que qualquer classe que a implemente deve proporcionar definições concretas de *todos* os seus métodos; nenhum é predefinido. Em outras palavras, se uma superclasse e uma interface implementada por uma classe contêm um método com a mesma assinatura, a própria classe é então obrigada a reimplementar aquele método, evitando, assim, a confusão que resultaria sob a herança múltipla.

Dizemos que uma classe `implementa` uma interface, não que a estende. Para implementar uma interface, uma classe concreta deve implementar todos os métodos de uma interface, da mesma forma como se estivesse tornando concreta uma classe abstrata. A não-implementação de todos os métodos de uma interface torna a classe resultante abstrata.

Uma interface comumente implementada em Java é `Comparable`:

```
public interface Comparable {
    public abstract int compareTo(Object obj);
}
```

A interface `compareTo` retorna um valor inteiro negativo se o receptor for menor do que `obj`, 0 se for igual e um valor positivo (≥ 1) se for maior. Assim, uma classe `Student` que permite ordenar por nome pode ser assim:

```
public class Student implements Comparable {
    private String lastName;
    private String firstName;
    ...
    public int compareTo(Object obj) {
        Student other = (Student) obj;
        int comp = lastName.compareTo(other.lastName);
        if (comp != 0) return comp;
        return firstName.compareTo(other.firstName);
    }
}
```

No método acima `compareTo`, o parâmetro deve ser do tipo `Object`, conforme especificado na interface `Comparable`. No entanto, um `Student` só pode ser comparado com outro

`Student`, daí a conversão na primeira linha do método. É responsabilidade do cliente garantir que a conversão não falhe.

Um conjunto de estudantes (`studentList`) poderia então ser facilmente ordenado usando:

```
java.util.Arrays.sort(studentList);
```

em que a assinatura é: `sort(Comparable[ ] obj)`. Isto é, o método de ordenação (sort) não se preocupa com o verdadeiro tipo de objeto que está sendo ordenado, apenas que ele implemente o método `compareTo`.

Outro exemplo do uso de interfaces é dado no programa Eight Queens (Oito Rainhas), na Seção 13.4.2.

13.2.9 Tabela de Método Virtual

A finalidade desta seção é explicar como o método virtual apropriado é chamado em tempo de execução. Para isso, vamos recordar nosso exemplo na Figura 13.11, na qual um objeto declarado como sendo da classe `MyList` em tempo de execução pode ser *qualquer* subclasse de `MyList`, como `MyStack` ou `Queue`. Considere o seguinte:

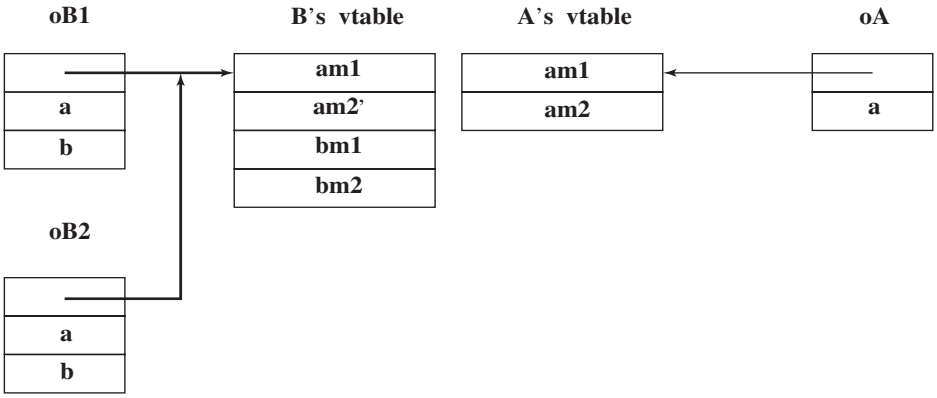
```
MyList myList;
...
System.out.println(myList.toString( ));
```

No instante da compilação, a classe em tempo de execução de `myList` é desconhecida; ela poderia ser uma `MyList`, uma `MyStack` ou outra subclasse de `MyList`. Assim o método real `toString` chamado pode ser o método que está definido na classe `MyList`, ou então ele pode ser aquele definido em uma de suas subclasses, dependendo da classe em tempo de execução de `myList`. Isso é claramente diferente das linguagens imperativas estaticamente tipadas, em que a função real que está sendo chamada é determinada em tempo de compilação.

O despacho dinâmico (*dynamic dispatching*) à base de método-por-método pode ser implementado por um dispositivo chamado Tabela de Método Virtual (*Virtual Method Table – VMT*). Abstratamente, considere a implementação a seguir. Cada classe tem sua própria VMT, e cada instância da classe tem uma referência (ou um ponteiro) para a VMT. Uma implementação simples da VMT seria uma tabela *hash*, que usa o nome do método (ou assinatura, no caso de sobrecarga) como a chave e o endereço em tempo de execução do método chamado como o valor.

No entanto, para linguagens estaticamente tipadas, a VMT é mantida como um conjunto. O método que está sendo chamado é convertido em um índice na VMT em tempo de compilação. Considere o seguinte exemplo:

```
class A {
    Obj a;
    void am1( ) { ... }
    void am2( ) { ... }
}
class B extends A {
    Obj b;
    void bm1( ) { ... }
    void bm2( ) { ... }
    void am2( ) { ... }
}
```



| Figura 13.14 Implementação VMT em Linguagens Estaticamente Tipadas

A classe `B` é uma subclasse de `A`. As classes `A` e `B` têm uma variável de instância e dois métodos virtuais. Além disso, a classe `B` sobrepõe o método `am2` na classe `A`.

A Figura 13.14 mostra a situação em que o programa tem dois objetos `B`, `oB1` e `oB2`, e um objeto `A`, `oA1`. Repare que as entradas de `oB1` e `oB2` da classe `B` contêm cada uma delas suas próprias cópias únicas das variáveis de instância não-estáticas para as classes `A` e `B`; além disso, cada objeto contém uma referência a uma tabela compartilhada de método virtual. Essa tabela contém uma entrada que tem uma referência para cada método virtual, incluindo aqueles da superclasse (e assim por diante, recursivamente). Isso ocorre independentemente da visibilidade dos métodos da superclasse. Lembre-se de que a classe `B` sobrepõe o método herdado `am2`; isso está representado no diagrama acrescentando-se uma plica (') na entrada na vtable de `B` para indicar que ela é distinta daquela outra na vtable de `A`. Esse exemplo indica claramente por que uma classe precisa de uma cópia da vtable de sua classe-mãe, em vez de meramente uma referência.

Conforme está ilustrado na Figura 13.14, ambos os métodos, `am2` na classe `A` e `am2'` (a `am2` sobreposta) na classe `B`, são armazenados na `vtable[1]`. Assim, dado o fragmento de código:

```
A b = new B( );
...
b.am2( );
```

o compilador chamará corretamente `am2` de `B`, já que o método chamado nesse caso é o método que está armazenado na `vtable[1]` do objeto `b`, qualquer que seja ele.

A implementação ilustrada na Figura 13.14 funciona bem para linguagens estaticamente tipadas com herança simples. Ellis e Stroustrup (1990) fornecem uma implementação similar para linguagens com herança múltipla.

Assim, o compilador mapeia a assinatura de cada método virtual em um índice na VMT. O custo em tempo de execução de se usar um método de instância (em vez de um método de classe) é o custo em tempo de execução de se usar um ponteiro (para acessar a VMT) mais um conjunto indexando a operação (para acessar o endereço do método).

13.2.10 Identificação em Tempo de Execução

Identificação de tipo em tempo de execução (Run-time type identification – RTTI) é a habilidade da linguagem para identificar em tempo de execução o tipo real ou a classe de um objeto. Recorde-se do Capítulo 5, em que é citado que todas as linguagens dinamicamente

tipadas têm essa habilidade, enquanto a maioria das linguagens imperativas estaticamente tipadas, como C, não têm essa habilidade. No nível de máquina, lembre-se de que os dados são basicamente não-tipados (Seção 5.1).

Em Java, por exemplo, dada qualquer referência a objeto `obj`, podemos determinar sua classe via:

```
Class c = obj.getClass( );
```

em que `getClass` é um método herdado da classe `Object`.[5]

Um dos usos comuns da RTTI ocorre quando temos que determinar se dois objetos são iguais. Suponha que estamos escrevendo um método desses para a classe `MyObject`, que tem subclasses. O código prototípico para um método `equals` em Java é:

```
1 public boolean equals(Object obj) {
2    if (obj == null) return false;
3    if (this == obj) return true;
4    if (getClass( ) != obj.getClass( )) return false;
5    MyObject other = (MyObject) obj;
6    // test contents of this vs. other
7    ...
8 }
```

A linha 3 é uma otimização que diz que se o receptor da mensagem e o parâmetro se referem ao mesmo objeto, então eles devem ser iguais. A linha 4 garante que o receptor da mensagem e o parâmetro são objetos da mesma classe. Supondo que o parâmetro é um subtipo de `MyObject`, então a linha 4 garante que:

- A conversão na linha 5 não gera uma exceção.
- Que se `x.equals(y)`, então `y.equals(x)`.

13.2.11 Reflexão

Reflexão é um mecanismo pelo qual um programa pode descobrir e usar os métodos de qualquer um de seus objetos e suas classes. A reflexão é essencial para ferramentas de programação que aceitam plug-ins (como Eclipse – **www.eclipse.org**) e para componentes JavaBeans.

Vimos na última seção que RTTI proporciona a habilidade para identificar em tempo de execução a classe de um objeto. Em Java, a classe `Class` proporciona as seguintes informações sobre um objeto:

- A superclasse ou a classe-mãe.
- Os nomes e os tipos de todos os campos.
- Os nomes e as assinaturas de todos os métodos.
- As assinaturas de todos os construtores.
- As interfaces que a classe implementa.

5. Essa discussão leva a admitir que nenhuma superclasse tenha superado o método `getClass`.

```
Class class = obj.getClass( );
Constructor[ ] cons = class.getDeclaredConstructors( );
for (int i=0; i < cons.length; i++) {
    System.out.print(class.getName( ) + "(" );
    Class[ ] param = cons[i].getParameterTypes( );
    for (int j=0; j < param.length; j++) {
        if (j > 0) System.out.print(", ");
        System.out.print(param[j].getName( );
    }
    System.out.println( ")" );
}
```

| Figura 13.15 Reflexão sobre Construtores

Por exemplo, o código dado na Figura 13.15 escreve no console as assinaturas de tipos de todos os construtores de um objeto arbitrário `obj`.

De maneira similar, podemos obter os nomes e as assinaturas de métodos para os métodos de uma classe arbitrária como objetos da classe `Method`. Um dos métodos da classe `Method` é o método `invoke`, que nos permite chamar o método:

```
Method printString = PrintStream.class.getDeclaredMethod(
    "print", new Class[ ] {String.class});
printString(System.out, new Object[ ] { "Tucker and Noonan"});
// same as: System.out.print("Tucker and Noonan");
```

Repare que a primeira instrução apanha o método com a assinatura desejada. A segunda instrução chama o método, no qual o primeiro argumento é um objeto que tem o método desejado e o segundo argumento são parâmetros, passados como um conjunto de `Objects`.

A framework para testes JUnit (**www.junit.org**) localiza dinamicamente todos os métodos em um caso de teste cujo nome começa com `test`. Todos esses métodos são automaticamente executados e seus resultados mostrados (sucesso ou falha) sem que o testador tenha de chamar explicitamente qualquer um deles. A implementação JUnit usa a reflexão.

Nas Seções 13.3–13.5, apresentamos três linguagens orientadas a objetos (Smalltalk, Java e Python, respectivamente), discutimos rapidamente as características de cada linguagem e depois examinamos alguns exemplos interessantes.

13.3 SMALLTALK

Smalltalk (Goldberg e Robson, 1989) foi a linguagem orientada a objetos original, desenvolvida no Xerox PARC (Palo Alto Research Center) durante os anos 70. O sistema Smalltalk era hospedado no Xerox Alto e consistia em um sistema operacional, um ambiente de desenvolvimento integrado, um examinador de classes (class browser) e outras coisas mais, junto com suporte para uma GUI baseada em mouse. Grande parte do sistema Smalltalk foi escrito na própria Smalltalk.

Steve Jobs, após visitar o Xerox PARC, usou a GUI baseada em Smalltalk como base para o projeto das GUIs para os computadores Apple Lisa e Macintosh. De forma semelhante, a Microsoft licenciou essa mesma interface da Xerox como base para o Windows.

Como linguagem, a Smalltalk era baseada na linguagem de simulação Simula 67. A Smalltalk teve um declínio nos anos 90 devido ao desenvolvimento de outras linguagens orientadas a objetos, como a Objective-C (usada no desenvolvimento do computador Next), a Eiffel e a Java. No entanto, o desenvolvimento da Squeak (Guzdial e Rose, 2000), uma implementação de fonte aberta (*open source implementation*), parece ter reanimado o interesse na Smalltalk. Squeak está disponível para várias plataformas. GNU Smalltalk (**www.gnu.org**) também pode ser utilizada.

13.3.1 Características Gerais

A Smalltalk é uma linguagem muito simples, tanto sintática quanto semanticamente. O poder da Smalltalk vem de suas bibliotecas de classes, muitas das quais são escritas na própria Smalltalk. É interessante notar que mesmo as estruturas de controle são objetos em Smalltalk.

Excluindo a parte léxica, a gramática da Smalltalk consiste em 21 produções (três páginas). Compare isso com os números dados na Tabela 2.2. Por exemplo, as regras para a avaliação da sintaxe de expressões de objeto são as seguintes (Guzdial e Rose, 2000):

- O valor de cada variável é um objeto, cada objeto é uma instância de alguma classe.
- Um método é iniciado com o envio de uma mensagem a um objeto. O objeto responde avaliando o método do mesmo nome, se ele tiver um. Caso contrário, a mensagem é enviada para o objeto pai. O processo continua até que seja encontrado o método; caso contrário, é gerado um erro.
- Em ordem de precedência, da mais alta para a mais baixa, há os seguintes tipos de mensagens:
  – Mensagens unárias, como `negated`, por exemplo:
  ```
  x negated.
  ```
  – Mensagens binárias, como + em:
  ```
  x + y.
  ```
  – Mensagens palavra-chave, como `go:` em:
  ```
  Turtle go: length.
  ```
  Na falta de parênteses, o código é avaliado da esquerda para a direita. Os parênteses podem ser usados livremente para melhorar a legibilidade.

Veja alguns exemplos:

- `x + y * z squared` é interpretado como:
  `(x + y) * (z squared)`,
  porque a mensagem unária `squared` tem a precedência mais alta, e então as mensagens e os operadores unários são executados da esquerda para a direita, isto é, primeiro +, depois *.
- `a max: b - c` é interpretado como:
  `a max: (b - c)`
  ou o máximo das grandezas `a` e `b - c`.
- `anArray at: i put: (anArray at: i + 1)`
  manda a mensagem `at:put:` para o objeto `anArray` com os valores
  `(i, anArray at: (i + 1))`.

Se admitirmos que `anArray` é um conjunto (array), ele é interpretado como a atribuição `anArray[i] = anArray[i + 1]`.

Por padrão, Smalltalk usa aritmética fracionária de precisão infinita, assim a expressão `1/3 + 2/6 + 3/9` dá como resultado 1. Isso elimina muitos dos problemas que normalmente são introduzidos pela aritmética do computador:

- As diferenças entre aritmética de inteiros e de ponto flutuante.
- A preocupação sobre se números como 0,2 têm uma representação exata em binário (veja a Seção 5.3).
- Problemas de estouro, como quando se tenta calcular o fatorial de 100.

Smalltalk é uma linguagem dinamicamente tipada, assim o despacho do método virtual requer alguma forma de consulta a uma tabela de busca com base no nome do método; uma implementação assim tão eficiente seria uma tabela *hash*. No entanto, qualquer forma de consulta a um dicionário é consideravelmente mais custosa em tempo de execução do que a implementação de conjunto dada na Seção 13.2.9. Diferentemente das linguagens estaticamente tipadas, chamadas para métodos virtuais não existentes não podem ser detectadas enquanto não forem feitas. Assim como uma linguagem dinamicamente tipada, a Smalltalk suporta identificação de tipo em tempo de execução e reflexão; por exemplo, `1 class` is `Integer`.

Há certo suporte para métodos abstratos que são implementados, gerando uma exceção. No entanto, alguém pode ainda usualmente criar um objeto como uma instância de uma classe abstrata e possivelmente usá-lo sem gerar um erro.

Uma instrução de atribuição é escrita, por exemplo, como:

```
a := b.
```

mas o operador de atribuição usualmente é mostrado como uma seta para a esquerda ←. Smalltalk usa semântica de referência para todos os objetos. Smalltalk requer que as variáveis sejam declaradas antes de serem usadas; variáveis temporárias são declaradas com o uso de:

```
| a b |
```

que declara as variáveis temporárias `a` e `b`. Como Smalltalk é uma linguagem dinamicamente tipada, não são declarados tipos com `a` e `b`.

Aquilo que normalmente consideramos como estruturas de controle são objetos bloco em Smalltalk. Um bloco é representado por colchetes e contém código não-avaliado. Blocos podem ter argumentos. Uma instrução `if` para calcular o máximo de `a` e `b` seria escrita:

```
(a > b) ifTrue: [ max := a ]
     ifFalse: [ max := b ].
```

Devido às regras de precedência, os parênteses que envolvem a condição não são necessários. Os argumentos `ifTrue:` e `ifFalse:` devem ambos ser blocos.

Essa seqüência de código é avaliada da seguinte maneira. Primeiro, é avaliada a expressão `a<b`, resultando em um valor boleano, *true* ou *false*. Esse valor é enviado à mensagem `ifTrue:ifFalse:` com os dois blocos não-avaliados como argumentos. Por exemplo, se o valor *true* for enviado à mensagem, então ele força o primeiro bloco a avaliar a si próprio.[6]

6. É interessante notar que o valor true é um objeto da classe `True`, assim o receptor da mensagem não precisa testar se a condição é verdadeira ou falsa.

De maneira similar, são usados blocos na construção de laços. Considere o seguinte fragmento de código C para calcular a soma de um conjunto de elementos:

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
```

O equivalente Smalltalk[7] é:

```
sum := 0.
1 to: n do: [ :i | sum := sum + (a at: i) ].
```

No entanto, é possível também o seguinte:

```
sum := 0.
a do: [ :x | sum := sum + x ].
```

Em cada caso, o bloco tem um parâmetro (`i` e `x`, respectivamente), que é usado no cálculo subseqüente. O método `do:` sobre o conjunto é análogo a um repetidor Java; ele não é interno, mas escrito inteiramente em Smalltalk. Nesse caso, o conjunto `a` responde à mensagem `do:` iterando sobre si próprio, executando o bloco uma vez para cada valor, efetivamente `a at: i`, no conjunto.

Para investigar melhor os blocos, considere a seguinte versão reescrita do exemplo da soma usando um laço *while*:

```
sum := 0.
i := 1.
[ i <= n ] whileTrue: [ sum := sum + (a at: i).
        i := i + 1].
```

O código parece similar a C, exceto quanto às diferenças de sintaxe. Mas, diferentemente da instrução `if` anterior, dessa vez a condição é escrita como um bloco. Se a condição não fosse escrita como um bloco, seria avaliada somente uma vez, não uma vez por iteração. Uma vez mais, os blocos são internos à Smalltalk, enquanto as estruturas de controle como o laço `while` acima não são. Examinamos uma possível implementação do método `whileTrue` para um bloco após uma breve divagação.

Primeiro, examinamos a natureza dos boleanos em Smalltalk. É interessante notar que a classe `Boolean` tem duas subclasses, `True` e `False`. Os valores *true* e *false* são objetos de suas respectivas classes. Considere a implementação a seguir do método `ifTrue: ifFalse:` para a classe `True` que implementa uma estrutura *if-else*:

```
ifTrue: trueBlock ifFalse: falseBlock
    ^ trueBlock value
```

A primeira linha é o cabeçalho do método, que toma dois argumentos, os blocos *true* e *false* não-avaliados, `trueBlock` e `falseBlock`, respectivamente. Como esse é um método da

7. Conjuntos (arrays) Smalltalk começam no índice um.

classe `True`, ele deverá executar o `trueBlock`. Isso é feito na segunda linha, enviando a `trueBlock` a mensagem `value`, que faz o bloco executar a si próprio uma vez. O método então retorna, indicado pelo circunflexo (^), que é mostrado como uma seta ascendente (↑), independente do que `trueBlock` retorne.

Um uso do método `ifTrue:ifFalse:` ocorreria ao computar o método `max`:

```
max: b
   self > b ifTrue: [ ^self ] ifFalse: [ ^b ]
```

Observe que, apesar de `self` e `b` já terem sido avaliados, no método `ifTrue:ifFalse:` devem ser passados blocos indicados por colchetes, não números. A palavra reservada `self` é o mesmo que `this` em C++ ou Java.

Repare que aquilo que mostramos sintaticamente como a estrutura global de uma definição de classe ou método é de certa forma arbitrária, já que a definição de uma classe é feita por meio do examinador de classes Smalltalk. A forma usada aqui é baseada na versão mostrada (impressa) das definições de classe de um dos sistemas Smalltalk que temos usado.

Outros métodos da classe `True` incluem:

```
ifTrue: aBlock
    ^ aBlock value

ifFalse: aBlock
    ^ nil

ifFalse: falseBlock ifTrue: trueBlock
    ^ trueBlock value
```

em que a palavra reservada `nil` é análoga à palavra reservada `null` em Java. Note que o método `ifFalse:` não faz nada, exceto retornar uma referência vazia (`nil`).

O método `whileTrue:` de um bloco teria essa aparência:

```
whileTrue: aBlock
    (self value) ifFalse: [ ^ nil ].
    aBlock value.
    ^ self whileTrue: aBlock
```

A primeira linha do corpo do método tem o bloco de condição avaliado, e se for falsa, retorna. Caso contrário, ela executa o corpo do bloco uma vez e então chama a si própria recursivamente.

13.3.2 Exemplo: Polinômios

Nesta seção desenvolvemos uma classe para representar polinômios; por exemplo:

$$3x^2 + 5x - 7$$

Para essa aplicação, representaremos um polinômio por um conjunto de seus coeficientes começando com o coeficiente de x^0 e prosseguindo até a potência mais alta com um coeficiente diferente de zero. Então o polinômio acima seria representado pelo conjunto `#(-7 5 3)`.

A classe `Polynomial` (dada na Figura 13.16) é uma subclasse de `Magnitude` e tem uma única variável de instância denominada `coefficient`. Os métodos para criar uma instância

```
Magnitude subclass: #Polynomial
    instanceVariableNames: 'coefficient'
    classVariableNames:: ''
    poolDictionaries: ''

new
    "Unary class constructor: return 0*x^0"
    ^ self new: #( 0 )

new: array
    "Keyword class constructor"
    ^ (super new) init: array

init: array
    "Private: initialize coefficient instance variable"
    coefficient := array deepCopy

degree
    "Highest power with a non-zero coefficient"
    ^ coefficient size - 1

coefficient: power
    "Coefficient of given power"
    (power >= coefficient size) ifTrue: [ ^ 0 ].
    ^ coefficient at: power + 1

asArray
    ^ coefficient deepCopy

= aPoly
    ^ coefficient = aPoly asArray

!= aPoly
    ^ (self = aPoly) not

< aPoly
    "not defined"
    ^ self shouldNotImplement
```

| Figura 13.16 Classe Polinomial em Smalltalk

da classe são: `new` e `new:`. O método de classe `new` cria o polinômio $0x^0$ usando o método `new:`. Este último espera um conjunto como argumento e usa isso como seu conjunto de coeficientes.

Na Figura 13.16, a primeira linha de cada método dá a assinatura do método. As linhas que parecem ser *strings* de aspas duplas são na realidade comentários Smalltalk.

Considere o método `coefficient:`. Se o argumento (`power`) for muito grande, o coeficiente real é zero. Caso contrário, ele retorna o coeficiente real da lista (método `at:`), ajustando o índice `power` para compensar o uso pela Smalltalk de uma indexação baseada em conjunto.

O método `asArray` é interessante sob o aspecto de que ele retorna não a lista dos próprios coeficientes, mas uma cópia da lista. Isso impede que um cliente mude os coeficientes reais.

Observe também que dois polinômios `p` e `q` podem ser comparados quanto à igualdade usando `p = q`. O método funciona por meio da comparação das duas listas de coeficientes quanto à igualdade. Note que não há definição matemática para `p < q`. As demais implementações do método são deixadas como exercício.

13.3.3 Exemplo: Números Complexos

Nesta seção, desenvolvemos uma classe de números complexos com base na noção matemática de números complexos da forma $a + b \cdot i$, na qual $i = \sqrt{-1}$. Não se preocupe se você não se lembra de nada sobre números complexos; esse exemplo é mais sobre como escrever a classe e não sobre matemática.

Nossa primeira observação é que essa classe precisa de duas variáveis de instância, usualmente conhecidas como parte real e parte imaginária, correspondendo a a e b acima, respectivamente. Chamaremos essas variáveis de instância de `realpart` e `imagpart`; todas as variáveis de instância em Smalltalk são privadas.

A classe `Complex` é uma subclasse da classe `Number`. A próxima decisão no projeto é tornar imutáveis os objetos da classe. Os objetos de uma *classe imutável* não podem ser alterados uma vez que tiverem sido criados. Isto é, uma vez que tenhamos criado um número complexo, não podemos alterá-lo, somente podemos criar outros novos. A soma de dois números (por exemplo, $x + y$), mesmo usando uma mensagem de palavra-chave, não deve alterar x, mas, sim, criar um novo número complexo.

A primeira tarefa é escrever os métodos da classe (dados na Figura 13.17). O método usual `new` está aqui, mas efetivamente desabilitado, chamando o método `error:` de `Object`. Assim, não são permitidos números complexos não-inicializados. O segundo método de classe (ou construtor) admite que seja passado um número complexo e faz uma cópia dele. O terceiro método da classe usa o método `new` de `Number` para criar um número complexo e então inicializa-o usando o método de instância `setReal:setImaginary:` de `Complex`. Assim, dados dois números a e b, a maneira de se criar $a + b \cdot i$ é:

```
Complex real: a imaginary: b
```

O método `setReal:setImaginary:` deverá ser declarado privado, porque ele nos permite mudar o valor de um número complexo. No entanto, todos os métodos são públicos; o comentário que diz que ele é privado não o torna privado. Como é típico de um método de inicialização, as duas primeiras instruções são atribuições para as variáveis de instância. Espera-se, mas não se obriga, que esse método somente seja chamado a partir do método de classe `real:imaginary:`.

A Figura 13.17 mostra alguns dos métodos necessários para implementar completamente números complexos. Os métodos acessores `real` e `imaginary` retornam as partes real e imaginária do número complexo.

Observe que soma, subtração etc. podem ser implementadas com o uso dos operadores interfixados comuns. A soma de dois números complexos é definida como:

$$(a + b \cdot i) + (c + d \cdot i) = (a + b) + (c + d) \cdot i$$

O método binário + implementa essa definição construindo um novo número complexo. Observe que de uma maneira muito parecida com o que ocorre no hardware do computador, a subtração de dois números complexos `a - b` é implementada como `a + b negated`, em que `negated` é o operador unário menos.

Para as relações, a igualdade de dois números complexos é definida como:

$$(a + b \cdot i) = (c + d \cdot i) \equiv (a = b) \text{ e } (c = d)$$

```
Object subclass: #Complex
    instanceVariableNames: 'realpart imagpart'
    classVariableNames:: ''
    poolDictionaries: ''

new
    "Unary constructor: invalid"
    ^ self error: 'use real:imaginary:'
new: aComplex
    "Class constructor: aComplex"
    ^ (super new) copy: aComplex
real: r imaginary: i
    "Class constructor"
    ^ (super new) setReal: r setImaginary: i
setReal: r setImaginary: i
    "Private instance method to initialize self"
    realpart := r.
    imagpart := i.
    ^ self
real
    "Return real part"
    ^ realpart
imaginary
    "Return imaginary part"
    ^ imagpart
+ val
    "Return new complex number: self + val"
    ^ Complex real: realpart + val real
      imaginary: imagpart + val imaginary
- val
    "Return new complex number: self - val"
    ^ self + val negated
negated
    "Return new complex number: - self"
    ^ Complex real: realpart negated imaginary: imagpart negated
= val
    "Return: self = val"
    ^ (realpart = val real) & (imagpart = val imaginary)
< val
    "Not mathematically defined"
    ^ self shouldNotImplement
```

| Figura 13.17 Métodos de Números Complexos

```
Object subclass: #Account
    instanceVariableNames: 'bal'
    classVariableNames:: ''
    poolDictionaries: ''
new
    "Constructor"
    ^ (super new) init
init
    "Private instance method"
    bal := 0

balance
    ^ bal

deposit: amount
    (amount < 0) ifTrue: [ ^ self error: 'Invalid amount' ].
    bal := bal + amount

withdraw: amount
    (amount < 0) ifTrue: [ ^ self error: 'Invalid amount' ].
    (bal < amount) ifTrue: [ ^ self error: 'Insufficient funds' ].
    bal := bal - amount
```

| Figura 13.18 Uma Classe *Account* (Conta Bancária)

Isto é, dois números complexos são iguais se as suas partes imaginária e real forem iguais. As relações `<`, `<=`, `>`, `>=` não são definidas para números complexos. A relação `!=` fica como exercício.

13.3.4 Exemplo: Conta Bancária

Nesta seção, desenvolvemos uma classe simples "conta bancária", que tem `Object` como sua classe-mãe. Uma conta bancária tem uma única variável de instância `balance` que contém a quantidade de dinheiro que está na conta. O saldo nunca pode se tornar negativo.

A Figura 13.18 mostra a classe *account*. O método `init` coloca o saldo em zero, enquanto o método `balance` retorna o saldo atual da conta.

Os métodos de palavras-chave `deposit:` e `withdraw:` são usados respectivamente para colocar dinheiro e tirar dinheiro da conta. Ambos os métodos verificam o saldo para certificar que ele não é negativo. O método `withdraw:` verifica se a conta não fica descoberta.

13.4 JAVA

A versão Java 1.0 foi lançada em 1995, e a versão Java 1.5 (agora chamada de Java 5) foi lançada em 2004. A linguagem tem demonstrado ser muito estável, sendo que a maior parte das alterações ocorreu na versão Java 1.1. O mesmo não se pode dizer sobre as alterações nas bibliotecas de classe Java Standard Edition (SE), que têm aumentado sempre de volume em cada nova versão.

Java é uma linguagem mista que contém tipos de dados primitivos, como `int`, e objetos. Java usa semântica de cópia para tipos de dados primitivos e semântica de referência para objetos (veja a Seção 7.4). Todos os métodos devem existir como parte da mesma classe

e são virtuais por padrão. Java (e sua prima, C#) se beneficia não tentando permanecer totalmente compatível retroativamente com C, diferente de C++. Java usa coleta automática de lixo, que simplifica muito a tarefa de gerenciamento de memória para o programador.

Java é uma linguagem orientada a objetos estaticamente tipada, de herança simples. Conforme vimos na Seção 13.2, Java tem suporte direto para:

- Classes internas.
- Visibilidade pública, protegida e privada para variáveis e métodos.
- Classes abstratas.
- Interfaces.
- Modelos.
- Identificação de tipo em tempo de execução.
- Reflexão.

Como Java foi usada como linguagem exemplo da Seção 13.2, não vamos repetir a discussão aqui.

Nesse estágio de sua vida, Java deve ser considerada um sucesso. Duas de suas maiores áreas de aplicação são as de educação em ciência da computação e de programação de servidores web.

Dois exemplos são apresentados nas Seções 13.4.1 e 13.4.2, a seguir. O primeiro exemplo se baseia na discussão de sintaxe abstrata de Clite, fornecendo uma boa ilustração do encapsulamento e da herança. O segundo exemplo mostra a aplicação de uma classe *backtracker* geral para uma solução orientada a objetos do problema das oito rainhas. Ambos os exemplos voltam a ocorrer nos próximos dois capítulos. Outros exemplos de Java podem ser encontrados no Capítulo 16, sobre programação orientada a eventos.

13.4.1 Exemplo: Diferenciação Simbólica

Este exemplo implementa a diferenciação simbólica de fórmulas simples de cálculo. Algumas regras familiares para diferenciação simbólica são dadas na Figura 13.19.

$$\frac{d}{dx}(c) = 0 \qquad c \text{ é uma constante}$$

$$\frac{d}{dx}(x) = 1$$

$$\frac{d}{dx}(u + v) = \frac{du}{dx} + \frac{dv}{dx} \qquad u \text{ e } v \text{ são funções de } x$$

$$\frac{d}{dx}(u - v) = \frac{du}{dx} - \frac{dv}{dx}$$

$$\frac{d}{dx}(uv) = u\frac{dv}{dx} + v\frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{u}{v}\right) = \left(v\frac{du}{dx} - u\frac{dv}{dx}\right) \Big/ v^2$$

| Figura 13.19 Regras de Diferenciação Simbólica

Expression = *Variable* | *Value* | *Binary*
Variable = *String* id
Value = int *value*
Binary = *Add* | *Subtract* | *Multiply* | *Divide*
Add = *Expression* esquerda, direita
Subtract = *Expression* esquerda, direita
Multiply = *Expression* esquerda, direita
Divide = *Expression* esquerda, direita

| Figura 13.20 *Expressões* de Sintaxe Abstrata

Por exemplo, diferenciando a função $2 \cdot x + 1$ com relação a `x` usando essas regras obtém-se:

$$\begin{aligned}\frac{d\,(2 \cdot x + 1)}{dx} &= \frac{d}{dx}(2 \cdot x) + \frac{d}{dx}1 \\ &= 2 \cdot \frac{d}{dx}x + x \cdot \frac{d}{dx}2 + \frac{d}{dx}1 \\ &= 2 \cdot 1 + x \cdot 0 + 0\end{aligned}$$

que simplificando resulta em 2.

Em Java, é conveniente representar expressões usando sintaxe abstrata. Um exemplo é a sintaxe abstrata de um programa Clite discutido na Seção 2.5.3. Aqui, a sintaxe abstrata necessária, resumida na Figura 13.20, é adaptada da sintaxe abstrata de uma expressão Clite. As diferenças primárias com relação a Clite são que há somente um tipo de valor, e as classes `Add`, `Subtract` etc. são subclasses de uma expressão `Binary`.

Como antes, cada classe na sintaxe abstrata torna-se uma classe em Java, com as classes `Expression` e `Binary` declaradas como abstratas. Cada classe concreta tem três métodos:

- Um construtor.
- Um método `diff` que implementa a regra apropriada.
- Um método `toString` para imprimir a expressão original e o resultado.

Nossa implementação é apresentada na Figura 13.21, com os métodos `toString` omitidos. Dos quatro operadores aritméticos, somente a classe `Add` é apresentada; as outras três ficam como exercício.

A primeira regra na Figura 13.19 diz que a diferenciação de uma constante com relação a *x* é 0. Isso é implementado pelo método `diff` da classe `Value`, que retorna zero.

Similarmente, a regra da adição da Figura 13.19 (regra três) diz que, para derivar uma soma, você deve primeiro derivar cada termo e depois somar os resultados. O método `diff` da classe `Add` faz exatamente isso, construindo uma nova expressão (árvore da sintaxe abstrata) para conter o resultado.

Uma aplicação do método `diff` para a expressão $2 \cdot x + 1$ resulta: `(((2) * (1)) + ((x) * (0))) + (0))`. Um formatador melhor, junto a um simplificador algébrico, fica como exercício.

```
public abstract class Expression {
    public abstract Expression diff(Variable x);
}

class Value extends Expression {
    private int value;
    public Value(int v) { value = v; }
    public Expression diff(Variable x) {
        return new Value(0);
    }
}
class Variable extends Expression {
    private String id;
    static final private Value zero = new Value(0);
    static final private Value one = new Value(1);
    public Variable(String s) { id = s; }
    public Expression diff(Variable x) {
        return id.equals(x.id) ? one : zero;
    }
}
abstract class Binary extends Expression {
    protected Expression left, right;
    protected Binary(Expression u, Expression v) {
        left = u; right = v;
    }
}
class Add extends Binary {
    public Add(Expression u, Expression v) {
        super(u, v);
    }
    public Expression diff(Variable x) {
        return new Add(left.diff(x), right.diff(x));
    }
}
```

| Figura 13.21 Diferenciação Simbólica em Java

13.4.2 Exemplo: Backtracking

Neste exemplo nós desenvolvemos uma versão orientada a objetos (Noonan, 2000) do *algoritmo backtracking* de Wirth (Wirth, 1976, p. 137):

> Um campo da programação particularmente intrigante é o assunto da "solução geral de problemas". A tarefa é determinar algoritmos para encontrar soluções para problemas específicos, não obedecendo a uma regra fixada de computação, mas, sim, por tentativa e erro. O padrão comum é decompor o processo de tentativa e erro em tarefas parciais. Freqüentemente, essas tarefas são expressas mais naturalmente em termos recursivos e consistem na exploração de um número finito de subtarefas.

Conforme veremos no Capítulo 15, backtracking é a base para a programação lógica. Ela é usada também em grande variedade de aplicações de algoritmos gráficos e inteligência artificial.

A solução geral que Wirth apresenta é um procedimento recursivo que executa iterações por meio de uma série de testes de movimentos. Cada movimento é testado para ver se ele satisfaz a um critério apropriado de *validez*. Se ele satisfizer, o movimento será registrado, e se o problema ainda não for resolvido, o procedimento chamará a si próprio recursivamente, para tentar o próximo nível; se a chamada recursiva falhar, então o movimento atual será desfeito e será tentado um movimento alternativo. O processo de experimentar movimentos continua até que seja encontrada uma solução completamente bem-sucedida ou todos os movimentos no nível atual serão experimentados sem sucesso.

Uma versão desse algoritmo, orientada a objetos, é apresentada na Figura 13.22. Abstratamente, o método `attempt` implementa o algoritmo geral. O parâmetro formal `level` é usado para manter registro do nível recursivo. A variável local `successful` é usada para manter registro para verificar se foi encontrada uma solução para o problema inteiro; assim, em cada nível recursivo, `successful` é inicialmente definida como falsa. Dentro de um nível, movimentos são gerados sucessivamente, testados e, se forem válidos, registrados; somente então o próximo nível será tentado recursivamente. Movimentos que não resultam em uma solução do problema geral são desfeitos. O processo termina quando é encontrada uma solução ou é esgotada a série de movimentos no nível corrente.

O construtor `Backtrack` toma um único parâmetro `b` que resume os detalhes do problema a ser resolvido. Conforme definido pela interface `Backtracker` na Figura 13.23, esses métodos são:

1. Um repetidor `moves`, que é o conjunto de todas as tentativas possíveis de solução em determinado nível.
2. Uma função boleana `valid`, que determina se o movimento corrente satisfaz o critério de validez.
3. Um método `record`, que registra o movimento corrente no nível recursivo corrente.
4. Um método boleano `done`, que determina se a solução, conforme foi registrada, resolve o problema.
5. Um método `undo`, que desfaz um movimento no nível corrente.

Para uma aplicação de *backtracking*, considere o problema *n-rainhas*, no qual n rainhas mutuamente inimigas devem ser colocadas em um tabuleiro de xadrez $n \times n$ em uma configuração em que nenhuma rainha pode capturar qualquer outra rainha em uma única rodada.[8] Apesar de um tabuleiro de xadrez normalmente ter 8×8 quadrados, consideramos o caso mais geral de $n \times n$, fazendo n ser um parâmetro em tempo de execução que pode ser definido como qualquer valor inteiro maior do que zero. Considere a seguinte posição (arbitrária) para o tabuleiro 4×4 mostrado abaixo:

	0	**1**	**2**	**3**
0	Q			
1				
2		Q		
3				

8. Uma rainha pode legalmente andar qualquer número de quadrados horizontalmente, verticalmente ou em qualquer diagonal em uma única rodada, desde que ela não passe sobre nem cruze sobre outra peça de xadrez no processo. Quando uma peça de xadrez entra em um quadrado ocupado por outra peça, dizemos que ela *captura* aquela peça.

```
import java.util.Iterator;

public class Backtrack {

Backtracker b;

public Backtrack (Backtracker b) { this.b = b; }

public boolean attempt (int level) {
   boolean successful = false;
   Iterator i = b.moves(level);
   while (!successful && i.hasNext()) {
       Object move = i.next();
       if (b.valid(level, move)) {
           b.record(level, move);
           if (b.done(level))
               successful= true;
           else {
               successful = attempt(level+1);
               if (!successful)
                   b.undo(level, move);
               } // if done
           } // if valid
       } // while
   return successful;
} // attempt
}
```

| Figura 13.22 Algoritmo Backtraking Orientado a Objetos

```
import java.util.Iterator;

public abstract interface Backtracker {
   public abstract Iterator moves(int level);
   public abstract boolean valid(int level, Object move);
   public abstract void record(int level, Object move);
   public abstract boolean done(int level);
   public abstract void undo(int level, Object move);
}
```

| Figura 13.23 Interface Backtracker

Suponha que `level` represente o número atual da coluna, que neste exemplo é 2.[9] O conjunto de possíveis `moves` (movimentos) representa todos os números de linha válidos, ou seja, {0...3}. Por exemplo, é fácil ver aqui que nenhuma rainha pode ser validamente colocada na coluna 2. Portanto, a busca por uma solução tem de retornar à coluna 1 e realocar aquela rainha da linha 2 para a linha 3, antes de continuar em frente novamente.

Ao projetar a classe `Queens`, precisamos decidir qual será a representação para o tabuleiro de xadrez. Uma possibilidade é uma matriz quadrada $n \times n$. No entanto, uma pequena investigação revela que essa representação é um tanto incômoda, ao verificar a validade ou a "segurança" de uma nova posição. Em vez disso, usamos um grupo de quatro conjuntos unidimensionais.

O primeiro conjunto, denominado `rowPos` (posição na fileira), usa um número de coluna como índice e indica, para aquela coluna, o número da linha, ou posição, de uma rainha, se tiver uma. Para o exemplo de tabuleiro mostrado acima:

```
rowPos[0] = 0
rowPos[1] = 2
```

Esse conjunto é usado para mostrar uma configuração de tabuleiro bem-sucedida. Define-se uma fileira como "segura" se nenhuma rainha ocupa aquela fileira. Essa informação pode ser representada por um conjunto boleano denominado `row` (fileira), que toma o número de uma fileira como índice e indica se há uma rainha naquela fileira. Para o exemplo de tabuleiro mostrado acima:

```
row[0] = true      row[2] = true
row[1] = false     row[3] = false
```

indica que as fileiras 0 e 2 têm rainhas e as outras fileiras não têm. Observamos em seguida que em uma diagonal sudoeste[10] a soma dos números da fileira e da coluna é constante:

	0	**1**	**2**	**3**
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6

Essa informação é capturada por um conjunto boleano denominado `swDiag`, que toma a soma dos números da fileira e da coluna como índice e indica se uma rainha ocupa ou não aquela diagonal sudoeste. Para um tabuleiro $n \times n$, o conjunto de índices varia de 0 a $2n - 1$. Para nosso tabuleiro definido acima, temos um conjunto de sete entradas `swDiag` no qual:

```
swDiag[0] = true
swDiag[3] = true
```

e todas as outras entradas são `false`, indicando que as diagonais sudoeste 0 e 3 têm rainhas e que as outras diagonais sudoeste não têm.

9. Repare que um esquema de solução equivalente poderia ser feito fileira por fileira em vez de coluna por coluna.

10. Uma diagonal sudoeste é uma diagonal que se estende do vértice superior direito até o vértice inferior esquerdo no tabuleiro.

Finalmente, observamos que em uma diagonal sudeste[11] a diferença entre os números de linha e coluna é constante:

	0	**1**	**2**	**3**
0	0	−1	−2	−3
1	1	0	−1	−2
2	2	1	0	−1
3	3	2	1	0

Essa informação pode ser capturada por um conjunto boleano ($2n-1$) denominado `seDiag`, que toma a diferença entre os números de linha e coluna como índice e indica se uma rainha ocupa aquela diagonal sudeste. Para um tabuleiro $n \times n$, o conjunto de índices varia de $-n + 1...n - 1$; como todos os conjuntos Java são baseados em 0, acrescentamos $n - 1$ à diferença, resultando em um intervalo de $0...2(n - 1)$. Para nosso tabuleiro acima, temos um conjunto de sete entradas `seDiag`, no qual:

```
seDiag[3] = true    (southeast diagonal 0 + 3)
seDiag[4] = true    (southeast diagonal 1 + 3)
```

marca a presença de rainhas nas diagonais sudeste 0 e 1, e as demais entradas são `false`. Isso completa nosso projeto do tabuleiro de xadrez.

Com esse design de tabuleiro, os métodos da classe `Queens` podem ser implementados. Ao construtor é passado o valor n, o tamanho do tabuleiro de xadrez; esse valor é armazenado como sendo a variável `MAXROW` e é usado para alocar os conjuntos citados acima. No entanto, os números reais de linha e coluna usados são baseados em 0 para simplificar seu uso como índices de conjuntos.

A implementação dos métodos de interface `Backtracker` para resolver o problema das oito rainhas é simples:

- Um `move` (movimento) varia de 0 a $n - 1$ para acomodar conjuntos baseados em 0; um teste `move` corresponde a uma posição potencial da linha dentro de uma coluna. Assim, o enumerador `moves` deve simplesmente executar iterações sobre os valores $0...n - 1$.
- O método `done` verifica se o número da coluna (seu parâmetro) mais um é pelo menos `MAXROW`.
- O método `valid` verifica se a linha, a diagonal sudoeste e a diagonal sudeste estão todas vazias (isto é, se todas são `false`).
- O registro de um movimento (o método `record`) é feito marcando-se a linha, a diagonal sudoeste e a diagonal sudeste para o quadrado ocupado (isso é, `true`); desfazer um movimento corresponde a marcar essas três posições como não ocupadas (isto é, `false`). A variável `rowPos` registra, para cada coluna, a posição da linha da rainha que está ocupando aquela coluna, e mais tarde é usada para exibição. É usado um método privado `setPosition` para registrar e desfazer movimentos.

A classe `Queens` completa é mostrada na Figura 13.24. Ela inclui também um método `display` para imprimir a solução. Um programa simples que pode ser usado para executar a

11. Uma diagonal sudeste é aquela que se estende do canto superior esquerdo até o canto inferior direito.

```
import java.io.*;
import java.util.Iterator;

public class Queen implements Backtracker {

int MAXROW = 8;       // tamanho do tabuleiro
boolean swDiag[];
boolean seDiag[];
boolean row[];
int rowPos[];

public Queen(int n) {
    MAXROW = n;
    swDiag = new boolean[2*MAXROW-1];
    seDiag = new boolean[2*MAXROW+1];
    row = new boolean[MAXROW];
    rowPos = new int[MAXROW];
}

private void setPosition(int rowNo, int colNo,
        boolean occupied) {
    row[rowNo] = occupied;
    swDiag[rowNo+colNo] = occupied;
    seDiag[rowNo-colNo+MAXROW-1] = occupied;
}

public boolean valid(int colNo, Object move) {
    int rowNo = ((Integer)move).intValue();
    System.out.println("Try: " + rowNo + ", " + colNo);
    return (!row[rowNo])
        && (!swDiag[rowNo+colNo])
        && (!seDiag[rowNo-colNo+MAXROW-1]);
}

public boolean done(int colNo) {
    return colNo + 1 >= MAXROW;
}

public void record(int colNo, Object move) {
    int rowNo = ((Integer)move).intValue();
    System.out.println("Record: " + rowNo + ", " + colNo);
    rowPos[colNo] = rowNo;
    setPosition(rowNo, colNo, true);
}
```

| Figura 13.24 Programa Orientado a Objetos n-Queens (n-Rainhas)

```
public void undo(int colNo, Object move) {
    int rowNo = ((Integer)move).intValue();
    System.out.println("Undo: " + rowNo + ", " + colNo);
    rowPos[colNo] = 0;
    setPosition(rowNo, colNo, false);
}

public void display( ) {
    for (int i = 0; i < MAXROW; i++) {
        for (int j = 0; j < MAXROW; j++)
            if (rowPos[j] == i)
                System.out.print("Q");
            else
                System.out.print(".");
        System.out.println();
    }
} // display

public Iterator moves(int level) {
    return new QueenIterator( );
}

private class QueenIterator
        implements Iterator {
    int cursor = 0;

    public boolean hasNext( ) {
        return cursor < MAXROW;
    }

    public Object next( ) {
        return new Integer(cursor++);
    }
    public void remove() { }
} // QueenIterator

} //class Queen
```

| Figura 13.24 Programa Orientado a Objetos n-Queens (n-Rainhas) (*continuação*)

simulação para diferentes valores de *n* pode ser encontrado no website do livro. O resultado da execução desse programa com $n = 4$ aparece abaixo:

```
Try: 0, 0                          Record: 1, 2     Try: 3, 1
Record: 0, 0      Try: 0, 3        Record: 3, 1
Try: 0, 1         Try: 1, 3        Try: 0, 2
Try: 1, 1         Try: 2, 3        Record: 0, 2
Try: 2, 1         Try: 3, 3        Try: 0, 3
Record: 2, 1                       Undo: 1, 2       Try: 1, 3
Try: 0, 2         Try: 2, 2        Try: 2, 3
Try: 1, 2         Try: 3, 2        Record: 2, 3
Try: 2, 2         Undo: 3, 1   .  .  Q  .
Try: 3, 2         Undo: 0, 0   Q  .  .  .
Undo: 2, 1        Try: 1, 0       .  .  .  Q
Try: 3, 1         Record: 1, 0    .  Q  .  .
Record: 3, 1      Try: 0, 1
Try: 0, 2         Try: 1, 1
Try: 1, 2         Try: 2, 1
```

Esse resultado mostra a dinâmica do *backtracking* e da busca exaustiva. Por exemplo, no primeiro passo, é feita uma tentativa de posicionar a rainha da primeira coluna em (0,0) que é depois finalmente abandonada no passo 26 após tentativas malsucedidas para localizar a rainha da segunda coluna em (2,1) e (3,1). A relocação da rainha da primeira coluna em (1,0) resulta em uma relocação bem-sucedida da rainha da segunda coluna em (3,1), bem como as outras duas rainhas em (0,2) e (2,3), após várias etapas adicionais.

Soluções para outros problemas de *backtracking*, como *knight's tour, exploração de grafos*, solução de quebra-cabeças, e assim por diante, podem ser produzidas com a implementação dos métodos definidos pela interface `Backtracker` de maneira apropriada. Com essa abordagem, o processo de implementação de determinado algoritmo de *backtracking* se resume em preencher os detalhes dos cinco métodos na interface `Backtracker`.

13.5 PYTHON

Muitos autores (como Lutz e Ascher, 1999, p. 4) consideram Python uma linguagem orientada a objetos. No entanto, nós temos uma visão um tanto mais ampla de que Python é uma linguagem multiparadigma que suporta os seguintes estilos de programação:

- Imperativa
- Orientada a objetos
- Funcional[12]

Nesse sentido, Python está mais próxima de C++ (que também é multiparadigma) do que da Smalltalk.

12. Veja o Capítulo 14.

Python, assim como Perl, é uma linguagem de *scripting* (veja a Seção 12.7) e faz parte do movimento LAMP (Linux, Apache, MySQl, Perl/Python/PHP). As características das linguagens de *scripting* são:

- Elas são usadas, às vezes, para "colar" duas ou mais aplicações juntas, transformando a saída de uma aplicação em uma forma apropriada como entrada de uma outra aplicação.
- Tipicamente elas são interpretadas, e não compiladas.
- Elas são dinamicamente tipadas.

Nesta discussão focalizaremos as características orientadas a objetos de Python. No entanto, para entender os exemplos de programas, primeiro explicamos a sintaxe e as semânticas básicas da linguagem.

13.5.1 Características Gerais

Comparada à linguagem Java, Python tem muito mais tipos internos (ints, floats, inteiros de precisão infinita, números complexos, strings etc.) e estruturas de dados básicas, que incluem:

- Listas ou conjuntos ampliáveis: `[1, [2, "allen"], "bob", 3.54]`;
- Dicionários ou conjuntos associativos, isto é, conjuntos com índices *string*: `{"Tucker": "Bowdoin", "Noonan": "William e Mary"}`;
- Tuplas, isto é, estruturas acessadas por posição, em vez de nome de campo: `(238111, "allen", "bob")`.

Embora algumas dessas estruturas possam ser implementadas como classes, elas não parecem ser classes. Os tipos de dados aritméticos possuem os operadores e as funções usuais, assim como as *strings*. Algumas funções matemáticas, funções *string* e outras funções são fornecidas por intermédio de módulos que têm de ser explicitamente importados.

As *strings* Python são consideradas uma lista de caracteres, assim os operadores lista aplicados a *strings* têm o efeito esperado. Por exemplo, se nós acessarmos determinado índice da *string*, obtemos um caractere, se tomarmos uma fatia de uma *string*, obtemos uma sub*string*.

No entanto, em geral, Python é menos tolerante do que Perl. Não podemos executar operações *string* sobre números nem podemos fazer operações aritméticas sobre *strings* cujos valores se parecem com números. Não podemos indexar além do fim de uma lista. Para aumentar uma *string* ou uma lista, temos de acrescentar ao fim da lista.

As instruções Python são razoavelmente convencionais, exceto quanto à sintaxe das instruções compostas ou blocos. Em Python, o cabeçalho de uma instrução composta ou bloco termina com dois-pontos (`:`) e o corpo é recuado e termina quando termina o recuo. Instruções compostas incluem: definições de classe, definição de método ou função, instruções if e laços. Assim, o problema da ambigüidade `if` de outras linguagens, em que uma instrução parece fazer parte de uma instrução `if`, mas não faz, não pode ocorrer em Python.

Assim como muitas linguagens orientadas a objetos, Python usa semântica de referência. No exemplo a seguir:

```
x = [1, 2, 3]
y = x
x[0] = "abc"
```

a última atribuição muda tanto x quanto y. Observe também que a indexação de lista começa em zero.

Python suporta herança múltipla. Todos os métodos e as variáveis de instância são públicos. Uma classe simples seria declarada assim:

```
class MyClass:
    def set(self, value):
        self.value = value
    def display(self):
        print self.value
```

Como uma instrução composta `class` termina em dois-pontos (`:`), e seu corpo, sendo formado pelos métodos denominados `set` e `display`, é recuado, assim também o corpo de cada método é recuado. `MyClass` não tem nenhuma superclasse. Dos dois métodos, o primeiro tem um parâmetro real e o outro, nenhum, já que o primeiro parâmetro de cada método é o receptor da mensagem. Convencionalmente, o nome `self` é usado para o receptor da mensagem.

O código a seguir declara três objetos e então define a variável de instância `value`:

```
x = MyClass( )
y = MyClass( )
z = MyClass( )
x.set(123);
set(y, “allen”)
z.value = “bob”
```

Em Python, todas as variáveis devem ser definidas antes de serem referenciadas; exceto para os parâmetros, as variáveis são declaradas atribuindo-se a elas um valor. A declaração de uma variável de instância fora de um construtor, como é feita pelo método `set`, é uma decisão de programação questionável, já que uma chamada a método `display` antes de qualquer chamada a método `set` resultará em um erro em tempo de execução.

As classes em Python podem ter construtores, destruidores e variáveis de classe. Todos os métodos de instância são públicos; todas as classes e variáveis de instância são também públicas. Como uma linguagem dinamicamente tipada, Python também suporta identificação de tipo em tempo de execução e reflexão, mas faltam classes abstratas e interfaces.

13.5.2 Exemplo: Polinômios

Nesta seção, desenvolvemos uma classe para representar polinômios, por exemplo:

$$3x^2 + 5x - 7$$

Para essa aplicação, representaremos um polinômio por um conjunto de seus coeficientes, começando com o coeficiente de x^0 e prosseguindo até a potência mais alta com um coeficiente diferente de zero. Assim, o polinômio acima seria representado pelo conjunto `[-7 5 3]`.

A classe `Polynomial` não tem uma superclasse e tem uma única variável de instância denominada `coefficient`. A definição de classe é dada na Figura 13.25.

Um construtor é sempre denominado `__init__` e pode ter tantos argumentos quantos desejarmos; nesse caso, um argumento é passado e usado para inicializar a variável de instância `coefficient`. Aqui, o operador concatenador de lista (+), juntamente a uma lista vazia, é usado para forçar a criação de uma cópia rasa (shallow copy) do argumento. Um destruidor, se fosse utilizado, seria denominado `__del__`.

Uma *string* entre aspas triplas vem imediatamente após cada cabeçalho de método; essas *strings* podem se estender sobre múltiplas linhas. Rigorosamente falando, elas são *strings*, não

```
class Polynomial:
    def __init__(self, coef):
        """constructor"""
        self.coefficient = [ ] + coef

    def degree(self):
        """Highest power with a non-zero coefficient"""
        return len(coefficient)

    def coefficient(self, power):
        """Coefficient of given power"""
        if power > len(coefficient):
            return 0
        return coefficient[power]

    def asList(self):
        """return copy of coefficient"""
        return [ ] + coefficient

    def __eq__(self, aPoly):
        """return self == aPoly"""
        return coefficient == aPoly.asList( )

    def __ne__(self, aPoly):
        """return self <> aPoly.asList( )"""
        return coefficient <> aPoly.asList( )

    def __str__(self):
        """return string representation"""
        r = ""
        p = len(coefficient) + 1
        while p > 0:
            p = p - 1
            if coefficient[p] == 0: continue
            if p < len(coefficient): r = r + "+"
            r = r + str(coefficient[p])
            if p == 0: continue
            r = r + "x"
            if p <= 1: continue
            r = r + "^" + str(p)
        return r
```

| Figura 13.25 Classe Polinomial em Python

comentários. No entanto, o utilitário de documentação Python `pydoc`, análogo ao utilitário `javadoc` de Java, procura essas *strings* e usa-as para gerar documentação. Um comentário comum começa com um sinal de libra (#) e se estende até o fim da linha.

Os métodos `degree` e `coefficient` são similares aos seus equivalentes em Smalltalk (Seção 13.3.2), exceto pelo fato de as listas em Python começarem no índice zero. O método `asList` retorna uma cópia rasa da lista `coefficient`; uma alternativa é importar e usar o método `copy` do módulo `copy`. Essa alternativa seria necessária se tivéssemos que fazer uma `deepcopy`.

Definindo os métodos `__eq__` e `__ne__`, podemos comparar dois polinômios `p` e `q` quanto a sua igualdade, escrevendo `p == q`. O método `__eq__` funciona se compararmos os coeficientes dos dois polinômios quanto à igualdade. Fica como exercício a descoberta de um erro um tanto sutil.

O método `__str__` é análogo ao método `toString` de Java. Ele é chamado sempre que a Python precisa de uma representação *string* de um objeto. O método funciona do índice mais alto da lista até zero. Para o conjunto de coeficientes `[3, 0, 5]`, ele gera a *string*:

```
5x^2+3
```

em lugar da *string* menos clara:

```
5x^2+0x^1+3x^0
```

em que o sinal circunflexo (`^`) é usado para representar o expoente. Repare que o operador que mais foi utilizado aqui é o operador de concatenação de *string*. Repare também que um corpo com bloco de uma linha, nesse caso o corpo da instrução `if`, pode ser escrito na mesma linha do cabeçalho do bloco.

13.5.3 Exemplo: Frações

Nesta seção, desenvolvemos uma classe aritmética fracionária de forma que a expressão `1/3 + 2/6 + 3/9` dá como resultado 1. Em nossa implementação usamos inteiros Python. Uma fração como `2/6` tem um numerador (`2`) e um denominador (`6`). A implementação funciona armazenando frações em uma forma normalizada:

1. No caso de um número negativo, o sinal é armazenado com o numerador. Assim, a fração `1/-3` seria armazenada como `-1/3`.
2. Uma fração é armazenada de forma que o maior fator comum entre o numerador e o denominador seja um. Assim, a fração `2/6` seria armazenada como `1/3`.

A Figura 13.26 dá uma implementação parcial da classe *fraction* (fração). O construtor tem valores padrão para o numerador e o denominador. Esse último é verificado para saber se ele é zero; se for zero, é gerada uma exceção. Caso contrário, a fração é convertida para a forma normalizada conforme foi definido acima. Uma solução melhor garantiria que ambas as partes da fração são inteiros estendidos (do tipo *long*).

Os métodos `numerator` e `denominator` retornam as partes correspondentes da fração.

```
class Fraction:
    def __init__(self, numer=1, denomin=1):
        """constructor: default value is 1/1"""
        if denomin == 0:
            raise ZeroDivisionError, "Divide by 0"
        div = gcd(numer, denomin)
        self.num = abs(numer / div)
        self.denom = abs(denomin / div)
        if numer * denomin < 0: self.num = - self.num

    def numerator(self):
        return self.num

    def denominator(self):
        return self.denom

    def __add__(self, other):
        return Fraction(other.denom * self.num
                        + self.denom * other.num,
                        self.denom + other.denom)

    def __neg__(self):
        return Fraction(- self.num, self.denom)

    def __sub__(self, other):
        return - other + self

    def __eq__(self, other):
        return (self.num == other.num and
                self.denom == other.denom)

    def __str__(self):
        return str(self.num) + "/" + str(self.denom)
```

| Figura 13.26 Classe Fração (Fraction) em Python

A soma de duas frações $\frac{a}{b}$ e $\frac{c}{d}$ é definida como:

$$\frac{a}{b} + \frac{c}{d} = \frac{d \times a + b \times c}{b \times d}$$

O código meramente implementa essa definição (método `__add__`), retornando uma nova fração. A negação unária (método `__neg__`) é implementada negando o numerador.

A subtração pode então ser implementada (método `__sub__`) negando-se o segundo operando e somando-o com o primeiro.

Duas frações $\frac{a}{b}$ e $\frac{c}{d}$ são iguais (método `__eq__`) se:

$a = c$ e $b = d$

Isso é assim porque as frações são armazenadas em forma normalizada. Os demais operadores aritméticos e lógicos ficam como exercícios.

O método final implementado é a função conversão de *string* (método `__str__`) que converte a fração $\frac{a}{b}$ na string `a/b`.

Executando o programa:

```
print Fraction(1, 3) + Fraction(-2, -6) + Fraction(3, 9)
```

é impresso o valor `1/1`, conforme era esperado.

Isso finaliza nosso breve passeio pelas características orientadas a objeto da linguagem Python. Um dos autores deste livro escreve todos os seus programas em Perl ou Python, mas ele prefere Python a Perl.

13.6 RESUMO

Uma linguagem orientada a objetos suporta encapsulamento de um tipo de dado com suas funções e as características de ocultamento de informação da abstração de dados. O mecanismo primário para fazer isso é a classe, que define um tipo. As instâncias de uma classe são conhecidas como *objetos*. As classes proporcionam um mecanismo para definir um tipo de dado abstrato em um programa. Uma vantagem da abordagem orientada a objetos para abstração de dados é que ela fornece um construtor, no qual a memória é alocada para o objeto e as variáveis de instância do objeto podem ser inicializadas.

Em linguagens que suportam destruidores, o destruidor de um objeto é chamado automaticamente quando o objeto é desalojado. Os usos de um destruidor incluem a liberação da memória para objetos que não podem mais ser referenciados, a liberação de recursos e o fechamento de arquivos.

As classes desempenham dois papéis complementares importantes em linguagens orientadas a objetos. Primeiro, elas definem o tipo de um objeto de forma que determinam quais mensagens (chamadas de métodos) são legais e precisamente que método está sendo chamado. Segundo, elas permitem verificação completa de tipo.

O paradigma orientado a objetos suporta reutilização de código por meio da herança. As classes existem, em uma linguagem orientada a objetos, em uma hierarquia de classes. Uma classe pode ser declarada como uma subclasse de outra classe, que é chamada classe-mãe ou superclasse. Dentro dessa hierarquia, cada subclasse é capaz de herdar variáveis e métodos de sua classe-mãe, simplesmente em virtude do fato de que ela é uma subclasse.

Um erro comum cometido pelos principiantes em linguagem orientada a objetos é confundir a relação *é-uma* (*é-um*) com a relação *tem-uma* (*tem-um*). Essa última não identifica uma relação subclasse–superclasse; em vez disso, ela identifica uma classe como uma classe cliente de outra classe, usualmente conhecida como *agregação*.

Em linguagens orientadas a objetos o polimorfismo se refere à ligação tardia de uma chamada a uma ou várias implementações diferentes de um método em uma hierarquia de herança. Tais métodos devem ser métodos de instância.

Identificação de tipo em tempo de execução (run-time type identification – RTTI) é a habilidade da linguagem para identificar em tempo de execução o tipo real ou a classe de um objeto. RTTI é também uma característica necessária para suportar reflexão, que é o mecanismo pelo qual um programa pode descobrir e usar os métodos de qualquer um de seus objetos e de suas classes.

As últimas três seções consideraram a história e as características gerais orientadas a objetos das linguagens Smalltalk, Java e Python. Para cada uma dessas linguagens foram apresentados pelo menos dois exemplos de classes.

EXERCÍCIOS

13.1 Dê um exemplo de uma característica Java ou C++ que promova cada um dos seguintes princípios de projeto orientado a objetos: reutilização de código, segurança de tipo, abstração e encapsulamento. Dê outro exemplo que viole cada um desses princípios.

13.2 Complete a implementação Java das classes `Queue`, `Deque` e `PriorityQueue`, que estão esboçadas na Figura 13.11.

13.3 Esboce uma classe e uma interface Java que captura a idéia específica de herança múltipla mostrada na Figura 13.12.

13.4 As classes Java `Collection` têm algumas idiossincrasias. Observamos uma delas, na qual a classe `Stack` é uma subclasse da classe `Vector`. Além disso, não há nenhuma classe `Queue` ou `PriorityQueue` predefinida em Java, apesar de essas coleções serem ferramentas de modelagem muito importantes na ciência da computação.

(a) Que outros pontos fracos você encontra nas classes Java `Collection`?
(b) Que estruturas de dados similares são predefinidas na biblioteca padrão C++? Essa biblioteca é mais completa do que a biblioteca Java? Explique as diferenças básicas entre as duas.
(c) Execute a mesma comparação da parte (b), mas entre Java e Ada.

13.5 Com base nas definições dadas neste capítulo, faça uma modesta pesquisa para determinar se Ada é ou não uma linguagem orientada a objetos. O que você diz sobre CLOS? Perl? Ruby?

13.6 Implemente os operadores restantes para a classe Smalltalk polinomial.

13.7 Implemente um método polinomial `differentiate` em Smalltalk que cria um novo polinômio que é a diferenciação $\frac{d}{dx}$ do receptor da mensagem.

13.8 Implemente os demais operadores para a classe Smalltalk de números complexos.

13.9 Considerando a implementação completa da sintaxe abstrata Clite, o analisador, o verificador de tipo e as semânticas, identifique oportunidades para o uso de modelos. Reimplemente o interpretador Clite em Java 1.5 de uma forma que aproveite as vantagens dessas oportunidades.

13.10 Complete a implementação orientada a objetos da classe `Expression` e suas subclasses para Clite redefinindo a função de verificação de tipo `V` de forma que ela se torne polimórfica sobre as subclasses de `Expression`. Refaça também todo o analisador e a verificação de tipo para expressões nesse estilo mais orientado a objetos, usando a mesma abordagem utilizada na Seção 13.4.1.

13.11 Usando as alterações que você fez nos Exercícios 13.9 e 13.10, teste o seu código interpretando o seguinte programa Clite (fornecido na distribuição do software):

```
int main() {
    float a, b, c, d, x, y;
    a = 4.0; b = 8.0; c = 3.0;
    x = 1.0;
    y = b*b - 4*a*c;
    while (x*x > y+0.0001 || x*x < y-0.0001 )
        x = (x + y/x)/2.0;
    d = (- b + x) / 2*a;
}
```

O que esse programa calcula (aproximadamente)?

13.12 Considere o operador unário negação (`-`) usado no exemplo acima. Qual seria uma alternativa para definir `Unary` como uma subclasse de `Expression`, como ela é agora implementada no interpretador Clite? Por exemplo, que mudanças de implementação seriam necessárias, se nós apenas acrescentássemos a negação unária como outro operador para a classe `Binary`?

13.13 Usando o exemplo Concordance do website do livro, amplie o programa de forma que ele possa relatar, para cada palavra na lista, sua freqüência (número de ocorrências), além dos números de linha em que ela ocorre.

13.14 Usando o seguinte *test driver*, compile e execute as classes Concordance fornecidas no website do livro.

```
import java.io.*;

class MakeConcordance {
  public static void main (String[] args) {
    Concordance dict = new Concordance( );
    if (args.length < 1) {
    System.err.println("No file name provided: ");
    System.exit(1);
    }
    try {
    BufferedReader in = new BufferedReader(
                    new FileReader(args[0]));
    Document doc = new Document(in, true, dict);
    } catch (Exception e) {
    System.err.println("File open or read error: " + e);
    System.exit(1);
    }
        Report r = new Report(dict, System.out);
  }
}
```

Use um simples arquivo de texto como entrada para confirmar que o programa funciona corretamente, com ou sem a opção de mostrar números de linha duplicados.

13.15 Usando o programa Concordance novamente, execute os experimentos a seguir. Para cada experimento execute a tarefa específica e responda às questões associadas conforme necessário. Para qualquer experimento que falhar (gera um erro de compilação), desfaça a alteração antes de prosseguir com o próximo experimento.

(a) Na classe `Concordance` mude sua superclasse de `HashMap` para `Hashtable`. Depois recompile e execute novamente `Concordance`. Alguma coisa mudou? Por que sim ou por que não? Explique.

(b) Na chamada para `add` na classe `Document`, remova o *cast* do primeiro argumento para `String`, e na declaração de `add` na classe `Concordance`, mude o tipo de `word` de `String` para `Object`. Depois recompile e execute novamente `Concordance`. Alguma coisa mudou? Por que sim ou por que não? Explique.

(c) Na chamada a `add` na classe `Document`, elimine a conversão do valor de `number` para `Integer` (via `new`Integer(number)), e na declaração de `add` mude o tipo do parâmetro `line` de `Integer` para `int`. Depois recompile e execute novamente `Concordance`. Alguma coisa mudou? Por que sim ou por que não? Explique.

(d) Na classe `Report`, mude `f.next()` para `f.next.toString()`, depois recompile e execute novamente `Concordance`. Alguma coisa mudou? Por que sim ou por que não? Explique.

13.16 Implemente o jogo knight's tour (Wirth, 1976, p. 137) em Java usando a classe `Backtrack` e sua interface associada `Backtracker`. Note que o menor tabuleiro de xadrez (maior do que 1) para o qual existe uma solução é 5×5; verifique isso.

É dado um tabuleiro $n \times n$. Um cavaleiro – que pode se mover de acordo com as regras do xadrez – é colocado no tabuleiro com as coordenadas iniciais (x_0, y_0). O problema é achar abrangência do tabuleiro inteiro, se existir uma, isto é, calcular um roteiro de $n^2 - 1$ movimentos de forma que cada quadrado do tabuleiro seja visitado exatamente uma vez.

13.17 Modifique o método `attempt` da classe `Backtrack` de maneira a generalizar todas as soluções. Teste a sua modificação no programa das oito rainhas.

13.18 Considere todas as classes que definem a sintaxe abstrata de Clite.

(a) Encontre uma classe que seja cliente de uma outra classe.

(b) Encontre uma classe que seja uma subclasse de outra classe.

13.19 Acrescente métodos display à sintaxe abstrata Clite de modo que a chamada `p.display()`, na qual `p` é um programa Clite abstrato, mostra todos os elementos do programa.

13.20 As classes em Java 1.4 não são *genéricas* como elas são em Ada, C++ e Java 1.5. Assim, esta última nos permite especificar `Vector<int> v` para definir um vetor de `int`'s. Isso nos permite extrair o i-ésimo valor de `v` e imediatamente executar operações aritméticas com aquele valor sem uma *cast* explícita. Liste rapidamente as vantagens e as desvantagens dos modelos, tanto na programação quanto no projeto da linguagem.

13.21 Considere a possibilidade de projetar uma implementação completa orientada a objetos de Clite, incorporando todas as suas verificações de tipo e suas regras de significado dentro das classes de sua sintaxe abstrata.

(a) Determine quais dessas regras apareceriam com cada classe.

(b) Que simplificações dessas regras se pode conseguir com essa mudança?

(c) Complete a implementação (Trabalho em Equipe).

13.22 Usando a classe `Stack` da Seção 7.5 como guia, acrescente um método `display` e forneça suas pré e pós-condições.

13.23 Reescreva o código para o método `pop` dado na Figura 7.31 de maneira que ele use o método `top`. A pós-condição para `pop` precisa ser alterada? Por que sim ou por que não?

13.24 Implemente os demais operadores para a classe polinomial Python.

13.25 Implemente um método polinomial `differentiate` em Python que crie um novo polinômio que é a diferenciação $\frac{d}{dx}$ do receptor da mensagem.

13.26 Implemente os demais operadores para a classe fraction (fração) em Python.

13.27 A classe `Set` é uma classe abstrata importante em Java. Sua implementação inclui `HashSet` e `SortedSet`. Faça algumas pesquisas para responder às seguintes questões:

(a) Qual é a diferença entre `HashSet` e `HashMap`? Essa primeira poderia ser uma candidata para implementação da classe `Concordance` em lugar da segunda? Por que sim ou por que não?

(b) Que classes comparáveis existem na C++ Standard Library para Java `HashSet` e `HashMap`? Quais são as principais diferenças em seus projetos?

(c) Que classes comparáveis existem em Ada para Java `HashSet` e `HashMap`? Quais são as principais diferenças em seus projetos?

14 Programação Funcional

"É melhor fazer cem funções operarem sobre uma estrutura de dados do que dez funções operarem sobre dez estruturas de dados."

Atribuído a Alan Perlis

VISÃO GERAL DO CAPÍTULO

A programação funcional emergiu como um paradigma distinto no início da década de 1960. Sua criação foi motivada pela necessidade dos pesquisadores no desenvolvimento de inteligência artificial e em seus subcampos – computação simbólica, prova de teoremas, sistemas baseados em regras e processamento de linguagem natural. Essas necessidades não eram particularmente bem atendidas pelas linguagens imperativas da época.

A linguagem funcional original era a Lisp, desenvolvida por John McCarthy (McCarthy, 1960) e descrita no *LISP 1.5 Programmer's Manual* (McCarthy et al., 1965). A descrição é notável tanto pela sua clareza quanto pela sua brevidade; o manual tem apenas 106 páginas!

Ele contém não apenas uma descrição do sistema Lisp, mas também uma definição formal da própria Lisp. Aqui está uma citação dos autores (McCarthy et al., 1965, p. 1):

> A linguagem Lisp serve primariamente para processamento simbólico de dados. Ela tem sido usada para cálculos simbólicos em cálculo diferencial e integral, projeto de circuitos elétricos, lógica matemática, jogos e outros campos da inteligência artificial.

Recorde-se do Capítulo 1, no qual foi dito que a característica essencial da *programação funcional* é que a computação é vista como uma função matemática mapeando entradas a saídas. Diferentemente da programação imperativa, não há uma noção de estado e, portanto, não há necessidade de uma instrução de atribuição. Assim, o efeito de um laço é obtido via repetição, pois não há uma maneira de incrementar ou decrementar o valor de uma variável no estado, já que não há variáveis. Em termos práticos, porém, muitas linguagens funcionais suportam as noções de variável, atribuição e laço. O importante aqui é que esses elementos não fazem parte do modelo "puro" de programação funcional, e, por isso, não vamos destacá-los neste capítulo.

Devido à sua impureza relativa, que explicaremos em breve, a programação funcional é vista por alguns como um paradigma mais confiável para projetos de software do que a programação imperativa. Porém, essa visão é difícil de documentar, pois a maioria das aplicações de inteligência artificial para as quais a programação funcional é usada não é facilmente acessível a soluções no paradigma imperativo, e reciprocamente. Para uma discussão mais cuidadosa dos méritos da programação funcional *versus* programação imperativa, veja Hughes, 1989.

14.1 FUNÇÕES E O CÁLCULO LAMBDA

Uma função matemática típica, como o quadrado de um número, freqüentemente é definida por:

```
Square(n) = n * n
```

Essa definição dá o nome da função, seguido de seus argumentos entre parênteses, seguido por uma expressão que define o significado da função. `Square` é entendida como sendo uma função que mapeia do conjunto de números reais **R** (seu *domínio*) para o conjunto de números reais **R** (seu *intervalo*), ou mais formalmente:

Square: $\mathbf{R} \rightarrow \mathbf{R}$

> **Definição:** Dizemos que uma função é *total* se ela é definida para todos os elementos em seu domínio, e *parcial* em caso contrário.

A função `Square` é total sobre todo o conjunto de números reais.

Em linguagens de programação imperativa, uma variável como `x` representa uma localização na memória. Assim, a instrução:

```
x = x + 1
```

significa literalmente "atualizar o estado do programa somando 1 ao valor armazenado na célula de memória denominada `x` e depois armazenar aquela soma novamente naquela célula

de memória". Assim, o nome `x` é usado para representar um valor (como em `x + 1`), muitas vezes chamado *valor-r* (*r-value*), e um endereço de memória é chamado *valor-l* (*l-value*).[1] (Veja a Seção 7.4.)

Na matemática, as variáveis são, de certa forma, diferentes em suas semânticas: elas sempre significam expressões reais e são imutáveis. Na matemática não há um conceito de "célula de memória" ou de atualização de um valor da memória ou do valor de uma célula de memória. Linguagens de programação funcional denominadas *puras* eliminam a noção de célula de memória de uma variável em favor da noção matemática; isto é, uma variável dá nome a uma expressão imutável, que também elimina o operador de atribuição. Uma linguagem funcional é *pura* se não houver nenhum conceito de um operador de atribuição ou de uma célula de memória; caso contrário, dizemos que ela é *impura*. No entanto, muitas linguagens de programação funcionais retêm alguma forma de operador de atribuição e são, portanto, *impuras.*

Uma conseqüência da falta de variáveis e atribuições baseadas em memória é que a programação funcional não tem nenhuma noção do estado, como foi feito na definição do significado da linguagem imperativa Clite. O valor de uma função como *Square* depende somente dos valores de seus argumentos, e não de qualquer computação prévia ou mesmo da ordem de avaliação de seus argumentos. Essa propriedade de uma linguagem funcional é conhecida como *transparência referencial*. Uma função tem *transparência referencial* se o seu valor depende somente dos valores de seus argumentos.

A base da programação funcional é o *cálculo lambda*, desenvolvido por Church (1941). Uma expressão lambda especifica os parâmetros e a definição de uma função, mas não seu nome. Por exemplo, veja a seguir uma expressão lambda que define a função `square` discutida acima.

$$(\lambda x \cdot x * x)$$

O identificador *x* é um parâmetro usado no corpo (sem nome) da função *x***x*. A aplicação de uma expressão lambda a um valor é representada por:

$$((\lambda x \cdot x * x)2)$$

que dá como resultado 4.

Este exemplo é uma ilustração de um cálculo lambda aplicado. O que Church realmente definiu foi um cálculo lambda *puro* ou *não-interpretado*, da seguinte maneira:

1. Qualquer identificador é uma expressão lambda.
2. Se *M* e *N* forem expressões lambda, então a *aplicação* de *M* a *N*, escrito como (*M N*), é uma expressão lambda.
3. Uma *abstração*, escrita como ($\lambda x \cdot M$), na qual *x* é um identificador e *M* é uma expressão lambda, é também uma expressão lambda.

Um conjunto simples de regras gramaticais BNF para a sintaxe desse cálculo lambda puro pode ser escrito como:

ExpressãoLambda → *variable* | *(M N)* | (λ *variable* · *M)*

M → ExpressãoLambda

N → ExpressãoLambda

1. Os termos valor-r (*r-value*) e valor-l (*l-value*) foram inventados originalmente porque eles se referiam aos valores retornados pelo lado direito e lado esquerdo (*right* e *left*) de uma atribuição, respectivamente.

Alguns exemplos de expressões lambda:

x
$(\lambda x \cdot x)$
$((\lambda x \cdot x)(\lambda y \cdot y))$

Na expressão lambda $(\lambda x \cdot M)$, dizemos que a variável x está *ligada* à subexpressão M. Uma *variável ligada* é uma variável cujo nome é igual ao nome do parâmetro; caso contrário, dizemos que a variável é *livre* (*free*). Dizemos que qualquer variável não-ligada em M é *livre*. Variáveis ligadas são simplesmente marcadores de lugar, assim como parâmetros de função nos paradigmas imperativo e orientado a objetos. Qualquer variável assim pode ser renomeada consistentemente com qualquer variável livre em M sem mudar o sentido da expressão lambda. Formalmente, as variáveis livres em uma expressão lambda podem ser definidas como:

$free(x) = x$
$free(MN) = free(M \cup free(N)$
$free(\lambda x \cdot M) = free(M) - \{x\}$

Uma substituição de uma expressão N por uma variável x em M, escrita como $M[x \leftarrow N]$, é definida da seguinte forma:

1. Se as variáveis livres de N não possuem ocorrências ligadas em M, então o termo $M[x \leftarrow N]$ é formado pela substituição de todas as ocorrências livres de x em M por N.
2. Caso contrário, assuma que a variável y é livre em N e ligada em M. Depois substitua consistentemente as ocorrências de ligações de y em M por uma nova variável, digamos u. Repita essa operação de renomear variáveis ligadas em M até que se aplique a condição do passo 1, depois proceda como no passo 1.

Os exemplos a seguir ilustram o processo de substituição:

$x[x \leftarrow y] = y$
$(xx)[x \leftarrow y] = (yy)$
$(zw)[x \leftarrow y] = (zw)$
$(zx)[x \leftarrow y] = (zy)$
$(\lambda x \cdot (zx))[x \leftarrow y] = (\lambda u \cdot (zu))[x \leftarrow y] = (\lambda u \cdot (zu))$
$(\lambda x \cdot (zx))[y \leftarrow x] = (\lambda u \cdot (zu))[y \leftarrow x] = (\lambda u \cdot (zu))$

O sentido de uma expressão lambda é definido pela seguinte regra de redução:

$((\lambda x \cdot M)N) \Rightarrow M[x \leftarrow N]$

Isso é chamado *redução-beta* e pode ser lido como "sempre que tivermos uma expressão lambda da forma $((\lambda x \cdot M)N)$, podemos simplificá-la pela substituição $M[x \leftarrow N]$". Uma redução-beta, portanto, representa uma aplicação singular de função.

Uma *avaliação* de uma expressão lambda é uma seqüência $P \Rightarrow Q \Rightarrow R \Rightarrow \cdots$ na qual cada expressão na seqüência é obtida pela aplicação de uma redução-beta à expressão anterior. Se P é uma expressão lambda, então uma *redux* de P é qualquer subexpressão obtida por uma redução-beta. Uma expressão lambda que não contém uma função de aplicação é chamada *forma normal.*

Um exemplo de avaliação é:

$$((\lambda y \cdot ((\lambda x \cdot xyz)a))b) \Rightarrow ((\lambda y \cdot ayz)b) \Rightarrow abz$$

Neste exemplo, nós avaliamos a expressão mais interna λ primeiro; poderíamos ter feito com a mesma facilidade a redução mais externa primeiro:

$$((\lambda y \cdot ((\lambda x \cdot xyz)a))b) = ((\lambda x \cdot xbz)a) = abz$$

A igualdade das expressões lambda é chamada *igualdade-beta* por razões históricas e também porque o termo sugere a redutibilidade beta de uma expressão para outra. Informalmente, se duas expressões lambda M e N forem iguais, escritas como $M = N$, então M e N podem ser reduzidas à mesma expressão até renomear suas variáveis. Igualdade-beta trata da aplicação de uma abstração $(\lambda x \cdot M)$ para um argumento N, e assim proporciona um modelo fundamental para as noções de chamada de função e passagem de parâmetro em linguagens de programação.

Uma linguagem de programação funcional é essencialmente um cálculo lambda aplicado com valores constantes e funções embutidas. A expressão lambda pura (xyx) pode facilmente ser escrita também assim: $(x \times x)$ ou $(x * x)$. Além do mais, essa última forma pode ser escrita em um estilo prefixo $(* \ x \ x)$. Quando somamos constantes como números com sua interpretação usual e suas definições para funções, como *, obtemos um cálculo lambda aplicado. Por exemplo, $(*2 \ x)$ é uma expressão em um cálculo lambda aplicado. Conforme veremos, Lisp/Scheme e Haskell puras são exemplos de cálculos lambda aplicados.

Uma distinção importante em linguagens funcionais é feita usualmente na forma como elas definem avaliação de função. Em linguagens como Scheme, todos os argumentos para uma função são normalmente avaliados no instante da chamada. Isso usualmente é chamado *avaliação rápida*, ou *chamada por valor* (conforme discutimos no Capítulo 9). *Avaliação rápida* em linguagens funcionais refere-se à estratégia de avaliar todos os argumentos para uma função no instante da chamada. Com a avaliação rápida, funções como `if` e `and` não podem ser definidas sem erros potenciais em tempo de execução, como na função Scheme

```
(if (= x 0) 1 (/ 1 x))
```

que define o valor da função como sendo 1 quando `x` é zero e `1/x` em caso contrário. Se todos os argumentos para a função `if` forem avaliados no instante da chamada, a divisão por zero não pode ser evitada.

> **Definição:** Uma alternativa para a estratégia da avaliação rápida é chamada *avaliação lenta*, na qual um argumento para uma função não é avaliado (ela é adiada) até que ele seja necessário.

Como um mecanismo de passagem de argumento, a avaliação lenta é similar (mas não idêntica) à chamada por nome, e é o mecanismo-padrão na linguagem Haskell. Scheme também tem mecanismos que permitem ao programador especificar o uso de avaliação lenta, mas não exploraremos esses mecanismos neste capítulo.

Uma vantagem da avaliação rápida é a eficiência, em que cada argumento passado a uma função é avaliado apenas uma vez, enquanto na avaliação lenta um argumento para uma função é reavaliado cada vez que ele é usado, e isso pode ocorrer mais de uma vez. Uma vantagem da avaliação lenta é que ela permite certas funções interessantes a serem definidas, de modo que não podem ser implementadas em linguagens rápidas.

Mesmo a definição anterior da função `if` torna-se isenta de erro com uma estratégia de avaliação lenta, já que a divisão `(/ 1 x)` só ocorrerá quando $x \neq 0$.

Em cálculo lambda puro, a aplicação da função:

$$((\lambda x \cdot *x\ x)5) = (*\ 5\ 5)$$

não dá qualquer interpretação para o símbolo 5 ou o símbolo *. Somente em um cálculo lambda aplicado conseguiríamos uma redução maior para o número 25.

Um aspecto importante da programação funcional é que as funções são tratadas como valores de primeira classe. Um nome de uma função pode ser passado como um parâmetro, e uma função pode retornar outra função como valor. Uma função dessas às vezes é chamada *forma funcional.* Um exemplo de uma forma funcional seria uma função `g` que toma como parâmetro uma função e uma lista (ou uma seqüência de valores) e aplica a função dada a cada elemento na lista, retornando uma lista. Usando `Square` como exemplo, então:

```
g(f, [x1, x2, ...]) = [f(x1), f(x2), ...]
```

torna-se

```
g(Square, [2, 3, 5]) = [4, 9, 25]
```

Nas Seções 14.2 e 14.3, exploramos o uso dessas e de muitas outras formas funcionais úteis.

14.2 SCHEME

Como linguagem original de programação funcional, Lisp tem muitas características que foram transportadas para linguagens posteriores, portanto, Lisp proporciona uma boa base para estudar outras linguagens funcionais. Com o passar dos anos, foram desenvolvidas muitas variantes de Lisp; hoje, somente duas variantes principais permanecem em uso difundido: Common Lisp (Steele, 1990) e Scheme (Kelsey et al., 1998) (Dybvig, 1996). Como linguagens que tentam unificar um conjunto de variantes, tanto Scheme quanto Common Lisp contêm um conjunto de funções equivalentes. Este capítulo apresenta um subconjunto puramente funcional de Scheme.

Quando vista como linguagem funcional pura, nosso subconjunto[2] Scheme não tem nenhuma instrução de atribuição. Em vez disso, os programas são escritos como funções (repetitivas) sobre valores de entrada que produzem valores de saída; os próprios valores de entrada não são alterados. Nesse sentido, a notação Scheme está muito mais próxima da matemática do que estão as linguagens de programação imperativa e orientada a objetos como C e Java.

Sem uma instrução de atribuição, nosso subconjunto Scheme faz uso intensivo das funções repetitivas para repetição, em lugar da instrução *while* que encontramos nas linguagens imperativas. Apesar da ausência das instruções *while*, pode-se provar que esse subconjunto é *Turing completo*, o que significa que qualquer função computável pode ser implementada naquele subconjunto. Isto é, uma linguagem funcional é Turing completo porque ela tem valores inteiros e operações, uma maneira de definir novas funções usando funções existentes, condicionais (instruções *if*) e repetição.

2. A linguagem Scheme completa tem uma instrução de atribuição chamada `set!`, que evitaremos usar nessa discussão.

Uma prova de que essa definição de Turing completo é equivalente àquela do Capítulo 12 para linguagens imperativas está além do escopo deste livro. No entanto, o uso de Scheme para implementar a semântica denotacional de Clite dada neste capítulo deve proporcionar uma evidência convincente, embora informal, de que uma linguagem puramente funcional é pelo menos tão poderosa quanto uma linguagem imperativa. O inverso também é verdadeiro, pois interpretadores Scheme e Lisp são implementados em máquinas von Neumann, que são a base para o paradigma imperativo.

14.2.1 Expressões

Expressões em Scheme são construídas em notação de prefixo de Cambridge, na qual as expressões ficam entre parênteses e o operador ou a função precede seus operandos, como no exemplo:

```
(+ 2 2)
```

Se essa expressão for apresentada a um interpretador Scheme, ele retorna o valor 4.

Uma vantagem dessa notação é que ela permite que operadores aritméticos como + e * tomem um número arbitrário de operandos:

```
(+)                     ; evaluates to 0
(+ 5)                   ; evaluates to 5
(+ 5 4 3 2 1)           ; evaluates to 15
(*)                     ; evaluates to 1
(* 5)                   ; evaluates to 5
(* 1 2 3 4 5)           ; evaluates to 120
```

Observe que ponto-e-vírgula em Scheme inicia um comentário, que continua até o fim da linha. Essas expressões aritméticas são exemplos de listas Scheme; dados e programas (funções) são representados por listas. Quando uma lista Scheme é interpretada por uma função, o operador ou o nome da função vem após o parêntese esquerdo e os demais números são seus operandos. Expressões mais complicadas podem ser construídas usando aninhamento:

```
(+  (*  5  4) (-  6  2))
```

que é equivalente a $5 * 4 + (6 - 2)$ em notação interfixada, e é avaliado como 24.

Variáveis globais são definidas em Scheme pelo uso da função `define`. Para definir uma variável `f` igual a 120 colocaríamos a expressão:

```
(define  f  120)
```

A função `define` é a única das que examinaremos que muda seu ambiente, em vez de simplesmente retornar um valor. No entanto, não trataremos `define` como uma instrução de atribuição em nosso subconjunto Scheme;[3] nós só a usamos para introduzir um nome global para um valor, como ocorre na matemática.

3. A função `set!` é uma verdadeira atribuição em Scheme porque ela pode ser usada para mudar o valor de uma variável existente. Muitos textos Scheme usam a função `set!` em um nível global como equivalente para `define`.

14.2.2 Avaliação de Expressões

Para entender como Scheme avalia as expressões, são aplicadas três regras principais.

Primeira, nomes ou símbolos são substituídos por suas ligações atuais. Supondo a definição da variável `f` da seção anterior:

```
f                          ; evaluates to 120
( + f 5)                   ; evaluates to 125
                           ; using the bindings for +, f
```

Esse uso de `f` é um exemplo da primeira regra.

A segunda regra é que as listas são avaliadas como chamadas de função escritas em notação de prefixo de Cambridge:

```
(+)                        ; calls + with no arguments
(+ 5)                      ; calls + with 1 argument
(+ 5 4 3 2 1)              ; calls + with 5 arguments
(+ (5 4 3 2 1))            ; error, tries to evaluate 5 as
                           ; a function
(f)                        ; error; f evaluates to 120, not
                           ; a function
```

A terceira regra é que constantes avaliam a si próprias:

```
5                          ; evaluates to 5
#f                         ; is false, predefined
#t                         ; is true, predefined
```

Pode-se impedir que um símbolo ou uma lista seja avaliado usando a função `quote (bloqueador de avaliação)` ou o apóstrofo (’), como em:

```
(define colors  (quote (red yellow green)))
(define colors '(red yellow green))
```

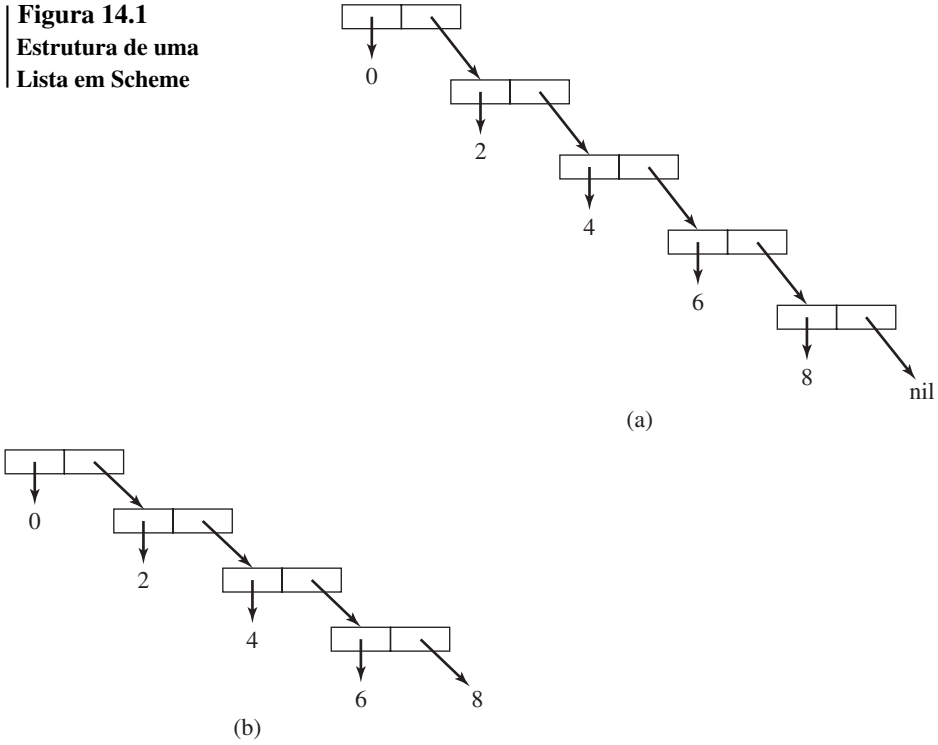
Observe que não há um apóstrofo de fechamento; o apóstrofo bloqueia o símbolo ou a lista vem imediatamente após ele. Você pode também bloquear símbolos:

```
(define x f)               ; defines x to be 120 (value of f)
(define x 'f)              ; defines x to be the symbol f
(define acolor 'red)       ; defines acolor to be red
(define acolor red)        ; an error, symbol red not defined
```

14.2.3 Listas

Conforme já vimos, a estrutura de dados fundamental de Scheme é a lista; ela é usada para comandos e dados. Já vimos muitos exemplos de listas de constantes. Nesta seção vamos ver como colocamos coisas nas listas e como as recuperamos depois.

Figura 14.1
Estrutura de uma Lista em Scheme



Primeiro, vamos definir algumas listas de números:

```
(define  evens '(0  2  4  6  8))
(define  odds '(1  3  5  7  9))
```

Graficamente, a lista `evens` seria representada conforme mostra a Figura 14.1a.

O símbolo `()` representa uma lista vazia; por convenção, as listas Scheme usualmente terminam com `()`.[4] Em termos de lista "linkada" de forma imperativa, o símbolo `()` pode ser visto como um ponteiro `null`. Se o símbolo `()` estiver faltando no fim da lista, Scheme o mostraria como:

```
(0 2 4 6 . 8)
```

Esse tipo de estrutura pode ser gerado por algumas das funções Scheme discutidas a seguir. A Figura 14.1 mostra a diferença em representação de memória entre uma lista que tenha `()` como seu último elemento (Figura 14.1a) e outra que não tenha (Figura 14.1b).

4. O valor `()` é implementado pela null reference, conforme discutido anteriormente no Capítulo 5.

A função básica usada para construir uma lista é `cons`, que toma dois argumentos, o segundo dos quais deverá ser uma lista:

```
(cons 8 ())                      ; gives (8)
(cons 6 (cons 8 ()))             ; gives (6 8)
(cons 4 (cons 6 (cons 8 ())))    ; gives (4 6 8)
(cons 4 (cons 6 (cons 8 9)))     ; gives (4 6 8 . 9)
```

Observe que o último exemplo cria uma lista irregular, já que o segundo argumento do `cons` mais interno não é uma lista.

Um nó em uma lista Scheme tem duas partes, o primeiro elemento ou o início da lista, e os demais elementos da lista ou final. A função `car` retorna o início da lista, enquanto a função `cdr` retorna o final.[5] Referindo-nos novamente à lista `evens` representada na Figura 14.1, os exemplos a seguir ilustram essas funções:

```
(car evens)              ; gives 0
(cdr evens)              ; gives (2 4 6 8)
(car (cdr evens))        ; gives 2
(cadr evens)             ; gives 2
(cdr(cdr evens))         ; gives (4 6 8)
(cddr evens)             ; gives (4 6 8)
(car '(6 8))             ; gives 6
(car (cons 6 8))         ; gives 6
(cdr '(6 8))             ; gives (8)
(cdr (cons 6 8))         ; gives 8, not (8)
(car '(8))               ; gives 8
(cdr '(8))               ; gives ()
```

Observe que Scheme permite que seqüências de `car`'s e `cdr`'s (até cinco) sejam abreviadas incluindo somente a letra do meio; assim, `cadr` é um `car` de um `cdr`, `cddr` é um `cdr` de um `cdr`, e assim por diante.

Funções de nível mais alto para juntar listas incluem as funções `list` e `append`. A função `list` toma um número variável de argumentos e constrói uma lista que consiste naqueles argumentos:

```
(list 1 2 3 4)                   ; gives (1 2 3 4)
(list '(1 2) '(3 4) 5)           ; gives ((1 2) (3 4) 5)
(list evens odds)                ; gives ((0 2 4 6 8) (1 3 5 7 9))
(list 'evens 'odds)              ; gives (evens odds)
```

5. Os termos `car` e `cdr` são vestígios dos primeiros tempos, quando Lisp foi implementada no IBM 704. Os registradores de endereço daquela máquina tinham duas partes, a parte do *address* e a parte do *decrement*. Assim, os termos originais Lisp `car` e `cdr` eram abreviaturas para "conteúdo da parte endereço do registrador" e "conteúdo da parte decremento do registrador", respectivamente. Embora a máquina 704 tenha caído na obscuridade, essas duas abreviaturas permanecem.

Ao contrário, a função `append` toma dois argumentos – e ambos os argumentos serão listas –, e ocorre um encadeamento da segunda lista ao fim da primeira lista:

```
(append '(1 2) '(3 4))     ; gives (1 2 3 4)
(append evens odds)        ; gives (0 2 4 6 8 1 3 5 7 9)
(append '(1 2) '())        ; gives (1 2)
(append '(1 2) (list 3))   ; gives (1 2 3)
```

Para acrescentar um único número ao início da lista de números pares, usaríamos a função `cons`:

```
(cons 10 evens)          ; gives (10 0 2 4 6 8)
```

A lista vazia `()` no fim da lista é importante; ao percorrer repetidamente uma lista, verificamos sempre se há uma lista vazia no fim.

Como Scheme foi projetada para processar listas, ela contém um conjunto de funções especiais de processamento de lista. Devido à necessidade freqüente, há uma função especial denominada `null?` para testar quanto à existência de uma lista vazia:

```
(null?  '())             ; returns #t
(null?  evens)           ; returns #f
(null?  '(1 2 3))        ; returns #f
(null?  5)               ; returns #f
```

Scheme contém um conjunto de funções para testar se um objeto é igual ou equivalente a outro. Essas funções são `equal?`, = e `eqv?`. Em lugar de relacionar suas diferenças, vamos nos basear na função `equal?`, que é razoavelmente geral. Essa função retorna *true* (`#t`) se os dois objetos tiverem a mesma estrutura e mesmo conteúdo; caso contrário, ela retorna *false*:

```
(equal? 5 5)                    ; returns #t
(equal? 5 1)                    ; returns #f
(equal? '(1 2) '(1 2))          ; returns #t
(equal? 5 '(5))                 ; returns #f
(equal? '(1 2 3) '(1 (2 3)))    ; returns #f
(equal? '(1 2) '(2 1))          ; returns #f
(equal? '() '())                ; returns #t
```

14.2.4 Valores Elementares

Até aqui, todos os valores elementares que vimos em Scheme são números ou símbolos (nomes). Há realmente alguns tipos diferentes de números, incluindo inteiros, racionais e de ponto flutuante. Outros tipos elementares de valores Scheme incluem caracteres, funções, símbolos e *strings*. Cada um desses tipos pode ser caracterizado usando um qualificativo apropriado; por exemplo, todos os tipos a seguir retornam `#t`:

```
(pair? evens)
(list? evens)
(symbol? 'evens)
(number? 3)
```

Além dos tipos de valores listados, há também valores boleanos : `#t` que significam *true* e valores `#f` que significam *false*. Todos os valores exceto `#f` e a lista vazia `()` são interpretados como `#t` quando usados como um atributo.

14.2.5 Fluxo de Controle

Os dois construtores de controle de fluxo que usaremos são o controle `if` e o controle `case`.

A função `if` vem em duas versões usuais: a versão if-then e a versão if-then-else. Abstratamente, isso aparece como:

```
(if  test  then-part)
(if  test  then-part  else-part)
```

Veja um exemplo de cada:

```
(if (<  x  0) (- 0 x))
(if (<  x  y) x y)
```

A primeira função `if` retorna o negativo de `x`, se `x` for menor que 0. A segunda função retorna o menor dentre os dois valores de `x` e `y`.

A função `case` é muito semelhante à case em Ada e à switch em Java; a função case tem um opcional `else` que, caso esteja presente, deve ser o último case. Um case simples para calcular o número de dias em um mês (ignorando os anos bissextos) é:

```
(case month
      ((sep apr jun nov) 30)
      ((feb) 28)
      (else  31)
)
```

Note que cada case específico tem uma lista de constantes, exceto `else`.

14.2.6 Definindo Funções

As funções Scheme são definidas pelo uso de `define`, que tem a seguinte forma geral:

```
(define  name  (lambda ( arguments )  function-body ))
```

Assim, uma função mínima pode ser definida por:

```
(define min (lambda (x y) (if (< x y) x y)))
```

Em outras palavras, Scheme (assim como sua antecessora Lisp) é um cálculo lambda aplicado com a habilidade de dar nomes a definições lambda específicas.

Como os programadores logo se cansam de colocar sempre a palavra `lambda` e o conjunto extra de parênteses, Scheme fornece uma maneira alternativa de escrever a função `define`:

```
(define ( name arguments )  function-body )
```

Usando essa alternativa, a função mínima pode ser escrita de forma mais simples:

```
(define (min x y) (if (< x y) x y))
```

A função a seguir calcula o valor absoluto de um número:

```
(define (abs x) (if (< x 0) (- 0 x) x))
```

Ocorrem funções mais interessantes quando usamos a repetição para definir uma função em termos dela própria. Um exemplo da matemática é a venerável função fatorial, que pode ser definida em Scheme da seguinte maneira:

```
(define (factorial n)
   (if (< n 1) 1 (* n (factorial (- n 1)))
))
```

Por exemplo, a aplicação da função (`factorial 4`) resulta em 24, se usarmos a definição acima da seguinte maneira:

```
(factorial 4) = (* 4 (factorial 3))
              = (* 4 (* 3 (factorial 2)))
              = (* 4 (* 3 (* 2 (factorial 1))))
              = (* 4 (* 3 (* 2 (* 1 (factorial 0)))))
              = (* 4 (* 3 (* 2 (* 1 1))))
              = (* 4 (* 3 (* 2 1)))
              = (* 4 (* 3 2))
              = (* 4 6)
              = 24
```

Mesmo tarefas iterativas simples como somar uma lista de números são executadas de forma recursiva em Scheme. Aqui, assumimos que os números formam uma lista `(a1 a2...an)` cujos elementos devem ser somados:

```
(define (sum alist)
   (if (null? alist) 0
       (+ (car alist) (sum (cdr alist)))
))
```

Observe que essa função de soma é similar em efeito à função soma interna de Scheme (+).

A função `sum` exibe um padrão comum a uma variedade de funções Scheme. A função é invocada por meio da lista após verificar primeiro se a lista está vazia. Esse teste geralmente é chamado *base case* em definições recursivas. O *passo recursivo* prossegue somando o primeiro elemento da lista à soma do restante da lista (a aplicação recursiva da função). É garantido que a função termina porque, em cada passo repetitivo do cálculo, o argumento da lista se torna menor tomando o `cdr` da lista.

Outras funções Scheme interessantes manipulam listas de símbolos, em lugar de números. Todas as funções apresentadas a seguir já estão definidas em Scheme; nós as mostramos aqui porque elas representam padrões recorrentes em programação Scheme.

O primeiro exemplo calcula o tamanho de uma lista, isto é, quantos elementos ela contém, contando sublistas como elementos simples. Veja alguns exemplos do uso da função `length`:

```
(length '(1 2 3 4))            ; returns 4
(length '((1 2) 3 (4 5 6)))    ; returns 3
(length '())                    ; returns 0
(length 5)                      ; error
```

A definição da função `length` segue de perto o padrão definido pela função `sum` com apenas algumas pequenas diferenças:

```
(define (length alist)
    (if (null? alist) 0 (+ 1 (length (cdr alist))))
))
```

A aplicação da função `(length '((1 2) 3 (4 5)))` resulta em:

```
(length '((1 2) 3 (4 5)))
    = (+ 1 (length '(3 (4 5)))
    = (+ 1 (+ 1 (length '((4 5)))))
    = (+ 1 (+ 1 (+ 1 (length ()))))
    = (+ 1 (+ 1 (+ 1 0)))
    = (+ 1 (+ 1 1))
    = (+ 1 2)
    = 3
```

Outra função comum interna é a função `member`, que testa para saber se um elemento `elt` (que pode ser uma lista) ocorre como um membro de determinada lista. Se não, a função retorna `#f`. Caso contrário, ela retorna o restante da lista, começando no elemento encontrado, que pode ser interpretado como `#t`. Alguns exemplos do uso da função `member`:

```
(member 4 evens)                ; returns (4 6 8)
(member 1 evens)                ; returns #f
(member 2 '((1 2) 3 (4 5)))     ; returns #f
(member '(3 4) '(1 2 (3 4) 5))  ; returns ((3 4) 5)
```

A função `member` é mais comumente usada como um atributo. Uma vez mais, a definição começa com o *base case*, ou seja, uma lista vazia, no qual o *case* `member` retorna a lista vazia. Caso contrário, ela testa para saber se o cabeçalho da lista é igual ao elemento procurado; se for, ela retorna a lista e, em caso contrário, ela se repete no final da lista.

```
(define (member elt alist)
   (if (null? alist) '()
       (if (equal? elt (car alist)) alist
           (member elt (cdr alist))
)))
```

Nossa última função simples é `subst`, que é usada para substituir seu primeiro argumento para todas as ocorrências de seu segundo argumento em uma lista (seu terceiro argumento). Assim como a função membro, a verificação quanto a objetos iguais só é feita no nível mais alto da lista:

```
(subst 'x 2 '(1 2 3 2 1))            ; returns (1 x 3 x 1)
(subst 'x 2 '(1 (2 3) 2 1))          ; returns (1 (2 3) x 1)
(subst 'x 2 '(1 (2 3) (2)))          ; returns (1 (2 3) (2))
(subst 'x '(2 3) '(1 (2 3) 2 3))     ; returns (1 x 2 3)
(subst '(2 3) 'x '(x o x o))         ; returns ((2 3) o (2 3) o)
```

Essa função é interessante porque ela deve construir a lista de saída como resultado da função, em vez de simplesmente retornar um de seus argumentos. Caso contrário, essa função é similar em estrutura à função `member`:

```
(define (subst y x alist)
   (if (null? alist) '()
       (if (equal? x (car alist))
           (cons y (subst y x (cdr alist)))
           (cons (car alist) (subst y x (cdr alist)))
)))
```

14.2.7 Expressões Let

Muitas vezes, ao se definir uma função, pode ocorrer uma subexpressão várias vezes. Scheme segue a convenção matemática de permitir a introdução de um nome para uma subexpressão. Isso é feito por meio da função `let`, que tem a seguinte forma geral:

```
(let (( var1  expr1 ) ( var2 expr2 ) ... ) body )
```

O uso criterioso de uma função `let` pode melhorar a legibilidade de uma definição de função. Um efeito colateral do uso de `let` é que Scheme só avalia a expressão uma vez, em vez de fazê-lo todas as vezes em que ela ocorre na definição. Veja a seguir um exemplo do uso de uma função `let`:

```
(let ((x 2) (y 3)) (+ x y))          ; returns 5
(let ((plus +) (x 2)) (plus x 3))    ; returns 5
```

Um uso mais interessante da função `let` ocorre quando ela aparece dentro da definição de uma função:

```
(define (subst y x alist)
   (if (null? alist) '()
       (let ((head (car alist)) (tail (cdr alist)))
           (if (equal? x head)
               (cons y (subst y x tail))
               (cons head (subst y x tail))
))))
```

Como na matemática, uma função `let` meramente introduz um conjunto de nomes para expressões comuns. Os nomes são ligados aos seus valores somente no corpo da função `let`. Um exemplo mais amplo da função `let` ocorre na Seção 14.2.7.[6]

Em todos os exemplos anteriores, os argumentos para uma função são átomos ou listas. Scheme também permite que uma função seja um parâmetro formal para outra função que está sendo definida:

```
(define (mapcar fun alist)
   (if (null? alist) '()
        (cons (fun (car alist)) (mapcar fun (cdr alist)))
))
```

A função `mapcar` tem dois parâmetros, uma função `fun` e uma lista `alist`; ela aplica a função `fun` a cada elemento de uma lista, criando uma lista a partir dos resultados. Como exemplo, considere uma função de um argumento que eleva o argumento ao quadrado:

```
(define (square x) (* x x))
```

A função `square` pode ser usada com `mapcar` em qualquer uma das duas variações a seguir para elevar ao quadrado todos os elementos de uma lista:

```
(mapcar square '(2 3 5 7 9))
(mapcar (lambda (x) (* x x)) '(2 3 5 7 9))
```

Na primeira variação, o primeiro parâmetro é uma função definida. Na segunda variação, é passada uma função sem nome com o uso da lambda *notation* para defini-la. Ambas as variações produzem o resultado `(4 9 25 49 81)`.[7]

Essa facilidade para definir formas funcionais permite às linguagens funcionais como Scheme, Lisp e Haskell uma facilidade de extensibilidade. Com essa facilidade, os usuários podem facilmente acrescentar muitas formas funcionais do tipo "aplicável a tudo". Muitas funções Scheme simples são definidas dessa maneira.

As Seções 14.3.8 a 14.3.10 desenvolvem vários exemplos interessantes que se combinam para ilustrar o valor exclusivo da programação funcional em Scheme. O primeiro exemplo reapresenta as semânticas de Clite, que originalmente foram discutidas e implementadas em Java (veja o Capítulo 8). O segundo exemplo, diferenciação simbólica, é um pouco mais clássico, enquanto o terceiro exemplo reconsidera o problema das oito rainhas, que originalmente foi implementado em Java (veja o Capítulo 13).

6. Em Scheme, a ordem de avaliação dos valores `expr1`, `expr2`, e assim por diante, em uma `let` não está implícita. Em outras palavras, cada valor é avaliado independentemente das ligações dos nomes `var1`, `var2`, e assim por diante. Se for necessário se referir a um nome anterior em uma expressão posterior dentro de uma `let`, a função `let*` deverá ser usada.

7. A função interna `map` em Scheme é equivalente a `mapcar`.

Observação

Tracing

Muitos interpretadores Scheme proporcionam um recurso *tracing*, ou passo-a-passo, para ajudar a entender os detalhes do comportamento de uma função. Isso é particularmente útil na depuração. Infelizmente, nenhuma função *tracing* particular faz parte da Standard Scheme (Linguagem Scheme padrão), e assim seu uso varia ligeiramente de uma implementação para outra. A seguir, damos um exemplo do uso de uma função *tracing* para executar passo a passo a ativação e a desativação de chamadas na função `factorial` discutida anteriormente:

```
> (trace factorial)              Trace: Value = 1
> (factorial 4)                  Trace: Value = 1
Trace: (factorial 4)             Trace: Value = 2
Trace: (factorial 3)             Trace: Value = 6
Trace: (factorial 2)             Trace: Value = 24
Trace: (factorial 1)             24
Trace: (factorial 0)             > (untrace factorial)
                                 > (factorial 4)
                                 24
```

Para algumas implementações Scheme, o efeito do *tracing* pode ser conseguido por meio da técnica imperativa padrão de encaixar comandos de impressão de depuração. Uma função conveniente para ser usada é `printf`, que é similar à função do mesmo nome em C. A função `printf` toma como seu primeiro argumento uma *string* que especifica como a saída deve ser mostrada; o código `~a` é usado para mostrar um valor, enquanto `~n` é usado para representar um código de fim de linha. Usando isso, podemos reescrever a função fatorial para obter um resultado similar ao que vimos acima, usando:

```
(define (factorial n)
   (printf "(factorial ~a ~n)" n)
   (if (<= n 0) 1
     ; else
     (let ((x (* n (factorial (- n 1)))))
          (printf "(factorial ~a) = ~a ~n" n x)
          x
     )
))
```

A seguir está um exemplo do resultado para essa função `factorial` modificada:

```
> (factorial 3)          (factorial 1) = 1
(factorial 3)            (factorial 2) = 2
(factorial 2)            (factorial 3) = 6
(factorial 1)            6
(factorial 0)
```

Isso funciona facilmente porque as funções `define` e `let` possibilitam uma seqüência de funções como um corpo e retornam o valor da última função computada.

14.2.8 Exemplo: Semânticas de Clite

Nesta seção, implementamos muitas das semânticas de Clite usando Scheme. Recorde-se do Capítulo 8, no qual foi dito que, para a linguagem elementar Clite, o ambiente é estático; assim o estado pode ser simplesmente representado como uma coleção de pares variável-valor. Isso é expresso da seguinte forma:

$$state = \{\langle var_1, val_1\rangle, \langle var_2, val_2\rangle, \ldots, \langle var_m, val_m\rangle\}$$

Aqui, cada var_i representa uma variável, e cada val_i representa seu valor atual atribuído.

O estado é como uma *janela de observação* em um ambiente de desenvolvimento integrado IDE (Integrated Development Environment). Ele está sempre ligado a uma instrução particular no programa-fonte, e para cada variável do programa mostra seu valor atual. Em nossa implementação Java, o estado foi implementado como uma tabela *hash* na qual o identificador de variável era a chave e o valor associado era o valor corrente da variável (veja os detalhes no Capítulo 8).

Uma idéia fundamental na implementação Scheme é que um estado é naturalmente representado como uma lista, e cada elemento da lista é um par que representa a ligação de uma variável ao seu valor. Assim, o estado Clite:

$$\{\langle x, 1\rangle, \langle y, 5\rangle\}$$

pode ser representado como a lista Scheme:

```
((x 1) (y 5))
```

Primeiro, implementamos as funções de acesso de estado `get` e `onion` (união de substituição) da implementação Java (veja o Capítulo 8). Lembre-se de que `get` é usada para obter o valor de uma variável a partir do estado atual. Em Scheme, o código necessário é similar à função `member`:

```
(define (get id state)
  (if (equal? id (caar state)) (cadar state)
      (get id (cdr state))
))
```

Como o sistema de tipos Clite requer que todas as variáveis usadas em um programa sejam declaradas, e nós assumimos que a sintaxe abstrata foi estaticamente verificada conforme descrito no Capítulo 6, não pode haver uma referência a uma variável que não esteja no estado. Assim, a função `get` é mais simples do que a função `member` nesse aspecto, já que ela não precisa ser verificada para uma lista nula. Um exemplo da função `get` é:

```
(get 'y '((x 5) (y 3) (z 1)))
     = (get 'y '((y 3) (z 1)))
     = 3
```

Um bom modelo para a função `onion` é a função `subst` já definida anteriormente, sendo que uma diferença é a estrutura das listas e a outra diferença é que deve haver exatamente uma ocorrência de cada variável no estado:

```
(define (onion id val state)
   (if (equal? id (caar state))
       (cons (list id val) (cdr state))
       (cons (car state) (onion id val (cdr state)))
))
```

A função `onion` pode então adotar a hipótese simplificadora de que a variável que estamos buscando ocorre dentro do estado, já que supomos verificação semântica estática conforme foi descrito no Capítulo 6. Portanto, não há necessidade de verificar a existência de uma lista nula como *base case*. A função `onion` pode também interromper a repetição uma vez que tenha sido encontrada a variável. Veja um exemplo da função `onion`:

```
(onion 'y 4 '((x 5) (y 3) (z 1)))
  = (cons '(x 5) (onion 'y '((y 3) (z 1)))
  = (cons '(x 5) (cons '(y 4) '((z 1))))
  = '((x 5) (y 4) (z 1))
```

Ao desenvolver funções semânticas para Clite, assumimos que as instruções em sintaxe abstrata Clite (veja a Figura 2.14) são representadas como listas Scheme da seguinte forma:

```
(skip)
(assignment target source)
(block s1 ... sn)
(loop test body)
(conditional test thenbranch elsebranch)
```

Aqui, cada campo individual para um tipo de instrução abstrata não é nomeado, como era em Java, mas sim identificado por sua posição na lista. Assim, em uma instrução `loop` o `test` é o segundo elemento (isso é, o `cadr`) da lista, enquanto o `body` é o terceiro elemento (o `caddr`).

A função de significado para uma instrução Clite pode ser escrita como uma simples instrução *case* em Scheme:

```
(define (m-statement statement state)
  (case (car statement)
        ((skip) (m-skip statement state))
        ((assignment) (m-assignment statement state))
        ((block) (m-block (cdr statement) state))
        ((loop) (m-loop statement state))
        ((conditional) (m-conditional statement state))
        (else ())
))
```

O propósito de uma *Instrução* abstrata é uma função de transformação de estado da forma que toma um *Estado* como entrada e produz um *Estado* como saída. A implementação

dessas funções de significado resulta diretamente das regras dadas no Capítulo 8 (e resumidas aqui). Assumimos também que foi executada uma verificação semântica estática, conforme descrito no Capítulo 6.

Uma instrução *Skip* corresponde a uma instrução vazia. Como tal, ela deixa o estado inalterado; o estado de saída é uma cópia do estado de entrada.

```
(define (m-skip statement state) state)
```

O propósito de um *Block* é exatamente aquele de suas instruções aplicadas ao estado corrente na ordem em que elas aparecem. Isto é, se um *Block* não tiver nenhuma instrução, o estado não é alterado. Caso contrário, o estado resultante do sentido da primeira *Statement* no *Block* torna-se a base para definir o sentido do resto do bloco. Essa é uma definição repetitiva.

Isso é implementado em Scheme, que interpreta primeiro o significado da primeira instrução na lista, e então aplica repetidamente essa função ao restante da lista. Nós já eliminamos o `block` tag do cabeçalho da lista na função `m-statement`.

```
(define (m-block alist state)
  (if (null? alist) state
      (m-block (cdr alist) (m-statement (car alist) state))
))
```

Uma instrução *Loop* tem um teste boleano, que é uma *Expression*, e um corpo *Statement*. Se o teste boleano não resultar *true* (*verdadeiro*), o significado (estado de saída) de um *Loop* será o mesmo que o estado de entrada; caso contrário, o significado será o estado resultante da primeira execução do corpo da instrução uma vez, depois passando o estado resultante para a reexecução do *Loop*.

A implementação Scheme segue quase diretamente dessa definição:

```
(define (m-loop statement state)
   (if (m-expression (car statement) state)
       (m-loop statement (m-statement (cdr statement) state))
       state
))
```

Finalmente, considere a função de significado Scheme para avaliação de expressão Clite apenas para inteiros. Para facilitar, usamos a seguinte representação de lista para cada tipo de expressão Clite abstrata:

```
(value val), where val is an integer
(variable id), where id is a variable name
(operator term1 term2), where operator is one of:
    plus, minus, times, div -- arithmetic
    lt, le, eq, ne, gt, ge -- relational
```

A função de significado para uma expressão abstrata Clite é implementada usando um *case* no tipo de expressão. O significado do valor da expressão é exatamente o próprio valor (isto é, o `cadr`). O significado de uma variável é o valor associado com o identificador da variável (o `the cadr`) no estado corrente.

O significado de uma expressão binária é obtido aplicando-se o operador aos significados dos operandos (o `cadr` e o `caddr`):

```
(define (m-expression expr state)
  (case (car expr)
        ((value) (cadr expr))
        ((variable) (get (cadr expr) state))
        (else (applyBinary (car expr) (cadr expr)
                           (caddr expr) state))
))
```

A função `applyBinary` definida no Capítulo 8 limitada aos inteiros é facilmente implementada como um *case* no operador. Aqui mostramos apenas os operadores aritméticos, deixando a implementação dos operadores relacionais como exercício:

```
(define (applyBinary op left right state)
  (let ((leftval (m-expression left state))
        (rightval (m-expression right state)))
    (case op
      ((plus)     (+ leftval rightval))
      ((minus)    (- leftval rightval))
      ((times)    (* leftval rightval))
      ((div)      (/ leftval rightval))
      (else #f))
))
```

A implementação dos operadores relacionais, bem como a própria instrução de atribuição, fica como exercício.

Como exemplo da aplicação da função `m-expression` à expressão `y+2` no estado `{<x,5>, <y,3>, <z,1>}`, considere o seguinte:

```
(m-expression '(plus (variable y) (value 2)) '((x 5) (y 3) (z 1)))
= (applyBinary '(plus (variable y) (value 2)) '((x 5) (y 3) (z 1)))
= (+ (m-expression '(variable y) '((x 5) (y 3) (z 1)))
     (m-expression '(value 2) '((x 5) (y 3) (z 1))))
= (+ (get 'y '((x 5) (y 3) (z 1)))
     (m-expression '(value 2) '((x 5) (y 3) (z 1))))
= (+ 3
     (m-expression '(value 2) '((x 5) (y 3) (z 1))))
= (+ 3 2)
= 5
```

Esse desenvolvimento, mesmo sendo de uma pequena fração da semântica de Clite, deverá ser suficiente para convencê-lo de que um modelo semântico completo para uma linguagem imperativa pode ser implementado em Scheme.

Assim, por meio da interpretação, Scheme é capaz de calcular qualquer função que possa ser programada em uma linguagem imperativa. O inverso também é verdade, pois os computadores modernos são fundamentalmente imperativos em sua natureza. Como os interpretadores Scheme são implementados nessas máquinas, qualquer função programada em Scheme pode ser computada por um programa imperativo. Portanto, de fato, as linguagens imperativas e as linguagens funcionais são equivalentes quanto ao poder computacional.

14.2.9 Exemplo: Diferenciação Simbólica

A utilidade da linguagem Scheme para manipulação simbólica é ampla, conforme sugere o exemplo anterior. O exemplo seguinte ilustra ainda mais parte do poder de Scheme fazendo diferenciação simbólica e simplificação de fórmulas simples de cálculo. Algumas regras familiares para diferenciação simbólica são dadas na Figura 14.2.

Por exemplo, diferenciando-se a função $2 \cdot x + 1$ em relação ao x usando essas regras, obtemos:

$$\begin{aligned}\frac{d(2 \cdot x + 1)}{dx} &= \frac{d(2 \cdot x)}{dx} + \frac{d1}{dx} \\ &= 2 \cdot \frac{dx}{dx} + x \cdot \frac{d2}{dx} + 0 \\ &= 2 \cdot 1 + x \cdot 0 + 0\end{aligned}$$

que comumente se simplifica resultando no valor 2.

$\frac{d}{dx}(c) = 0$	c é uma constante
$\frac{d}{dx}(x) = 1$	
$\frac{d}{dx}(u+v) = \frac{du}{dx} + \frac{dv}{dx}$	u e v são funções de x
$\frac{d}{dx}(u-v) = \frac{du}{dx} - \frac{dv}{dx}$	
$\frac{d}{dx}(uv) = u\frac{dv}{dx} - v\frac{du}{dx}$	
$\frac{d}{dx}\left(\frac{u}{v}\right) = \left(v\frac{du}{dx} - u\frac{dv}{dx}\right) \Big/ v^2$	

| Figura 14.2 Regras de Diferenciação Simbólica

Em Scheme, é conveniente representar expressões usando a notação Polish prefix:

```
(+ term1 term2)
(- term1 term2)
(* term1 term2)
(/ term1 term2)
```

A função requerida para fazer a diferenciação simbólica primeiro testa, para determinar se a expressão é uma constante ou a variável que está sendo diferenciada, como nas duas primeiras regras descritas. Caso contrário, a expressão é uma lista que começa com um operador, e o código aplica uma das quatro regras restantes usando um *case* no operador. Foi usada uma função `let` para fazer o código Scheme ficar o mais semelhante possível às regras da Figura 14.2.

```
(define  (diff x  expr)
  (if  (not (list?  expr))
    (if  (equal?  x  expr)  1  0)
    (let ((u (cadr expr)) (v (caddr expr)))
       (case  (car expr)
            ((+)  (list  '+  (diff  x  u) (diff  x  v)))
            ((-)  (list  '-  (diff  x  u) (diff  x  v)))
            ((*)  (list  '+
                      (list  '* u (diff x v))
                      (list  '* v (diff  x  u))))
            ((/)  (list  'div  (list '-
                      (list  '* v (diff  x  u))
                      (list  '* u (diff x v)))
                      (list  '* u v)))
))))
```

Uma aplicação da função `diff` à expressão $2 \cdot x + 1$ resulta em:

```
(diff 'x '(+ (* 2 x) 1))
   = (list '+ (diff 'x '(* 2 x)) (diff 'x 1))
   = (list '+ (list '+ (list '* 2 (diff 'x 'x))
                       (list '* x (diff 'x 2)))
              (diff 'x 1))
   = (list '+ (list '+ (list '* 2 1) (list '* x (diff 'x 2)))
              (diff 'x 1))
   = (list '+ (list '+ '(* 2 1) (list '* x (diff 'x 2)))
              (diff 'x 1))
   = (list '+ (list '+ '(* 2 1) (list '* x 0)) (diff 'x 1))
   = (list '+ (list '+ '(* 2 1) '(* x 0)) (diff 'x 1))
   = (list '+ '(+ '(* 2 1) '(* x 0)) (diff 'x 1))
   = (list '+ '(+ '(* 2 1) '(* x 0)) 0)
```

que, na forma interfixada, é $2 * 1 + 0 * x + 0$. O resultado normal, 2, apareceria após simplificar essa expressão. Fica como exercício escrever um simplificador de expressão.

14.2.10 Exemplo: O Problema das Oito Rainhas

No Capítulo 13, desenvolvemos uma versão orientada a objetos do algoritmo *backtracking* de Wirth (Wirth, 1976). A solução geral que Wirth apresenta é um procedimento repetitivo que executa uma iteração por meio de uma série de movimentos. Cada movimento é testado para ver se ele satisfaz a um critério apropriado de *validade*. Se ele satisfizer, o movimento será registrado, e se o problema ainda não estiver resolvido, o procedimento chamará a si próprio repetidamente para tentar o próximo nível. Se a chamada repetitiva falhar, então o movimento atual será desfeito e será tentado o próximo movimento. O processo de tentar movimentos continua até que seja encontrada uma solução completamente satisfatória ou todos os movimentos no nível atual tenham sido tentados sem sucesso.

A solução iterativa geral para esse problema, de acordo com Wirth (1976, p. 136), pode ser conseguida na seguinte notação semelhante a C:

```
boolean try(Solution) {
 boolean successful = false;
 initialize moves;
 while (more moves && !successful) {
  select next move;
  if (move is valid) {
   record move in Solution;
   if (Solution not done) {
     successful = try(Solution);
     if (!successful)
        undo move in Solution;
   }
  }
 }
 return successful;
}
```

Esta seção desenvolve uma versão puramente funcional desse algoritmo e então especializa-o para resolver o problema das oito rainhas. Essa solução é interessante porque mostra alguns dos aspectos positivos e negativos da programação funcional, em constraste com a programação imperativa e orientada a objetos.

Há efetivamente dois problemas que devemos resolver na conversão desse modelo geral, imperativo, em outro puramente funcional. O primeiro problema é que a função `try` retorna dois resultados: se foi ou não encontrada uma solução de forma bem-sucedida e a própria solução.

Na versão imperativa descrita acima, a função `try` retorna `successful` como valor da função, e é retornada a solução (*Solution*) como parâmetro de referência: os comandos *record* e *undo* são atribuições generalizadas implementadas como chamadas de função. Uma linguagem puramente funcional não tem parâmetros e atribuições de referência, nem pode retornar mais do que um único resultado. Para resolver esse problema em Scheme, nós podemos `cons` o valor da variável `successful` destacar a própria solução.

O segundo problema é que o modelo de Wirth usa um laço *while* para iterar através de uma série de movimentos até que seja encontrada uma solução completamente bem-sucedida ou não haja mais movimentos nesse nível. Uma codificação Scheme direta desse problema poderia usar as características imperativas de Scheme para essencialmente duplicar

o algoritmo de Wirth. No entanto, para apresentar um equivalente puramente funcional à estrutura imperativa de laço, devemos substituir o laço por uma função repetitiva.

Desenvolvemos esse algoritmo um tanto às avessas, dependendo do seu ponto de vista. Isso significa que atacaremos as instruções *if* internas primeiro, depois o laço *while*, e finalmente a função global. Funções particulares ao problema atual que está sendo resolvido, como quando a solução está esgotada quando não há mais movimentos, e assim por diante, permanecerão não especificadas por enquanto.

A primeira função, chamada `tryone`, foi desenvolvida para resolver a instrução *se o movimento for válido* no modelo acima:

```
(define (tryone move soln)
  (let ((xsoln (cons move soln)))
    (if (valid move soln)
    (if (done xsoln) (cons #t xsoln)
        (try xsoln))
    (cons #f soln)
)))
```

A função `tryone` só é chamada se a variável `successful` for *false*; assim, o parâmetro `soln` não tem o valor de `successful` no início da lista. No entanto, ela retorna uma solução com o valor de `successful` no início da lista.

Observe que usamos uma função `let` de uma forma restrita para evitar computar a solução estendida `xsoln` duas vezes. A função primeiro verifica se o movimento atual é válido, dada a solução parcial atual. Se ele não for válido, então ele retorna *false* para `successful` e a solução corrente via `(cons #f soln)`. Se o movimento for válido, então ele verifica se a solução estendida resolve o problema (função `done`). Nesse caso, ele retorna *true* (verdadeiro) para `successful` e a solução estendida. Caso contrário, ele chama repetidamente `try` com a solução estendida para tentar continuar estendendo a solução.

Em seguida, convertemos o laço *while* em uma função repetitiva, usando a estratégia a seguir. Isto é, qualquer laço *while* imperativo da forma:

```
while (test) {
   body
}
```

pode ser convertido em uma função repetitiva da forma:

```
(define (while test body state)
   (if (test state)
       (let ((onepass (body state)))
            (while test body onepass)
       )
       state
))
```

Aqui, a variável `onepass` fornece o estado que resulta ao executar o corpo do laço *while* uma vez. Assim, se o teste resulta em *true* (*verdadeiro*), a instrução *while* é executada novamente após fazer uma passagem pelo corpo do laço *while*. Caso contrário, é retornado o estado atual.

O estado do programa para as oito rainhas é um pouco mais complicado, mas uma estratégia geral de conversão de laço produz a seguinte função:

```
(define (trywh move soln)
  (if (and (hasmore move) (not (car soln)))
      (let ((atry (tryone move (cdr soln))))
       (if (car atry) atry (trywh (nextmove move) soln))
  )
  soln
))
```

Observe que a função `let` aparece de forma restrita para evitar escrever a chamada à função `tryone` duas vezes. Observe também que a função `trywh` espera que o valor da variável `successful` esteja no início da lista, enquanto a função `tryone` não, já que `tryone` só é chamada quando `successful` é *false* (*falso*).

Finalmente, implementamos a função `try`. Ela é chamada com uma solução parcial sem a variável `successful`, retorna a `cons` da variável `successful` e assim é encontrada a solução. Ela é responsável por obter o primeiro movimento para inicializar o laço *while*:

```
(define (try soln) (trywh 1 (cons #f soln)))
```

Para especificar essa estratégia geral e resolver um problema em particular, devemos implementar várias funções.

- As funções `hasmore` e `nextmove` servem para gerar testes de movimentos.
- A função `valid` verifica se um teste de movimento estende de forma válida a solução parcial corrente.
- A função `done` testa para determinar se uma solução estendida resolve o problema.

Ilustramos implementações dessas funções desenvolvendo uma solução para o problema das oito rainhas.

Uma preocupação inicial no problema das oito rainhas é a de como armazenar a posição (linha e coluna) de cada uma das rainhas. Recorde-se da Seção 13.4.2, em que desenvolvemos a solução uma coluna de cada vez, armazenando a posição da linha para cada coluna usando uma matriz. Na solução desenvolvida aqui, armazenamos a posição da linha para cada coluna usando uma lista, mas com uma diferença fundamental. Armazenamos a lista em ordem inversa, de maneira que a linha acrescentada mais recentemente fica sempre no início da lista. Por exemplo, o tabuleiro com três rainhas nas posições (linha, coluna) mostradas na Figura 14.3 é representado como a lista a seguir:

```
(5 3 1)
```

Se a variável `N` representa o número de linhas e colunas do tabuleiro de xadrez, podemos definir as funções para gerar movimentos como:

```
(define (hasmore move) (<= move N))
(define (nextmove move) (+ move 1))
```

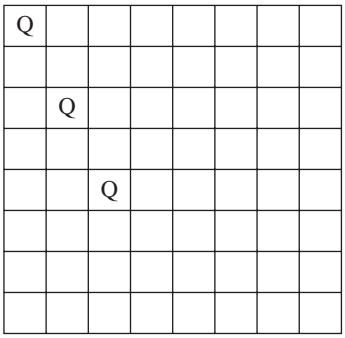


Figura 14.3
Três Rainhas em um Tabuleiro de Xadrez 8 × 8

que gera números de linha na seqüência de 1 a `N`. Semelhantemente, podemos definir uma solução a ser "feita" da seguinte maneira:

```
(define (done soln) (>= (length soln) N))
```

Agora, tudo o que resta é definir se um teste de fileira estende ou não a solução parcial atual de forma válida. Lembre-se do Capítulo 13, no qual é dito que devem ser satisfeitas três condições:

1. A fileira testada não deve estar ocupada. Isso significa que a fileira testada (ou o movimento) não deve ser um membro da solução atual.
2. A diagonal sudoeste formada pela fileira e coluna testada não deve estar ocupada. A diagonal sudoeste é a soma dos números de linha e coluna.
3. A diagonal sudeste formada pela fileira e coluna não deve estar ocupada. A diagonal sudeste é a diferença dos números de linha e coluna.

Dado um número de linha e coluna, as diagonais sudoeste e sudeste são facilmente calculadas como:

```
(define (swDiag row col) (+ row col))
(define (seDiag row col) (- row col))
```

Para testar uma solução, devemos primeiro converter uma lista de posições de linhas em uma lista de posições diagonais sudoeste e sudeste. Para um dado teste `soln` a posição da linha do teste de movimento é `(car soln)` e o número de coluna associado é `(length soln)`. As funções `selist` e `swlist` desenvolvem essas listas para qualquer teste de solução.

```
(define (selist alist)
   (if (null? alist)
        '()
        (cons (seDiag (car alist) (length alist))
              (selist (cdr alist)))))
(define (swlist alist)
   (if (null? alist)
        '()
         (cons (swDiag (car alist)(length alist))
               (swlist (cdr alist)))))
```

Finalmente, as três condições para o teste de solução podem ser testadas por meio da função `valid`. Essa função verifica se um teste de movimento corrente representando a posição de uma linha estende, de forma válida, a solução parcial atual. Isto é, o movimento (linha) não é um membro da solução, e o movimento (posição de linha e coluna associada) não é um membro da diagonal sudoeste nem da diagonal sudeste.

```
(define (valid move soln)
  (let ((col (length (cons move soln))))
    (and (not (member move soln))
      (not (member (seDiag move col) (selist soln)))
      (not (member (swDiag move col) (swlist soln)))
)))
```

Esse programa pode ser testado usando a chamada `(try ())`, na qual a variável global `N` define o tamanho do problema. Por exemplo, a declaração:

```
(define N 8)
```

irá particularizar a solução para um tabuleiro 8×8. Isso encerra nossa implementação funcional do *backtracking* e do problema das oito rainhas.

Esse exercício foi interessante por várias razões. Por um lado, ele mostra o poder da programação funcional. Por outro lado, nossa solução mostra algumas das fraquezas da programação funcional pura:

1. A conversão de um programa que tenha um laço iterativo, em uma função repetitiva, pode ser desnecessariamente tediosa.
2. O uso de uma lista para retornar múltiplos valores de uma função é inapropriado quando comparado ao uso de parâmetros de referência ou ao retorno de um objeto com variáveis de instância nomeadas.

Para compensar o primeiro ponto fraco (e também em favor da eficiência), Scheme estende a "Lisp pura" incluindo características imperativas tais como variáveis locais, instruções de atribuição e laços iterativos. O segundo ponto fraco é realmente uma fraqueza do sistema de tipo de Scheme, um problema que está substancialmente corrigido em linguagens funcionais posteriores como a Haskell. Discutiremos o sistema de tipo de Haskell na próxima seção.

14.3 HASKELL

Alguns desenvolvimentos recentes em programação funcional não são bem absorvidos pelas linguagens tradicionais, Common Lisp e Scheme. Nesta seção, introduzimos uma linguagem funcional mais moderna, Haskell (Thompson, 1999), cujas características assinalam mais claramente as direções presente e futura na pesquisa e nas aplicações de programação funcional. As características distintas e salientes de Haskell incluem sua estratégia de avaliação lenta e seu sistema de tipo. Embora Haskell seja uma linguagem fortemente tipada (todos os erros de tipo são identificados), às vezes um erro de tipo não é detectado até que o elemento do programa que contenha o erro seja realmente executado.

14.3.1 Introdução

Haskell tem uma sintaxe simples para escrever funções. Considere a função fatorial, que pode ser escrita em qualquer uma das maneiras a seguir:

```
-- equivalent definitions of factorial
fact1 0 = 1
fact1 n = n * fact1 (n - 1)

fact2 n = if n == 0 then 1 else n * fact2 (n - 1)

fact3 n
   | n == 0 = 1
   | otherwise = n * fact3 (n - 1)
```

Um hífen duplo (`--`) inicia um comentário Haskell, que continua até o fim da linha. A primeira versão, `fact1`, é escrita em um estilo recursivo, de modo que os casos especiais são definidos primeiro e seguidos pelo caso geral. A segunda versão, `fact2`, usa o estilo de definição mais tradicional if-then-else. A terceira versão, `fact3`, usa *guards* em cada direção; esse estilo é útil quando há mais de duas alternativas. Nos exemplos de aplicações, serão usados os três estilos.

Note a simplicidade da sintaxe. Diferentemente de Scheme, não há uma `define` introduzindo a definição da função, não há parênteses envolvendo os argumentos formais, não há vírgulas separando os argumentos. Além disso, não há um símbolo explícito de continuação (como em programação Unix shell) nem um terminador explícito (o ponto-e-vírgula em C/C++/Java). Em lugar disso, como em Python, Haskell acredita no recuo de construções continuadas. Em `fact3`, como ela é escrita sobre mais de uma linha, as *guards* ou a tecla pipe (`|`) devem ficar todas com o mesmo afastamento. Porém, os sinais de igual não precisam ficar alinhados. Uma definição extensa para uma *guard* convencionalmente começaria em uma nova linha e ficaria afastada do símbolo *guard*.

Em Haskell, os argumentos para uma função não ficam entre parênteses, tanto na definição da função quanto na sua invocação. Além disso, a invocação de função se liga mais fortemente do que operadores interfixados. Desse modo, a interpretação normal de `fact n - 1` é $fact(n) - 1$, que não é o que se deseja. Então, o valor `n - 1` deve ser colocado entre parênteses, já que é um único argumento para `fact` nas três variantes. A grandeza matemática $fact(n-1) * n$ seria escrita como:

```
fact (n - 1) * n
```

em que os parênteses são necessários, de maneira que o valor `n - 1` seja interpretado como o único argumento para `fact`.

Haskell é sensível a maiúsculas/minúsculas. Funções e variáveis devem começar com uma letra minúscula, enquanto tipos começam com uma letra maiúscula. Além disso, uma função não pode redefinir uma função padrão Haskell. Conforme veremos, as funções em Haskell são fortemente tipadas e polimórficas.

E, também, como a maioria das linguagens funcionais, Haskell usa, por padrão, inteiros de precisão infinita:

```
> fact2 30
265252859812191058636308480000000
```

uma resposta que claramente excede o maior valor `int` em um programa C/C++/Java.

Em Haskell, como em qualquer linguagem funcional, funções são objetos de primeira classe, em que funções não-avaliadas podem ser passadas como argumentos, construídas e retornadas como valores de funções. Além disso, funções podem ser *restringidas* quando um argumento *n* de função puder ter alguns de seus argumentos fixados. Uma *função restringida* é uma função de *n* argumentos, na qual alguns de seus argumentos são fixos. Como exemplo desse último caso, suponha que queremos definir uma função que dobra seu argumento:

```
double1 x = 2 * x
double2 = (2 *)
```

As funções `double1` e `double2` são equivalentes; a segunda é um exemplo de uma função restringida.

Com essa rápida introdução, vamos começar uma exploração mais sistemática de Haskell.

14.3.2 Expressões

As expressões em Haskell normalmente são escritas em notação interfixada, na qual o operador ou a função aparece entre seus operandos, como no exemplo a seguir:

```
2+2       -- compute the value 4
```

Quando essa expressão é apresentada a um interpretador Haskell, ele calcula o valor 4. Há os operadores usuais aritmético e relacional em Haskell, e podem ser criadas operações mais complicadas usando parênteses e as relações internas de precedência entre esses operadores. Veja, por exemplo, uma expressão Haskell que calcula o valor 48:

```
5*(4+6)-2
```

que seria equivalente à expressão Scheme `(- (* 5 (+ 4 6) 2))`. Além disso, podemos escrever expressões Haskell usando notação prefixada, desde que coloquemos parâmetros em todos os operadores e operandos disjuntos (*nonatomic*). Isso é ilustrado pela seguinte expressão (equivalente à expressão interfixada acima):

```
(-) ((*) 5 ((+) 4 6)) 2
```

Na Tabela 14.1 há um resumo mais completo dos operadores Haskell e das suas relações de precedência.

Os operadores *right-associative* (associativos à direita) são avaliados da direita para a esquerda quando eles são adjacentes em uma expressão no mesmo nível de parênteses; os operadores *left-associative* (associativos à esquerda) são avaliados da esquerda para a direita. Por exemplo, a expressão Haskell

```
2^3^4
```

representa 2 elevado à potência 3^4 (ou 2^{81} ou 2417851639229258349412352), e não 2^3 elevado à quarta potência (ou 2^{12} ou 4096). Os operadores não-associativos não podem aparecer adjacentes em uma expressão. Isso é, a expressão `a+b+c` é permitida, mas `a<b<c` não é.

| Tabela 14.1 Resumo dos Operadores Haskell e suas Precedências

Precedência	Associativo à Esquerda	Não-Associativo	Associativo à Direita
9	`!, !!, //`		`.`
8			`**, ^, ^ ^`
7	`*, /, 'div', 'mod', 'rem', 'quot'`		
6	`+, -`	`:+`	
5		`\\`	`:, ++`
4		`/=, <, <=, ==, >, >=, 'elem', 'notElem'`	
3			`&&`
2			`||`
1	`», »=`	`:=`	
0			`$, 'seq'`

O significado de muitos desses operadores deveria ser auto-explicativo. Muitos outros serão explicados na discussão seguinte.[8]

14.3.3 Listas e Extensões de Listas

Assim como Lisp e Scheme, a estrutura fundamental de dados de Haskell é a lista. Listas são coleções de elementos de certo tipo e podem ser definidas pela enumeração de seus elementos, como mostram as definições a seguir, para duas pequenas listas de números:

```
evens = [0, 2, 4, 6, 8]
odds = [1, 3 .. 9]
```

A lista `odds` é definida pela convenção matemática familiar usando reticências (. .) para omitir todos os elementos intermediários quando o padrão for óbvio. Graficamente, a lista `evens` é representada na Figura 14.1(a).

Alternativamente, uma lista pode ser definida por intermédio de algo chamado gerador, que toma a seguinte forma:

```
moreevens = [2*x | x <- [0..10]]
```

Isso significa, literalmente, "a lista de todos os valores `2*x` tais que `x` é um elemento na lista `[0..10]`". O operador `<-` representa o símbolo matemático ∈, que representa a participação na lista.

8. Haskell também suporta a definição de operadores adicionais, desde que eles sejam formados a partir dos seguintes símbolos: `# $ % * + . / < = > ? \ ^ | | : ~`.

Uma *extensão* de lista pode ser definida se usarmos um gerador, e a lista que ele define pode ser infinita. Por exemplo, a linha a seguir define a lista infinita que contém todos os inteiros pares não-negativos:

```
mostevens = [2*x | x <- [0,1 .. ]]
```

Aqui, o gerador é a expressão `x <- [0,1 .. ]`. Alternativamente, essa lista infinita poderia ter sido definida por:

```
mostevens = [0,2 .. ]
```

Esse exemplo ilustra uma grande diferença entre a Haskell e as linguagens funcionais tradicionais. Listas infinitas e as funções que calculam valores a partir delas são comuns em Haskell. Elas são possíveis devido ao compromisso geral da Haskell com a *avaliação lenta*, que diz simplesmente para não avaliar nenhum argumento para uma função até o momento em que for absolutamente necessário.[9] Para listas infinitas, isso significa que elas são armazenadas na forma não-avaliadas; no entanto, o *n*-ésimo elemento, não importa quão grande seja o valor de *n*, pode ser calculado sempre que for necessário.

Os geradores podem ter condições anexadas a eles, assim como na matemática. A função a seguir calcula os fatores de um número:

```
factors n = [ f | f <- [1..n], n `mod` f == 0]
```

Isso pode ser lido assim: os fatores de `n` são todos os números `f` no intervalo de um para `n` tal que `f` divide `n` exatamente. Observe que a expressão:

```
n `mod` f == 0
```

poderia ser escrita como:

```
mod n f == 0
```

Quando o nome de uma função é usado como um operador interfixado, o nome deve ser colocado entre caracteres delimitadores.

A função básica para construir uma lista é o operador interfixado : , que toma um elemento e uma lista como seus dois argumentos.[10] Aqui estão alguns exemplos, nos quais `[]` representa a lista vazia:

```
8:[]        -- gives [8]
6:8:[]      -- gives 6:[8] or [6,8]
4:[6,8]     -- gives [4,6,8]
```

Uma lista Haskell tem duas partes: o primeiro elemento ou a *head* da lista, e a lista dos demais elementos ou seu final (*tail*). As funções `head` e `tail` retornam essas duas

9. Lembre-se da distinção entre avaliação "rápida" e "lenta" feita pela primeira vez no Capítulo 9, no qual foi discutida a passagem de parâmetros. Deve ficar claro que a avaliação rápida de argumentos para uma função proibiria a definição de listas infinitas ou funções que operam sobre elas.

10. Esse operador é semelhante à função `cons` de Scheme.

partes, respectivamente. Referindo-se à lista `evens` representada na Figura 14.1, os exemplos a seguir ilustram essas funções:

```
head evens            --gives 0
tail evens            --gives [2,4,6,8]
head (tail evens)     --gives 2
tail (tail evens)     --gives [4,6,8]
head [6,8]            --gives 6
head 6:[8]            --gives 6
tail [6,8]            --gives [8]
tail [8]              --gives []
```

Combinando geradores e encadeamento de lista, pode ser definida a série de números primos usando a função `sieve`:

```
primes = sieve [2..]
    where
    sieve (p:xs) = p : sieve [ a | a <- xs, a 'mod' p /= 0]
```

Primeiro, observe o uso da cláusula `where`, que torna a definição de `sieve` local à definição de `primes` (análogo ao uso de `let` em Scheme ou na matemática). A definição diz que a lista de primos até `n` é retornada por `sieve` na lista de números de dois até `n`. A função `sieve`, dada uma lista que consiste de uma head `p` e um final `xs` (que é uma lista), é constituída da lista cuja *head* é `p` (que deve ser primo) e cujo final é o valor de `sieve` aplicado a `xs` com todos os múltiplos de `p` removidos. A segunda definição de `sieve` é um exemplo de reconhecimento de padrão e é explicada na Seção 14.3.6.

O principal operador para juntar listas é ++.[11] Esse operador é ilustrado pelos exemplos a seguir:

```
[1,2]++[3,4]++[5]       -- gives [1,2,3,4,5]
evens ++ odds           -- gives [0,2,4,6,8,1,3,5,7,9]
[1,2]++[]               -- gives [1,2]
[1,2]++3:[]             -- gives [1,2,3]
1++2                    -- error; wrong type of arguments for ++
```

Como a Haskell foi projetada para processar listas, ela contém um conjunto de funções especiais de processamento de lista. Devido à necessidade que aparece freqüentemente, há uma função especial `null` para testar quanto a uma lista vazia:

```
null []           -- gives True
null evens        -- gives False
null [1,2,3]      -- gives False
null 5            -- error; wrong type of argument for null
```

A Haskell contém funções para testar se um objeto é igual ou equivalente a outro. A função principal está incorporada no operador interfixado ==, que é razoavelmente geral.

11. Esse operador é semelhante à função `append` de Scheme.

Essa função retorna `True` se os dois objetos tiverem a mesma estrutura e o mesmo conteúdo; caso contrário, ela retorna `False` ou um erro de tipo:

```
5==5                    -- returns True
5==1                    -- returns False
[1,2]==[1,2]            -- returns True
5==[5]                  -- error; mismatched argument types
[1,2,3]==[1,[2,3]]      -- error; mismatched argument types
[1,2]==[2,1]            -- returns False
[]==[]                  -- returns True
```

Tipos lista podem ser definidos e, mais tarde, usados na construção de funções. Para definir um tipo lista `IntList` de valores `Int`, por exemplo, é usada a seguinte instrução:

```
type IntList = [Int]
```

Observe que o uso de sinais de parênteses assinala que o tipo que está sendo usado é uma espécie particular de lista – uma lista cujas entradas são do tipo `Int`.[12]

14.3.4 Tipos e Valores Elementares

Até aqui, todos os valores que vimos em Haskell são inteiros, símbolos predefinidos (nomes de função) e nomes de tipos. Haskell suporta vários tipos de valores elementares, incluindo boleanos (chamados de `Bool`), inteiros (`Int` e `Integer`), caracteres (`Char`), cadeias de caracteres (`String`) e números em ponto flutuante (`Float`).

Conforme já observamos, os valores boleanos são `True` e `False`. O tipo `Int` suporta um intervalo finito de valores (-2^{31} até $2^{31}-1$, que é o intervalo usual para representação em 32 bits). No entanto, o tipo `Integer` suporta inteiros de qualquer tamanho, e, portanto, contém uma série infinita de valores.

Caracteres em Haskell são representados entre aspas simples, como `'a'`, e são usadas convenções familiares de escape para identificar caracteres especiais, como `'\n'` para nova linha, `'\t'` para tabulação, e assim por diante. As *strings* são representadas como uma série de caracteres entre aspas duplas (`""`) ou uma lista de valores `Char`. Isto é, o tipo `String` é equivalente ao tipo `[Char]`. Assim a lista `['h','e','l','l','o']` é equivalente à String `"hello"`. Ou seja, a seguinte definição de tipo está implícita em Haskell:

```
type String = [Char]
```

Devido a essa equivalência, muitos operadores String são o mesmo que operadores lista. Por exemplo, a expressão

```
"hello" ++ "world"
```

representa encadeamento de *string*, e resulta em `"helloworld"`.

12. Os nomes de tipos Haskell são diferentes de outros nomes pelo fato de começarem com letra maiúscula.

Valores em ponto flutuante são escritos em notação decimal ou notação científica. Cada um dos seguintes valores representa o número 3.14.

```
3.14
0.000314e4
```

Há várias funções disponíveis para transformar valores em ponto flutuante em Haskell, incluindo os seguintes, cujos significados são razoavelmente auto-explicativos (argumentos para funções trigonométricas são expressos em radianos):

```
abs acos atan ceiling floor cos sin
log logBase pi sqrt
```

14.3.5 Fluxo de Controle

Os principais construtores de controle de fluxo em Haskell são os comandos *guarded* e o *if-then-else*.[13] O comando *guarded* é uma generalização de um *if-then-else* generalizado, e pode ser escrito mais abreviadamente. Por exemplo, suponha que queremos encontrar o máximo de três valores, x, y e z. Então podemos expressar isso como um if-then-else da seguinte maneira:

```
if x >= y && x >= z then x
else if y >= x && y >= z then y
     else z
```

Alternativamente, podemos expressar isso como um comando *guarded* da seguinte maneira:

```
| x >= y && x >= z = x
| y >= x && y >= z = y
| otherwise = z
```

O comando *guarded* é muito usado quando se definem funções Haskell, conforme veremos a seguir.

14.3.6 Definindo Funções

As funções Haskell são definidas em duas partes. A primeira parte identifica o nome da função, do domínio e do intervalo, e a segunda parte descreve o significado da função. Assim, uma definição de função tem a seguinte forma:

```
name :: Domain -> Range
name x y z
       | g1 = e1
       | g2 = e2
       :
       | otherwise = e
```

13. Haskell tem também uma função `case`, que é similar à `case` em Ada e a `switch` em Java e C. No entanto, essa função parece ser relativamente sem importância em programação Haskell, já que seu significado é *subsumed* pelo comando *guarded*.

Aqui, o corpo da função é expresso como um comando *guarded*, e o domínio e o intervalo podem ser de quaisquer tipos. Por exemplo, uma função máxima para três inteiros pode ser definida como:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z = x
  | y >= x && y >= z = y
  | otherwise        = z
```

A primeira linha em uma definição de função pode ser omitida, caso em que Haskell deriva aquela linha automaticamente, dando à função a interpretação mais ampla possível. Quando omitimos a primeira linha da definição acima, Haskell deriva o seguinte para ela:

```
max3 ::  Ord a => a -> a -> a -> a
```

Essa notação significa que se `a` é *qualquer* tipo ordenado (`Ord`), então o significado de `max3` é explicado pela definição acima. Um *tipo ordenado* em Haskell é qualquer tipo que aceita os operadores relacionais (==, !=, >=, >, < = e <), que permite ordenar seus valores individuais. Assim, nossa função `max3` está agora bem definida em argumentos que são `Int`, `Float`, `String` ou qualquer outro tipo cujos valores são ordenados, conforme está ilustrado nos exemplos a seguir:

```
> max3 6 4 1
6
> max3 "alpha" "beta" "gamma"
"gamma"
```

A função `max3` é um exemplo de uma função *polimórfica*.

> **Definição:** Uma *função polimórfica* é aquela cuja definição se aplica igualmente bem a argumentos de vários tipos, dentro das restrições dadas pela função assinatura.

Por exemplo, uma tentativa de calcular o valor máximo entre uma *string* e dois inteiros dá o seguinte erro em tempo de execução:

```
> max3 "a" 2 3
ERROR: Illegal Haskell 98 class constraint
*** Expression : max3 "a" 2 3
*** Type       : Num [Char] => [Char]
```

Funções em Haskell podem ser definidas com o uso de recursão ou iteração. Aqui está uma definição de função fatorial recursiva:

```
fact :: Integer -> Integer
fact n
     | n == 0 = 1
     | n > 0  = n*fact(n-1)
```

Aqui está sua parte que calcula de modo iterativo (`product` é uma função interna):

```
fact n = product[1..n]
```

Tarefas iterativas simples, como somar uma lista de números, também podem ser escritas recursivamente ou iterativamente em Haskell. A função a seguir também ilustra como Haskell usa reconhecedor de padrão para distinguir diferentes estruturas e partes de lista.

```
mysum []        = 0
mysum (x:xs)    = x + mysum xs
```

A primeira linha define `mysum` para o caso especial (base) em que a lista está vazia. A segunda linha define `mysum` para os outros casos, nos quais a lista é não-vazia – lá, a expressão `x:xs` define um padrão que separa o head (`x`) do resto (`xs`) da lista, de maneira que eles podem ser distinguidos na definição da função. Então, o reconhecedor de padrão proporciona uma alternativa para escrever um comando geral ou uma instrução if-then-else para distinguir casos alternativos na definição da função.

Essa função soma é definida em qualquer lista de valores numéricos – `Int`, `Integer`, `Float`, `Double` ou `Long`. Aqui estão alguns exemplos, com seus resultados:

```
> mysum [3.5,5]
8.5
> mysum [3,3,4,2]
12
```

Haskell fornece um conjunto de funções transformadoras de lista em sua biblioteca-padrão chamada Prelude. A Tabela 14.2 fornece um resumo dessas funções. Nessas e em outras definições de função, a notação `a` ou `b` significa "qualquer tipo de valor". Isto é, as várias funções na Tabela 14.2 são polimórficas até a extensão especificada em seu domínio e intervalo. Por exemplo, o tipo do elemento da lista passado para a função `head` não afeta seu significado.

Outra função simples é a função `member`, que testa para saber se um elemento ocorre como um membro de uma lista e retorna `True` ou `False` de forma correspondente. A definição começa com uma lista vazia como caso básico e retorna `False`. Caso contrário, ela testa para saber se o *head* da lista é igual ao elemento procurado; se for, ela retorna `True` ou, caso contrário, ela retorna o resultado da chamada repetitiva a si própria no final da lista.

```
member :: Eq a => [a] -> a -> Bool
member alist elt
   | alist == []          = False
   | elt == head alist    = True
   | otherwise            = member (tail alist) elt
```

Uma maneira alternativa de definir funções em Haskell é explorar seus recursos de reconhecedor de padrão. Considere o seguinte:

```
member [] elt      = False
member (x:xs) elt  = elt == x || member xs elt
```

Tabela 14.2 Algumas Funções Lista Comuns em Haskell

Função	Domínio e Intervalo	Explicação
`:`	`a -> [a] -> [a]`	Acrescenta um elemento no início de uma lista
`++`	`[a] -> [a] -> [a]`	Junta (concatena) duas listas unidas
`!!`	`[a] -> Int -> a`	x !! n retorna o n-ésimo elemento da lista x
`length`	`[a] -> Int`	Número de elementos em uma lista
`head`	`[a] -> a`	Primeiro elemento em uma lista
`tail`	`[a] -> [a]`	Todos os elementos da lista, exceto o primeiro
`take`	`Int -> [a] -> [a]`	Toma n elementos do início de uma lista
`drop`	`Int -> [a] -> [a]`	Retira n elementos do início de uma lista
`reverse`	`[a] -> [a]`	Inverte a ordem dos elementos de uma lista
`elem`	`a -> [a] -> Bool`	Verifica se um elemento ocorre em uma lista
`zip`	`[a] -> [b] -> [(a,b)]`	Faz uma lista de pares *
`unzip`	`[(a,b)] -> ([a],[b])`	Faz um par de listas
`sum`	`[Int] -> Int`	Soma os elementos de uma lista
	`[Float] -> Float`	
`product`	`[Int] -> Int`	Multiplica os elementos de uma lista
	`[Float] -> Float`	

*Um "par" é um exemplo de uma *tupla* em Haskell (veja a Seção 14.3.7).

Essa definição alternativa é equivalente à primeira. Sua segunda linha combina as duas últimas linhas da primeira definição, aproveitando as vantagens da função disjunção (`||`) pela avaliação lenta de seus argumentos.

Nos exemplos anteriores, os argumentos para uma função são valores simples ou listas. Haskell também permite que uma função seja um argumento para outra função que está sendo definida:

```
flip f x y = f y x
```

Nesse caso, a função `flip` toma uma função `f` como argumento; `flip` então chama `f` com seu argumento invertido. Um dos usos de `flip` é definir `member` em termos de `elem`:

```
member xs x = elem x xs
member = elem . flip
```

em que o ponto representa composição de função. Isto é, *f.g(x)* é definida para se tornar *f*(*g*(*x*)). Muitas vezes, é preferível a segunda forma, e não a primeira.

Outro exemplo de uma função como um argumento é:

```
maphead :: (a -> b) -> [a] -> [b]
maphead fun alist = [ fun x | x <- alist ]
```

A função `maphead` tem dois parâmetros, uma função `fun` e uma lista `alist`. Ela aplica a função `fun` a cada elemento `x` de `alist`, criando uma lista dos resultados. A função `maphead` é equivalente à função interna `map`.

Como exemplo, considere a função quadrado:

```
square x = x*x
```

Essa função pode ser combinada com `maphead` para elevar ao quadrado todos os elementos de uma lista:

```
maphead square [2,3,5,7,9]        -- returns [4,9,25,49,81]
maphead (\x -> x*x) [2,3,5,7,9] -- an alternative
```

No primeiro exemplo, o primeiro parâmetro é o nome de uma função predefinida. No segundo exemplo, é definida uma função sem nome, usando a notação lambda[14] e passada para a função maphead.

14.3.7 Tuplas

Uma *tupla* em Haskell é uma coleção de valores de diferentes tipos,[15] colocados entre parênteses e separados por vírgulas. Aqui está uma tupla que contém uma `String` e um `Integer`:

```
("Bob", 2771234)
```

Valores tupla são definidos de uma maneira similar a valores de lista, exceto pelo fato de eles serem colocados entre parênteses `( )` e não entre colchetes `[ ]`, e seu tamanho ser inflexível. Por exemplo, aqui está uma definição para o tipo `Entry`, que pode representar uma entrada em um catálogo de telefone:

```
type Entry = (Person, Number)
type Person = String
type Number = Integer
```

Essa definição combina o nome de uma pessoa e seu número de telefone como uma tupla. O primeiro e o segundo membros de uma tupla podem ser selecionados com o uso das funções internas `fst` e `snd`, respectivamente. Por exemplo:

```
fst ("Bob", 2771234) = "Bob"
snd ("Bob", 2771234) = 2771234
```

Continuando esse exemplo, é natural definir um tipo `Phonebook` como uma lista de pares de nomes e números de pessoas:

```
type Phonebook = [(Person, Number)]
```

Agora podemos definir funções úteis nesse tipo de dado, como, por exemplo, a função `find` que retorna todos os números de telefone para determinada pessoa `p`:

```
find :: Phonebook -> Person -> [Number]
find pb p = [n | (person, n) <- pb, person == p]
```

14. O símbolo \ é usado em Haskell para aproximar o símbolo grego λ na formação de uma expressão lambda. Em geral, a expressão lambda ($\lambda x\ .\ M$) é escrita em Haskell como `(\x->M)`.

15. O análogo para uma tupla em C/C++ é a `struct`.

Essa função retorna a lista de todos os números `n` do catálogo telefônico para os quais há uma entrada (`person,n`) e `person==p` (a pessoa desejada). Por exemplo, se:

```
pb = [("Bob", 2771234), ("Allen", 2772345), ("Bob", 2770123)]
```

então, a chamada

```
find pb "Bob"
```

retorna a lista:

```
[2771234, 2770123]
```

Além disso, podemos definir funções que acrescentam e excluem entradas de um catálogo telefônico:

```
addEntry :: Phonebook -> Person -> Number -> Phonebook
addEntry pb p n = [(p,n)] ++ pb

deleteEntry :: Phonebook -> Person -> Number -> Phonebook
deleteEntry pb p n = [entry | entry <- pb, entry /= (p, n)]
```

O sistema de tipo Haskell é uma ferramenta mais poderosa do que esse exemplo mostra. Ela pode ser usada para definir novos tipos de dados repetidamente, conforme ilustraremos na próxima seção.

As demais seções desenvolvem exemplos interessantes que ilustram alguns dos valores especiais da programação funcional em Haskell. O primeiro exemplo revê as semânticas de Clite, que foram originalmente discutidas e implementadas em Java.

14.3.8 Exemplo: Semânticas de Clite

Nesta seção, implementamos grande parte das semânticas de Clite usando Scheme. Lembre-se do Capítulo 8, no qual foi citado que, para a linguagem elementar Clite, o ambiente é estático, assim o estado pode ser simplesmente representado como uma coleção de pares variável-valor. Isso é expresso da seguinte maneira:

$$state = \{\langle var_1, val_1\rangle, \langle var_2, val_2\rangle, \ldots, \langle var_m, val_m\rangle\}$$

Aqui, cada var_i representa uma variável e cada val_i representa seu valor atribuído no momento.

O estado é como uma *janela de observação* em um ambiente de desenvolvimento integrado (*integrated development environment* – IDE). Ele está sempre ligado a uma instrução particular no programa e mostra para cada variável de programa seu valor atual. Em nossa implementação Java, o estado era implementado como uma tabela *hash* na qual o identificador da variável tem a chave e o valor associado tem o valor atual da variável.

Um ponto de partida para a implementação Haskell é representar um estado como uma lista de pares, com cada par representando a ligação de uma variável com seu valor. Isto é, aplicam-se as seguintes definições de tipo:

```
type State = [(Variable, Value)]
type Variable = String
data Value = Intval Integer | Boolval Bool
             deriving (Eq, Ord, Show)
```

A terceira linha nessa definição é um exemplo de uma definição *tipo algébrico* em Haskell, na qual o novo tipo `Value` é definido como um valor `Integer` ou `Bool`. A cláusula `deriving (Eq, Ord, Show)` declara que esse novo tipo herda a qualidade, as características de ordenação e a exibição de seus tipos componentes `Integer` e `Bool`, permitindo-nos assim usar as funções igualdade (==), ordem (<) e exibição (`show`) em todos os seus valores.

Assim, o estado Clite:

$\{\langle x, 1\rangle, \langle y, 5\rangle\}$

pode ser representado como uma lista Haskell:

```
[("x",(Intval 1)),("y",(Intval 5))]
```

Em seguida, implementamos as funções de acesso de estado `get` e `onion` (união de substituição) da implementação Java (veja o Capítulo 8).

Recorde-se de que a função `get` obtém o valor de uma variável a partir do estado atual. Em Haskell, o código necessário é similar à função-membro `member`:

```
get :: Variable -> State -> Value
get var (s:ss)
    | var == (fst s) = snd s
    | otherwise      = get var ss
```

na qual as funções (`fst s`) e (`snd s`) retornam o primeiro e o segundo membros de uma tupla `s`, respectivamente. Como o sistema de tipo Clite requer que todas as variáveis usadas em um programa sejam declaradas, não pode haver uma referência a uma variável que não esteja no estado. Assim, a função `get` é mais simples do que a função `member`, já que o *case* para a lista nula não precisa ser testado.

Uma aplicação da função `get` é:

```
get "y" [("x",(Intval 1)),("y",(Intval 5)),("z",(Intval 4))]
    = get "y" [("y",(Intval 5)),("z",(Intval 4))]
    = (Intval, 5)
```

Um bom modelo para a função `onion` (união de substituição) é a função `subst` definida anteriormente, sendo que uma diferença é a estrutura das listas e a outra é que deve haver exatamente uma ocorrência de cada variável no estado:

```
onion :: Variable -> Value -> State -> State
onion var val (s:ss)
    | var == (fst s) = (var, val) : ss
    | otherwise      = s : (onion var val ss)
```

Uma vez mais a função `onion` adota a hipótese simplificadora de que a variável pela qual estamos procurando ocorre dentro do estado; portanto, não há necessidade de verificar para uma lista nula como o *base case*. A outra hipótese simplificadora é que há

apenas uma única ocorrência da variável dentro do estado; portanto, a função `onion` não continua a se repetir, uma vez que ela encontra a primeira instância.

Uma aplicação da função `onion` é:

```
onion "y" (Intval 4) [("x",(Intval 1)),("y",(Intval 5))]
   = ("x",(Intval 1)) : onion "y" (Intval 4)
     [("y",(Intval 5))]
   = [("x",(Intval 1)),("y",(Intval 4))]
```

Em nossa discussão das funções semânticas para Clite, assumimos que instruções em sintaxe abstrata Clite (veja a Figura 2.14) são representadas como tipos de dados repetitivos Haskell da seguinte maneira:

```
data Statement = Skip |
               Assignment Target Source |
               Block [ Statement ] |
               Loop Test Body |
               Conditional Test Thenbranch Elsebranch
               deriving (Show)
type Target = Variable
type Source = Expression
type Test = Expression
type Body = Statement
type Thenbranch = Statement
type Elsebranch = Statement
```

Agora o significado para uma declaração abstrata Clite pode ser escrito como a seguinte função Haskell sobrecarregada[16] m:

```
m :: Statement -> State -> State
```

O significado de uma declaração (*Statement*) abstrata é uma função de transformação de estado que toma um *State* como entrada e produz um *State* como saída. A implementação dessas funções de significado é conseqüência direta das regras dadas no Capítulo 8 (e resumidas aqui). Assumimos também que foi executada uma verificação de semânticas estáticas, conforme foi descrito no Capítulo 6.

Uma instrução *Skip* corresponde a uma instrução vazia. Como tal, ela deixa o estado inalterado; o estado de saída é uma cópia do estado de entrada.

```
m (Skip) state = state
```

Uma instrução *Loop* tem um teste boleano, que é uma *Expression* (expressão), e um corpo *Statement*. Se o teste boleano não resultar em *true* (*verdadeiro*), o sentido (estado de saída) de um *Loop* será o mesmo que o estado de entrada; caso contrário, o sentido será o estado

16. Uma função *sobrecarregada* é ligeiramente diferente de uma função polimórfica. A primeira se refere a uma função que tem diferentes definições, dependendo dos tipos de seus argumentos. A segunda, como já vimos, tem uma definição que se aplica a todos os tipos de argumentos.

resultante de se executar primeiro seu corpo uma vez, depois passando o estado resultante para a reexecução do *Loop* (laço).

A implementação Haskell segue quase diretamente dessa definição:

```
m (Loop t b) state
   | (eval t state) == (Boolval True) = m(Loop t b)(m b state)
   | otherwise                        = state
```

Uma instrução *Assignment* (atribuição) consiste de um destino *Variable* (variável) e uma origem *Expression*. O estado de saída é computado a partir do estado de entrada substituindo o *Value* (valor) do destino *Variable* pelo valor calculado da origem *Expression*, que é avaliada usando o estado de entrada. Todas as outras variáveis têm no estado de saída o mesmo valor que elas tinham no estado de entrada.

A implementação Haskell de uma instrução de atribuição Clite utiliza a função de união de substituição junto com o sentido de *Assignment*.

```
m (Assignment target source) state
    = onion target (eval source state) state
```

O sentido de uma *Conditional* (condicional) depende da verdade ou da falsidade de seu teste boleano no estado corrente. Se o teste for verdadeiro, então o sentido da *Conditional* terá o sentido da *Statement* `thenbranch`; caso contrário, ele terá o sentido da *Statement* `elsebranch`.

Semelhantemente, a implementação do sentido de uma instrução *Conditional* segue diretamente dessa definição:

```
m (Conditional test thenbranch elsebranch) state
   | (eval test state) == (Boolval True)
            = m thenbranch state
   | otherwise
            = m elsebranch state
```

Finalmente, considere a função de significado Haskell para avaliação de expressão Clite apenas para inteiros. Para facilitar, escolhemos uma definição apropriada de tipo algébrico para expressões Clite abstratas:

```
data Expression = Var Variable |
                  Lit Value |
                  Binary Op Expression Expression
                  deriving (Eq, Ord, Show)
type Op = String
```

A função de significado para uma *Expression* Clite pode agora ser implementada da seguinte maneira:

```
eval :: Expression -> State -> Value
eval (Var v) state = get v state
eval (Lit v) state = v
```

$$\frac{d}{dx}(c) = 0 \qquad c \text{ é uma constante}$$

$$\frac{d}{dx}(x) = 1$$

$$\frac{d}{dx}(u+v) = \frac{du}{dx} + \frac{dv}{dx} \qquad u \text{ e } v \text{ são funções de } x$$

$$\frac{d}{dx}(u-v) = \frac{du}{dx} - \frac{dv}{dx}$$

$$\frac{d}{dx}(uv) = u\frac{dv}{dx} - v\frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{u}{v}\right) = \left(v\frac{du}{dx} - u\frac{dv}{dx}\right) \Big/ v^2$$

| Figura 14.4 Regras de Diferenciação Simbólica

Deixamos como exercício a definição Haskell de `eval` para expressões com operadores aritméticos e relacionais.

Esse desenvolvimento de uma pequena fração das semânticas formais de Clite deverá convencê-lo de que um modelo semântico completo para uma linguagem imperativa pode ser facilmente definido em Haskell.

14.3.9 Exemplo: Diferenciação Simbólica

A utilidade da Haskell para manipulação de símbolos é ampla, como mostrou o exemplo anterior. Esse próximo exemplo ilustra ainda melhor alguns dos recursos da Haskell fazendo diferenciação simbólica e simplificação de fórmulas simples de cálculo. Algumas regras familiares para diferenciação simbólica são dadas na Figura 14.4.

Por exemplo, diferenciando a função $2 \cdot x + 1$ com relação a `x` usando essas regras resulta:

$$\begin{aligned}\frac{d(2 \cdot x + 1)}{dx} &= \frac{d(2 \cdot x)}{dx} + \frac{d1}{dx} \\ &= 2 \cdot \frac{dx}{dx} + x \cdot \frac{d2}{dx} + 0 \\ &= 2 \cdot 1 + x \cdot 0 + 0\end{aligned}$$

que, simplificando, resultaria em 2.

Em Haskell, é conveniente representar expressões usando tipos de dados repetitivos (análogo à notação Polish prefixa de Scheme):

```
Add expr1 expr2
Sub expr1 expr2
Mul expr1 expr2
Div expr1 expr2
```

A função necessária para fazer a diferenciação simbólica primeiro testa para determinar se a expressão é uma constante ou a variável que está sendo diferenciada, como nas duas primeiras regras na tabela acima.

Caso contrário, a expressão é uma lista que começa com um operador, e o código aplica uma das quatro regras restantes usando o reconhecedor de padrão no operador.

```
data Expr = Num Int | Var String | Add Expr Expr |
            Sub Expr Expr | Mul Expr Expr |
            Div Expr Expr
             deriving (Eq, Ord, Show)
diff :: String -> Expr -> Expr

diff x (Num c) = Num 0
diff x (Var y) = if x == y then Num 1 else Num 0
diff x (Add u v) = Add (diff x u) (diff x v)
diff x (Sub u v) = Sub (diff x u) (diff x v)
diff x (Mul u v) = Add (Mul u (diff x v))
                       (Mul v (diff x u))
```

Uma aplicação da função `diff` à expressão $2 \cdot x + 1$ resulta:

```
Add (Add (Mul (Num 2) (Num 1))
    (Mul (Var "x") (Num 0)) (Num 0))
```

Para tornar o resultado um pouco mais claro, representamos aqui um formatador que produz uma versão com muitos parênteses na expressão de saída:

```
formatExpr (Num n) = show n
formatExpr (Var x) = x
formatExpr (Add a b) =
    "(" ++ formatExpr a ++ " + " ++ formatExpr b ++ ")"
formatExpr (Sub a b) =
    "(" ++ formatExpr a ++ " - " ++ formatExpr b ++ ")"
formatExpr (Mul a b) =
    "(" ++ formatExpr a ++ " * " ++ formatExpr b ++ ")"
formatExpr (Div a b) =
    "(" ++ formatExpr a ++ " / " ++ formatExpr b ++ ")"
```

Para a mesma expressão dada acima, o formatador produz:

```
(((2 * 1) + (x * 0)) + 0)
```

que é um pouco mais clara. O resultado normal, 2, ocorreria após simplificar essa expressão. Fica como exercício escrever um simplificador de expressão como esse.

14.3.10 Exemplo: O Programa das Oito Rainhas

Enfim, voltamos novamente nossa atenção para o problema de colocar N rainhas mutuamente antagônicas em um tabuleiro $N \times N$ de forma que nenhuma rainha possa capturar

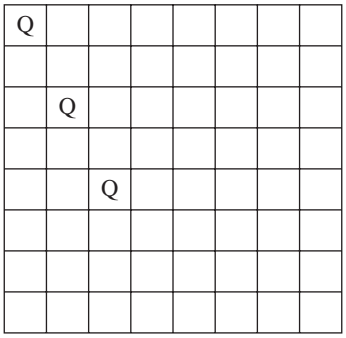


Figura 14.5
Três Rainhas em um Tabuleiro de Xadrez 8 × 8

outra rainha em um único movimento. Ao desenvolver a solução, usaremos as mesmas codificações das diagonais usadas na Seção 13.4.2. Porém, em vez de tentar converter o algoritmo *backtracking* de Wirth (Wirth, 1976) e depois adaptá-lo ao problema das oito rainhas, desenvolvemos uma versão puramente funcional do zero.

A primeira decisão é que a função desejada produza uma lista de todas as soluções possíveis, em que cada solução liste a posição da fila de cada rainha em ordem de coluna. Por exemplo, o tabuleiro com três rainhas nas posições (linha, coluna) mostradas na Figura 14.5 é representado pela seguinte lista:

```
[0, 2, 4]
```

No entanto, o programa é mais bem entendido quando se trabalha da direita para a esquerda. Tentando estender uma solução parcialmente segura, primeiro construímos novas listas com um teste de número de fileira (tomado da seqüência `[0..n-1]`). Lembre-se da Seção 13.4.2, em que uma *(row, col)* é segura:

- Se a fileira de teste não é um elemento da solução existente.
- Se as diagonais sudoeste e sudeste estiverem desocupadas. Na Seção 13.4.2, a diagonal sudoeste foi computada como *row* + *col* e a diagonal sudeste como *row* − *col*.

Essa verificação *segura* está englobada nas funções `safe` e `checks`. A função `safe` é passada para uma posição `b` e a um próximo teste de fileira `q`. Nesse programa, o tabuleiro é construído da direita para a esquerda; como as soluções são simétricas, a direção a partir da qual se quer trabalhar é puramente uma questão de preferência ou eficiência.

A linha `q` é segura em relação ao tabuleiro atual se as condições acima forem satisfeitas. Nesse caso, `checks` é chamada uma vez por cada valor de índice de `b`, isto é, de 0 a `length b - 1`. A verificação de linha para cada `i` é simplesmente: `q /= b!!i`. A verificação da diagonal sudoeste para cada `i` deverá ser `q+n /= b!!i - (n-i-1)`, que simplificando se torna `q - b!!i /= -i- 1`. Por uma análise similar, a verificação da diagonal sudeste para cada `i` se simplifica tornando-se `q - b!!i /= i+1`. A verificação no programa combina os dois casos tomando o valor absoluto.

Em um programa funcional, não há armazenamento global para armazenar e acessar as diagonais. Mesmo as linhas ocupadas são armazenadas como uma lista e passadas

como um argumento. Assim, nós preferimos computar dinamicamente as informações de diagonal conforme necessário, em vez de passá-las na lista de argumentos.

A solução para esse problema é:

```
queens n = solve n
    where
    solve 0 = [ [ ] ]
    solve (k+1) = [ q:b | b <- solve k,
         q <- [0..(n - 1)], safe q b ]
    safe q b = and [not (checks q b i) |
         i <- [0..(length b - 1)] ]
    checks q b i = q == b!!i || abs(q - b!!i) == i+1
```

Observe o uso da cláusula para ocultar as definições das funções helper `solve`, `safe` e `checks`. Veja também que o argumento formal para `queens`, ou seja, `n`, está referenciado em `solve`. Note a brevidade e a simplicidade dessa solução.

Repare que esse programa computa todas as soluções para um dado *n* como uma lista de listas de fileiras. Assim, `solve 0` retorna uma lista formada pela lista vazia, já que isso pode ser interpretado como uma solução válida. A função `solve` estende cada solução válida anterior (lista interna) filtrando cada número legal de linha com cada solução válida para saber se ela é segura. A função `safe` usa a função `checks` para saber se a linha ou as diagonais estão ocupadas, produzindo para cada tentativa de solução estendida uma lista de boleanos que primeiro é invertida e depois colocada junto. Nesse caso, uma linha ou diagonal ocupada produz *true* (verdadeiro), que é invertida como *false* (falso). Qualquer *false* (falso) na lista torna o *and* falso, resultando na rejeição da tentativa de solução estendida.

A execução desse programa para vários valores de `n` inclui:

```
> queens 0
[[]]
> queens 1
[[0]]
> queens 2
[]
> queens 3
[]
> queens 4
[[2,0,3,1],[1,3,0,2]]
```

que diz que:

- Para $n = 0$ uma solução consiste em não colocar nenhuma rainha.
- Para $n = 1$ uma solução consiste em colocar uma rainha em (0, 0).
- Para $n = 2, 3$ não há soluções.
- Para $n = 4$ há duas soluções, mas uma é a imagem espelhada da outra.

14.4 RESUMO

Este capítulo abrange os princípios e as pesquisas sobre aplicações do paradigma da programação funcional. A programação funcional é diferente dos outros paradigmas porque ela modela com precisão a idéia matemática de uma função.

As aplicações da programação funcional são fortemente baseadas na inteligência artificial. Este capítulo ilustra essas aplicações em Scheme e Haskell. Scheme é uma derivada de Lisp, a primeira linguagem de programação funcional importante. Haskell é uma linguagem mais recente, cujas distinções se beneficiam de sua estratégia de avaliação lenta.

EXERCÍCIOS

14.1 Avalie as seguintes expressões lambda usando redução-beta *rápida* (use as interpretações-padrão para números e boleanos onde for necessário).
(a) $((\lambda x \cdot x * x)5)$
(b) $((\lambda y \cdot ((\lambda x \cdot x + y + z)3))2)$
(c) $((\lambda v \cdot (\lambda w \cdot w))((\lambda x \cdot x)(y(\lambda z \cdot z))))$

14.2 Avalie as expressões no exercício anterior usando redução-beta *lenta*. Você obtém os mesmos resultados?

14.3 Avalie as seguintes expressões, usando o seu interpretador Scheme:
(a) `(null? ())`
(b) `(null? '(a b c d e))`
(c) `(car '(a (b c) d e))`
(d) `(cdr '(a (b c) d e))`
(e) `(cadr '(a (b c) d e))`

14.4 Avalie a expressão Scheme `(sum 1 2 3 4 5)`, mostrando todos os passos da expansão da função soma dados neste capítulo.

14.5 Escreva uma função Scheme denominada `elements` que conta o número de elementos em uma lista; por exemplo: `(elements '(1 (2 (3) 4) 5 6))` é `6`, enquanto o tamanho da mesma lista é `4`.

14.6 No interpretador Clite da Seção 14.2.8, procure manualmente:

```
(m-expression '(plus (times (variable a) (variable b))
                     (value 2)) '((a 2) (b 4)))
```

14.7 Para o interpretador Clite discutido neste capítulo, implemente em Scheme o significado dos operadores relacionais e Boleano.

14.8 Para o interpretador Clite discutido neste capítulo, implemente em Scheme o significado da instrução de atribuição.

14.9 Acrescente uma implementação do operador unário `not` ao interpretador Clite.

14.10 (Trabalho de Equipe) Reescreva a sintaxe abstrata de Clite na forma de listas Scheme. Depois use essas definições para implementar um verificador de tipo para Clite em Scheme usando a implementação Java como modelo.

14.11 (Trabalho de Equipe) Reescreva a sintaxe abstrata de Clite na forma de listas Scheme. Depois use essas definições para implementar as semânticas para Clite em tempo de execução em Scheme usando a implementação Java como modelo.

14.12 Estenda o programa de diferenciação simbólica de maneira que ele diferencie funções com expoentes e também somas e produtos. Essa extensão deve se basear no seguinte conhecimento:

$$\frac{du^n}{dx} = nu^{n-1}\frac{du}{dx} \qquad \text{para inteiros } n > 0$$

14.13 Use o seu programa de diferenciação simbólica estendido para diferenciar as seguintes funções:
(a) $x^2 + 2x + 1$
(b) $(5x - 2y)/(x^2 + 1)$

14.14 Considere o problema de simplificar uma expressão algébrica, como o resultado da diferenciação simbólica. Nós sabemos que identidades matemáticas como $x + 0 = x$ e $1 \cdot x = x$ são usadas para simplificar expressões.
(a) Formule um conjunto de regras para simplificação, com base nas propriedades de 0 e 1 quando somados ou multiplicados com outra expressão.
(b) Escreva uma função Scheme que simplifique uma expressão algébrica arbitrária usando essas regras.
(c) Formule um conjunto de regras de simplificação com base na soma e na multiplicação de constantes.
(d) Escreva uma função Scheme que simplifique uma expressão algébrica arbitrária usando essas regras.
(e) Estenda as funções Scheme acima de maneira que as regras sejam aplicadas repetidamente até que não resulte mais nenhuma simplificação. *Dica*: aplique as simplificações e o teste para ver se a expressão "mais simples" é diferente da original.

14.15 Considere o problema backtracking das oito rainhas. Ative o tracing para as funções `try`, `trywh` e `tryone`.
(a) Mostre que não há solução quando `N` for 3.
(b) Mostre que, quando `N` é 5, a solução é encontrada sem backtracking.
(c) Encontra-se uma solução quando `N` é 6? O backtracking é usado?

14.16 Implemente o knight's tour em Scheme para um tabuleiro de tamanho 5 quando o cavaleiro começa no quadrado (1, 1). Para uma descrição do knight's tour, veja o Exercício 13.16.

14.17 Escreva um programa Scheme usando backtracking para resolver (facilmente) um quebra-cabeça Sudoku (**www.sudoku.com**).

14.18 Usando a solução generativa Haskell para o problema das oito rainhas (Seção 14.3.10) como modelo, escreva uma solução generativa para o problema das oito rainhas em Scheme.

14.19 Avalie as seguintes expressões usando o seu interpretador Haskell:
(a) `[1,2,3,4,5]!!2`
(b) `[1,2,3,4,5]!!5`
(c) `head [1,2,3,4,5]`
(d) `tail [1,2,3,4,5]`

14.20 Para o exemplo do catálogo telefônico discutido na Seção 14.3.7, avalie as seguintes funções, supondo que a lista `pb` tenha os valores iniciais mostrados lá.
(a) `addEntry pb "Jane" 1223345`
(b) `deleteEntry pb "Bob" 2770123`

14.21 Reescreva a função `deleteEntry` da Seção 14.3.7 de maneira que ela exclua todas as entradas para determinada pessoa. Por exemplo, se `pb` é conforme definido aqui, então a função

```
deleteEntry "Bob"
```

irá excluir *todas* as entradas de pb para a pessoa denominada "Bob", e não apenas uma.

14.22 Escreva uma função Haskell denominada `elements` que conte o número de elementos em uma lista. Ela deve produzir os mesmos resultados da função `length`.

14.23 Implemente a função de significado Haskell para um Bloco Clite, usando a seguinte definição matemática:

$$M(Block\ \text{b}, State\ \sigma) = \sigma \qquad \text{se b} = \phi$$
$$= M((Block)b_{2\ldots n}, M((Statement)b_1, \sigma)) \qquad \text{se b} = b_1 b_2 \ldots bn$$

14.24 No interpretador Haskell discutido na Seção 14.3.8, implemente as funções de significado para os operadores Clite aritméticos, relacionais e boleanos.

14.25 Acrescente uma implementação do operador unário Clite `!` para a versão Haskell do interpretador Clite.

14.26 Após completar os dois exercícios anteriores, deduza todos os passos na aplicação da função Haskell `eval` à Expressão `y+2` no estado `{<x,5>, <y,3>, <z,1>}`. Isto é, mostre todas as etapas na dedução do seguinte resultado:

```
eval(Binary "+" (Vary) (Lit (Intval 2)))[("x",(Intval 5)),
             ("y",(Intval 3)),("z",(Intval 1)))]
= (Value 5)
```

14.27 Dê uma definição repetitiva da função Haskell `length` que calcule o número de entradas em uma lista.

14.28 Considere a seguinte implementação Haskell (correta, mas ineficiente) da função familiar Fibonacci:

```
fibSlow n
   | n == 0 = 1
   | n == 1 = 1
   | n > 1  = fibSlow(n-1) + fibSlow(n-2)
```

A correção dessa função é aparente, já que ela é uma codificação direta da definição matemática familiar.

$$fib(0) = 1$$
$$fib(1) = 1$$
$$fib(n) = fib(n-1) + fib(n-2) \text{ if } n > 1$$

(a) Mas a eficiência dessa função é suspeita. Experimente executar `fibSlow(25)` e depois `fibSlow(50)` no seu sistema e veja quanto tempo levam os cálculos. O que causa essa ineficiência?

(b) Uma definição alternativa da função *fib* pode ser enunciada da seguinte maneira: defina uma função `fibPair` que gera uma tupla de dois elementos contendo o n-ésimo número de Fibonacci e seu sucessor. Defina outra função `fibNext` que gera a próxima tupla a partir da atual. Depois, a própria função Fibonacci, que de uma forma otimista chamamos de `fibFast`,

é definida selecionando o primeiro membro do n-ésimo `fibPair`. Em Haskell, isso é escrito assim:

```
fibPair n
   | n == 0   = (1,1)
   | n > 0    = fibNext(fibPair(n-1))
fibNext (m,n)= (n,m+n)
fibFast n = fst(fibPair(n))
```

Experimente rodar a função `fibFast` para calcular o 25º e o 50º números de Fibonacci. Ela deverá ser consideravelmente mais eficiente do que `fibSlow`. Explique.

14.29 Reescreva o formatador Haskell para diferenciação simbólica que remova quaisquer parênteses desnecessários. *Dica*: evite introduzi-los no início.

14.30 Escreva um simplificador de expressão Haskell para diferenciação simbólica (veja o Exercício 14.14).

14.31 Usando a solução do problema das oito rainhas como modelo, escreva um programa Haskell para resolver (facilmente) um quebra-cabeça Sudoku (**www.sudoku.com**).

14.32 (Trabalho de Equipe) Dê uma sintaxe abstrata de Clite completa como tipos de dados repetitivos Haskell. Depois use essas definições para implementar um verificador de tipo para Clite em Haskell usando como modelo a implementação Java.

14.33 (Trabalho de Equipe) Dê uma sintaxe abstrata de Clite completa como tipos de dados repetitivos Haskell. Depois use essas definições para implementar as funções de significado para Clite em Haskell usando como modelo a implementação Java.

14.34 (Trabalho de Equipe) Desenvolva uma sintaxe concreta e abstrata para um pequeno subconjunto de Scheme ou Haskell. Depois use essas definições para implementar uma semântica em tempo de execução para Scheme ou Haskell usando Java como linguagem de implementação.

Programação Lógica

15

"P: Quantas pernas tem um cachorro, se chamarmos sua cauda de perna?
R: Quatro. Chamar uma cauda de perna não a transforma em uma perna."

Abraham Lincoln

VISÃO GERAL DO CAPÍTULO

A programação lógica (declarativa) surgiu como um paradigma distinto nos anos 70. A programação lógica é diferente dos outros paradigmas porque ela requer que o programador declare os objetivos da computação, em vez dos algoritmos detalhados por meio dos quais esses objetivos podem ser alcançados. Os objetivos são expressos como uma coleção de asserções, ou regras, sobre os resultados e as restrições da computação. Por essa razão, a programação lógica, às vezes, é chamada programação *baseada em regras*.

As aplicações de programação declarativa se classificam em dois domínios principais: inteligência artificial e acesso de informações em bancos de dados. No campo da inteligência artificial, Prolog tem sido influente. Alguns subcampos da inteligência artificial usam outras linguagens declarativas, como MYCIN, para modelar sistemas especializados. Na área de bancos de dados, a Structured Query Language (SQL) tem sido bem popular.

Para que o assunto seja compreendido com profundidade, este capítulo focaliza apenas a programação lógica com Prolog, e estuda suas aplicações em processamento de linguagem natural e na solução de problemas.

Duas características interessantes e diferenciadas dos programas lógicos são *não-determinismo* e *backtracking*. Um programa lógico não-determinístico pode encontrar várias soluções para um problema em vez de apenas uma, como seria normal em outros domínios de programação. Além disso, o mecanismo backtracking que possibilita o não-determinismo está dentro do interpretador Prolog, e, portanto, é implícito em todos os programas Prolog. Ao contrário, o uso de outras linguagens para escrever programas backtracking requer que o programador defina o mecanismo de backtracking explicitamente, como vimos na Seção 13.4.2. Neste capítulo, veremos o poder do backtracking e do não-determinismo.

15.1 LÓGICA E CLÁUSULAS DE HORN

Um programa lógico expressa as especificações para soluções de problemas com o uso de expressões em lógica matemática. Esse estilo evoluiu a partir das necessidades dos pesquisadores em processamento de linguagem natural e na prova automática de teorema. As linguagens de programação convencionais não são particularmente bem adequadas para as necessidades desses pesquisadores. No entanto, escrever a especificação de um teorema ou uma gramática (como a gramática BNF, usada para definir a sintaxe Clite) como uma expressão lógica formal proporciona um veículo eficaz para estudar o processo de prova de teorema e análise de linguagem natural em um cenário experimental de laboratório.

Assim, a lógica proposicional e predicativa (veja o Apêndice B) proporciona os fundamentos formais para a programação lógica. A cláusula de Horn é uma variante particular de lógica predicativa que está por trás da sintaxe da Prolog.

> **Definição**: Uma *cláusula de Horn* tem uma parte mais importante h, que é um atributo, e um corpo, que é uma lista de atributos $p_1, p_2, \ldots, p_n$.

Cláusulas de Horn são escritas no seguinte estilo:

$$h \leftarrow p_1, p_2, \ldots, p_n$$

Isso significa que h é *verdadeiro (true)* somente se p_1, p_2, . . . e p_n forem simultaneamente *true*.

Por exemplo, suponha que queremos capturar a idéia de que está nevando em alguma cidade C somente se houver precipitação na cidade C e a temperatura na cidade C estiver no ponto de congelamento. Podemos escrever isso como a seguinte cláusula de Horn:

$$nevando(C) \leftarrow precipitação(C), congelando(C)$$

Há uma correspondência limitada entre cláusulas de Horn e predicados. Por exemplo, a cláusula de Horn acima pode ser escrita de forma equivalente ao seguinte predicado:

$$precipitação(C) \wedge congelando(C) \supset nevando(C)$$

Essa expressão é logicamente equivalente a um dos seguintes predicados, que usa as propriedades dos predicados resumidas no Apêndice B:

$$\neg(precipitação(C) \wedge congelando(C)) \vee nevando(C)$$
$$\neg precipitação(C) \vee \neg congelando(C) \vee nevando(C)$$

Portanto, qualquer cláusula de Horn pode ser escrita de forma equivalente como um predicado.

Infelizmente, o inverso não é verdadeiro; nem todos os predicados podem ser traduzidos em cláusulas de Horn. A seguir está um procedimento de seis etapas (Clocksin e Mellish, 1997, Capítulo 10) que, sempre que possível, traduz um predicado p em uma cláusula de Horn.

1 Elimine as implicações de p, usando a propriedade implicação na Tabela B.5.

2 Mova a negação para dentro em p, usando propriedades de Morgan e de qualificação, de forma que somente termos individuais sejam negados.

3 Elimine quantificadores existenciais de p, usando uma técnica chamada *skolemization.* Aqui, a variável existencialmente quantificada é substituída por uma única constante. Por exemplo, a expressão $\exists x P(x)$ é substituída por $P(c)$, na qual c é uma constante escolhida arbitrariamente no domínio de x. Para cada quantificador existencial desses, deve ser escolhida uma constante c diferente.

4 Mova todos os quantificadores universais para o início de p; desde que não haja conflitos de nome, esta etapa não muda o significado de p. Supondo que todas as variáveis estejam universalmente quantificadas, podemos cancelar os quantificadores sem mudar o significado do predicado.

5 Use as propriedades distributiva, associativa e comutativa da Tabela B.5 para converter p para a *forma normal conjuntiva*. Nessa forma, os operadores conjunção e disjunção estão aninhados não mais do que dois níveis abaixo, com conjunções no nível mais alto.

6 Converta as disjunções e implicações encaixadas, usando a propriedade implicação. Se cada uma dessas implicações tiver um único termo à sua direita, então cada uma delas pode ser reescrita como uma série de cláusulas de Horn equivalentes a p.

Para ilustrar esse procedimento, considere a transformação do seguinte predicado para a forma conjuntiva normal:

$$\forall x(\neg literate(x) \supset (\neg writes(x) \wedge \neg \exists y(reads(x, y) \wedge book(y))))$$

Aplicando o passo 1, remove-se a implicação, deixando:

$$\forall x(literate(x) \vee (\neg writes(x) \wedge \neg \exists y(reads(x, y) \wedge book(y))))$$

O passo 2 move as negações de forma que elas fiquem adjacentes aos termos individuais:

$$\forall x(literate(x) \vee (\neg writes(x) \wedge \forall y(\neg(reads(x, y) \wedge book(y)))))$$
$$= \forall x(literate(x) \vee (\neg writes(x) \wedge \forall y(\neg reads(x, y) \vee \neg book(y))))$$

Como não há quantificadores existenciais, a *skolemization* não é necessária. O passo 4 move todos os quantificadores para a esquerda e então cancela-os, resultando:

$$\forall x \forall y(literate(x) \vee (\neg writes(x) \wedge (\neg(reads(x, y) \wedge book(y)))))$$
$$= literate(x) \vee (\neg writes(x) \wedge (\neg reads(x, y) \vee \neg book(y)))$$

Agora podemos converter isso para a forma conjuntiva da seguinte maneira:

$$literate(x) \vee (\neg writes(x) \wedge (\neg reads(x, y) \vee \neg book(y)))$$
$$= (literate(x) \vee \neg writes(x)) \wedge (literate(x) \vee \neg reads(x, y) \vee \neg book(y))$$
$$= (\neg writes(x) \vee literate(x)) \wedge (\neg reads(x, y) \vee \neg book(y) \vee literate(x))$$

Esses dois conjuntos agora convertem de volta para as implicações:

$$(\neg writes(x) \vee literate(x)) \wedge (\neg reads(x, y) \vee \neg book(y) \vee literate(x))$$
$$= (writes(x) \supset literate(x)) \wedge (\neg(\neg reads(x, y) \vee \neg book(y)) \supset literate(x))$$
$$= (writes(x) \supset literate(x)) \wedge ((reads(x, y) \vee book(y)) \supset literate(x))$$

que são equivalentes às seguintes cláusulas de Horn:

$$literate(x) \leftarrow writes(x)$$
$$literate(x) \leftarrow reads(x, y), book(y)$$

Infelizmente, a conversão de um predicado para uma forma conjuntiva normal nem sempre garante uma série de cláusulas de Horn equivalentes. Considere o seguinte predicado, que representa a afirmação "Toda pessoa instruída lê ou escreve".

$$\forall x(literate(x) \supset reads(x) \vee writes(x))$$

que se reduz à seguinte forma clausal:

$$\neg literate(x) \vee reads(x) \vee writes(x)$$

Mas isso se converte na seguinte implicação:

$$literate(x) \supset reads(x) \vee writes(x)$$

que não tem um único termo à direita. Portanto, não há uma cláusula de Horn equivalente para esse predicado.

15.1.1 Resolução e Unificação

É denominado resolução o ato de se fazer uma única inclusão a partir de um par de cláusulas de Horn. O princípio da resolução é similar à idéia da transitividade em álgebra.

> **Definição**: Quando aplicada às cláusulas de Horn, a *resolução* diz que se h é a cabeça de uma cláusula de Horn e ela corresponde a um dos termos de uma outra cláusula de Horn, então aquele termo pode ser substituído por h.

Em outras palavras, se nós temos as cláusulas

$$h \leftarrow terms$$
$$t \leftarrow t_1, h, t_2$$

então podemos resolver a segunda cláusula para $t \leftarrow t_1, terms, t_2$. Por exemplo, considere as seguintes cláusulas:

$$speaks(Mary, English)$$
$$talkswith(X, Y) \leftarrow speaks(X, L), speaks(Y, L), X \neq Y$$

A primeira é uma cláusula de Horn com uma lista vazia de termos, assim ela é incondicionalmente *true*. Portanto, a resolução nos permite deduzir o seguinte:

$$talkswith(Mary, Y) \leftarrow speaks(Mary, English), speaks(Y, English), Mary \neq Y$$

com a hipótese de que às variáveis X e L são atribuídos os valores "Mary" e "English" na segunda regra. A resolução, portanto, ajuda a chegar às conclusões.

Definição: A atribuição de valores a variáveis durante a resolução é chamada *instanciação.*

Definição: *Unificação* é um processo de correspondência de padrões que determina que instanciações, em particular, podem ser feitas a variáveis ao mesmo tempo em que se faz uma série de resoluções simultâneas.

A unificação é repetitiva, assim ela eventualmente encontra todas as instanciações possíveis para as quais podem ser adotadas resoluções. Ilustramos a unificação em detalhe na Seção 15.2.

15.2 PROGRAMAÇÃO LÓGICA EM PROLOG

Prolog é a linguagem principal usada em programação lógica. O desenvolvimento de Prolog foi baseado em dois poderosos princípios descobertos por Robinson (Robinson, 1965) chamados *resolução* e *unificação*. A Prolog surgiu em 1970, resultado do trabalho de Colmerauer, Rousseau e Kowalski (Kowalski e Kuehner, 1970), e tem sido a principal linguagem de programação lógica até os dias atuais. As aplicações da programação lógica estão espalhadas pelas áreas de processamento de linguagem natural, raciocínio automático e prova de teoremas, pesquisa em bases de dados e sistemas especializados.

15.2.1 Elementos de um Programa Prolog

Os programas Prolog são feitos a partir de termos, que podem ser constantes, variáveis ou estruturas. Uma *constante* é um átomo (tipo `the`, `zebra`, `'Bob'`, e `'.'`) ou um inteiro não-negativo (como 24). Uma *variável* é uma série de letras (`A-Z, a-z, _`) que deve começar com uma letra maiúscula (como `Bob`).[1] Uma *estrutura* é um predicado com zero ou mais argumentos, escritos em notação funcional. Por exemplo, veja algumas estruturas Prolog:

```
n(zebra)
speaks(Who, russian)
np(X, Y)
```

O número de argumentos é chamado *aridade* da estrutura (1, 2 e 2, nestes exemplos).

Fatos e regras Prolog são realizações da idéia formal das cláusulas de Horn, como foram introduzidas na Seção 15.1. Um *fato* é um termo seguido de um ponto (.) e é similar a uma cláusula de Horn sem o lado direito; uma variável não pode ser um fato. Uma *regra* é um termo seguido de :- e uma série de termos separados por vírgulas que termina em um ponto (.). Uma regra tem a seguinte forma:

`term:-`$\texttt{term}_1$`,` $\texttt{term}_2$`,...,` $\texttt{term}_n$`.`

Isso é equivalente à cláusula de Horn:

$term \leftarrow term_1, term_2, \ldots, term_n$

Regras são interpretadas como asserções "somente se", com a vírgula desempenhando o papel de operador lógico "and". Assim, a forma acima afirma que `term` é *verdadeiro* (*true)* somente se $\texttt{term}_1$, $\texttt{term}_2$, ... e $\texttt{term}_n$ forem simultaneamente *verdadeiros*.

1. Uma constante não pode começar com uma letra maiúscula a menos que ela seja colocada entre aspas.

Um programa Prolog é uma série de fatos e regras. Veja aqui um exemplo:

```
speaks(allen, russian).
speaks(bob, english).
speaks(mary, russian).
speaks(mary, english).
talkswith(Person1, Person2) :- speaks(Person1,L),
  speaks(Person2,L), Person1 \= Person2.
```

Esse programa afirma quatro fatos: que `allen` e `mary` falam `russian` e `bob` e `mary` falam `english`. Ele também tem uma regra que define a relação `talkswith` entre duas pessoas, que é true exatamente quando ambas falam a mesma língua, representada pela variável `L`, e elas são pessoas diferentes. O operador \= especifica que `Person1` não pode ser a mesma pessoa (igual a) `Person2`.

Uma regra Prolog *tem sucesso* quando há instanciações (que são atribuições temporárias) de suas variáveis para as quais todos os termos à direita do operador :- simultaneamente têm sucesso para aquelas atribuições. Caso contrário, dizemos que a regra *falhou*. Um fato sempre tem sucesso; isto é, ele é universalmente *verdadeiro*. Por exemplo, a regra no exemplo acima tem sucesso para as instanciações

```
Person1 = allen
Person2 = mary
```

pois há uma instanciação da variável `L` (=`russian`) para a qual os três predicados:

```
speaks(allen, L)
speaks(mary, L)
allen \= mary
```

têm sucesso simultaneamente. No entanto, essa regra falha para outras instanciações, como, por exemplo:

```
Person1 = allen
Person2 = bob
```

pois elas não compartilham de uma instanciação comum da variável `L`.

Para fazer um exercício com um programa Prolog, podemos escrever *queries* (consultas) em resposta ao sinal de prompt ?- de Prolog. Aqui está uma simples *consulta* que faz a pergunta "Quem fala russo?"

```
?- speaks(Who, russian).
```

Em resposta, o programa Prolog tenta atender a uma query procurando um fato ou uma série de aplicações fato/regra que responderá à query, ou seja, uma atribuição de valores para as variáveis na query que faz um fato ou uma regra ter êxito, no sentido de correspondência de padrões.

Carregando e Executando Programas Carregar arquivos com definições de função Prolog tem o mesmo efeito de digitá-las diretamente no interpretador Prolog. Por exemplo,

para carregar o arquivo denominado `diff` que contém a função `d`, podemos usar o comando:

```
?- consult(diff).
```

Fatos e regras Prolog são usualmente digitados em um arquivo separado, para o qual é usado o comando `consult` para carregar aquele arquivo no interpretador. O interpretador então afirma cada um dos fatos e das regras que estão definidos no programa. Após o carregamento do programa, podem ser feitas queries ao interpretador na forma de asserções com variáveis, e o interpretador tentará responder a elas.

Aqui estão algumas diretrizes gerais para escrever programas Prolog:

1. Identificadores que começam com uma letra maiúscula ou um caractere sublinha são tomados como variáveis; todos os outros identificadores são tratados como constantes.
2. Não deve haver nenhum espaço entre o nome do predicado e o parêntese esquerdo que abre sua lista de argumentos.
3. Todos os fatos, as regras e as queries devem terminar com um ponto.
4. Um arquivo de programa deve terminar com um fim de linha.

Por exemplo, suponha que temos o seguinte arquivo de programa Prolog chamado `speaks`:

```
speaks(allen, russian).
speaks(bob, english).
speaks(mary, russian).
speaks(mary, english).
talkswith(Person1, Person2) :-
     speaks(Person1, L), speaks(Person2, L), Person1 \= Person2.
```

Então podemos carregar esse programa com:

```
?- consult(speaks).
speaks compiled, 0.00 sec, 1,312 bytes.

Yes
```

A resposta `Yes` diz que o programa Prolog verificou com sucesso a sintaxe e carregou o arquivo, assim podemos ir em frente e propor queries.

Unificação, Ordem de Avaliação e *Backtracking* Considere a seguinte query:

```
?- speaks(Who, russian).
```

Ao procurar uma solução, o programa Prolog examina todos os fatos e as regras que tiverem uma parte importante (head) que corresponda a uma função mencionada na query (neste caso, `speaks`). Se houver mais de uma (neste caso, há quatro), ele irá considerá-las na ordem em que elas aparecerem no programa. Como `russian` é uma constante, a única variável a ser resolvida é a variável `Who`. Usando o primeiro fato no programa, Prolog responde com:

```
Who = allen
```

já que aquela atribuição à variável `Who` faz o primeiro fato ter sucesso. Nesse ponto, o usuário pode querer saber se há outras maneiras de satisfazer à mesma query, na qual é digitado um ponto-e-vírgula (;). Prolog continua sua pesquisa por intermédio do programa, relatando o próximo sucesso, se houver um. Quando não há mais instanciações bem-sucedidas para a variável `Who`, Prolog finalmente responde `No` e o processo pára. Aqui está uma interação completa:

```
?- speaks(Who, russian).
Who = allen ;
Who = mary ;
No
```

Outro tipo de query que esse programa pode manipular é aquela que faz perguntas como "Bob conversa com Allen?" ou "Quem conversa com Allen?" ou "Que pessoas falam umas com as outras?". Essas perguntas podem ser escritas na forma das seguintes queries Prolog, com as respostas às duas primeiras mostradas abaixo delas:

```
?- talkswith(bob, allen).
No
?- talkswith(Who, allen).
Who = mary ;
No
?- talkswith(P1, P2).
```

Para ver como essas queries são satisfeitas, precisamos observar como a regra para `talkswith` é avaliada. Qualquer query da forma `talkswith(X, Y)` apela para aquela regra, que pode ser satisfeita somente se houver uma instanciação comum para as variáveis `X`, `Y` e `L`, para as quais `speaks(X, L)`, `speaks(Y, L)` e `X\=Y` são simultaneamente satisfeitas. Esses três termos às vezes são chamados de *subobjetivos* do objetivo principal `talkswith(X, Y)`.

Prolog tenta satisfazer aos subobjetivos em uma regra da esquerda para a direita, de forma que é feita primeiro uma pesquisa dos valores de `X` e `L` para a qual `speaks(X, L)` é satisfeita. Uma vez encontrados tais valores, esses mesmos valores serão usados onde suas variáveis aparecerem na pesquisa para satisfazer aos subobjetivos adicionais para aquela regra, como `speaks(Y, L)` e `X\=Y`.

O processo de tentar satisfazer a primeira query acima está diagramado na *árvore de pesquisa* mostrada na Figura 15.1.

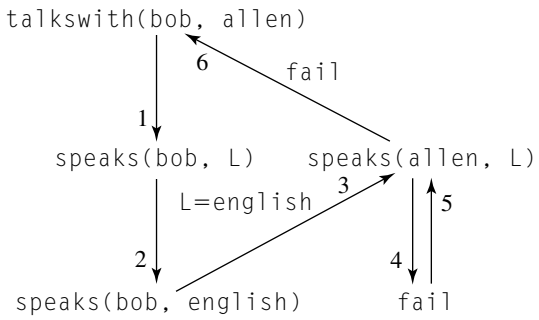
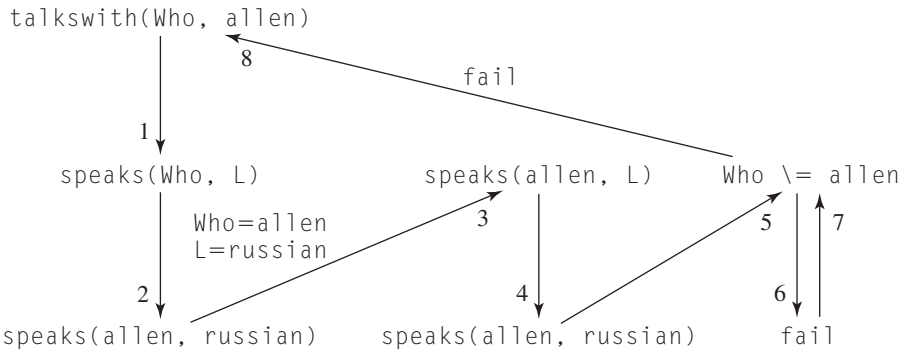


Figura 15.1
Tentando Satisfazer a Query `talkswith (bob, allen)`



| Figura 15.2 Primeira Tentativa de Satisfazer a Query `talkswith(Who, allen)`

Esse processo falha porque a única instanciação de `L` que satisfaz `speaks(bob, L)` não satisfaz simultaneamente `speaks(allen, L)` para esse programa. Os números atribuídos às setas nesse diagrama indicam a ordem na qual são tentados os subobjetivos.

A resposta `no` indica que o programa Prolog não pode encontrar mais soluções para uma query. Em geral, o programa Prolog opera sob aquilo que chamamos *suposição de um mundo fechado,* o que significa que qualquer coisa que não tenha sido informada a ele não é *verdade*. Nesse exemplo, o mundo fechado contém apenas fatos que foram declarados sobre pessoas específicas que falam línguas específicas, e nada mais.

O processo para satisfazer a segunda query acima é um pouco mais complexo. Ele falha na seqüência mostrada na Figura 15.2. Embora os subobjetivos `speaks(Who,L)` e `speaks(allen,L)` sejam satisfeitos pela instanciação `Who=allen`, o terceiro subobjetivo falha, pois `allen\=allen` falha.

Uma vez ocorrida essa falha, o processo retrocede (backtracks) para o passo denominado 2 na figura e procura por outras instanciações de `Who` e `L` que satisfarão aos três subobjetivos simultaneamente. Assim, o processo eventualmente continua com as instanciações `Who=mary` e `L=russian`, mas nenhuma outra.

Pesquisa em Base de Dados – A Árvore Genealógica Pode ficar evidente que a Prolog é bem adequada para solução de problemas que requeiram pesquisas em uma base de dados. Na verdade, o desenvolvimento de sistemas especializados durante os anos 70 e 80 foi facilitado em grande parte por programas Prolog. O programa `speaks` nesta seção pode ser visto como um programa muito simples para pesquisa de base de dados, em que os primeiros quatro fatores representam a base de dados e a regra representa as restrições sob as quais pode ocorrer uma pesquisa bem-sucedida. Na Figura 15.3 aparece um exemplo ligeiramente diferente.

Esse programa modela um pequeno grupo de pessoas cuja family "tree" é mostrada na Figura 15.4.[2] O diagrama confirma que "pai" relaciona duas pessoas em níveis adjacentes

2. Rigorosamente falando, essa não é realmente uma estrutura em árvore, pois alguns nós têm mais de um pai e há mais de um nó "raiz" em potencial. Mas é uma "árvore genealógica" no sentido coloquial.

```
parent(A,B) :- father(A,B).
parent(A,B) :- mother(A,B).
grandparent(C,D) :- parent(C,E), parent(E,D).

mother(mary, sue).
mother(mary, bill).
mother(sue, nancy).
mother(sue, jeff).
mother(jane, ron).

father(john, sue).
father(john, bill).
father(bob, nancy).
father(bob, jeff).
father(bill, ron).
```

| Figura 15.3 Uma Árvore Genealógica Parcial

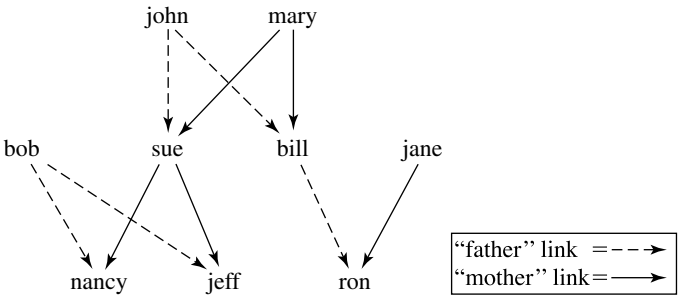
na árvore e "avô" (grandparent) relaciona duas pessoas que estão distantes entre si dois níveis.

Podemos consultar essa base de dados de várias maneiras para fazer diferentes perguntas. Por exemplo, a pergunta "Quem é um avô de Ron" pode ser colocada assim:

```
?- grandparent(Who, ron).
```

Podem ser definidas relações adicionais para possibilitar uma variedade mais ampla de perguntas a serem feitas. Por exemplo, a relação `sibling` pode ser definida entre duas pessoas diferentes que compartilham o mesmo pai:

```
?- sibling(X, Y) :- parent(W, X), parent(W, Y), X \= Y.
```



| Figura 15.4 Uma Pequena "Árvore" Genealógica

Deve-se notar que Prolog sofre da *síndrome do mundo fechado*. Um programa Prolog somente *sabe* aquilo que foi dito a ele. Neste exemplo, um `father` (pai) poderia facilmente ser tanto uma mulher quanto um homem. O sistema não tem senso de gênero; ele não entende que um pai biológico deve ser um homem nem que os pais de uma pessoa devem ser distintos.

Listas A estrutura básica de dados em programação Prolog é a lista, que é escrita como uma série de termos separados por vírgulas e envolvidos por colchetes `[` e `]`. Sentenças usualmente são representadas como listas de átomos (coisas minúsculas), como no exemplo a seguir:

```
[the, giraffe, dreams]
```

A lista vazia é representada por `[]`, enquanto uma entrada do tipo *não importa* (*don't care)* em uma lista é representada por uma sublinha (_). Os exemplos a seguir representam listas de um, dois e três elementos, respectivamente.

```
[X]
[X, Y]
[_, _, Z]
```

O elemento `Head` (primeiro) de uma lista é distinguido dos demais elementos por uma barra vertical. Assim,

```
[Head | Tail]
```

representa uma lista cujo primeiro elemento é `Head` e cujos demais elementos formam a lista `[Tail]`, muitas vezes chamado de *final* (*tail*) da lista.

Aqui está uma função Prolog simples que define o encadeamento de duas listas para juntá-las e formar uma só. Os dois primeiros argumentos para essa função representam as duas listas que estão sendo encadeadas, e a terceira representa o resultado.

```
append([], X, X).
append([Head | Tail], Y, [Head | Z]) :- append(Tail, Y, Z).
```

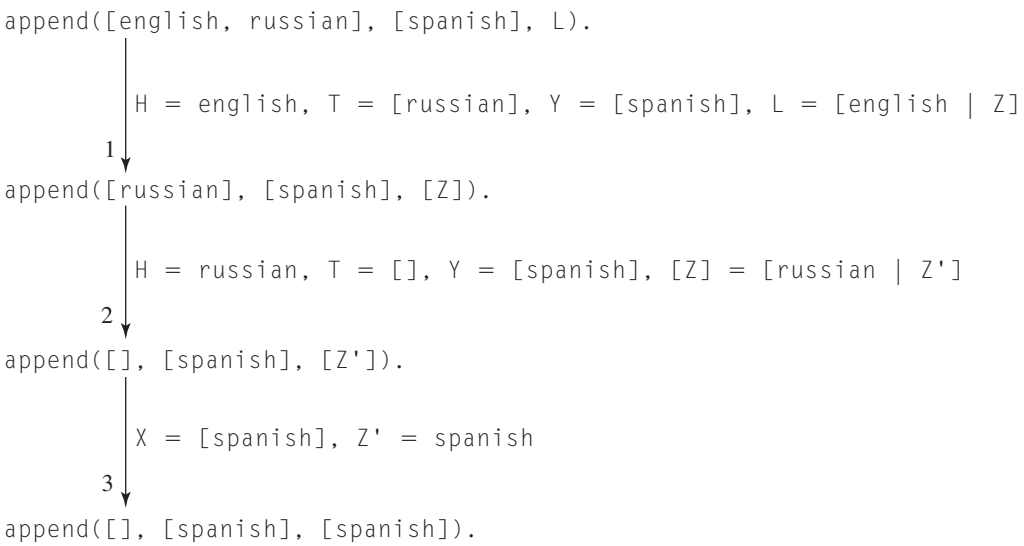
Essa definição repetitiva um tanto estranha tem duas partes. O "base case" é definido pela primeira linha, que simplesmente afirma que a lista vazia encadeada com qualquer outra lista retorna aquela outra lista como resultado. O case repetitivo, definido pela segunda linha, diz que se `Z` é o resultado do encadeamento de listas `Tail` e `Y`, então o encadeamento de qualquer nova lista `[Head | Tail]` com `Y` dá o resultado `[Head | Z]`.

É importante entender a dinâmica de execução para essa função, pois essa forma de definição repetitiva ocorre freqüentemente em Prolog. Considere a query

```
?- append([english, russian], [spanish], L).
```

que produziria a lista `L = [english, russian, spanish]`. A Figura 15.5 traz uma árvore de pesquisa parcial que rastreia as instanciações de `H`, `T`, `X`, `Y` e `Z` à medida que esse resultado é desenvolvido.

As duas primeiras chamadas usam a regra repetitiva, que separa a head `H` do primeiro argumento e chama de volta `append` com uma lista mais curta como primeiro argumento.



| Figura 15.5 Árvore de Pesquisa Parcial para `append([english, russian], [spanish], L)`

A chamada final usa o *base case*, que força a instanciação da variável `X = [spanish]` e `Z' = spanish`.[3] Desse modo, o resultado `[H | Z']` na segunda chamada é resolvido como `[russian, spanish]` e identificado com a lista `Z` na primeira chamada. Finalmente, a lista `L` é determinada para `[english, russian, spanish]`, usando esse valor que acaba de ser descoberto para `Z` como sendo sua cauda.

Aqui estão mais três funções com listas que são muito úteis em programação Prolog. As duas primeiras, chamadas *prefixo* e *sufixo*, significam exatamente o que seus nomes sugerem. X é um prefixo de Z se houver um Y que pudermos acrescentar a X para fazer Z. A definição de Y como sufixo de Z é similar.

```
prefix(X, Z) :- append(X, Y, Z).
suffix(Y, Z) :- append(X, Y, Z).
```

A terceira função útil define repetidamente a noção de participante em uma lista. X é um membro de uma lista se ele for idêntico ao primeiro elemento ou a um membro do fim (*tail*) da lista.

```
member(X, [X | _]).
member(X, [_ | Y]) :- member(X, Y).
```

Observe aqui o uso da notação "não importa" (*don't care*) em partes da definição que não afetam a definição. Na primeira linha, não nos importamos em saber como é o resto da lista se `X` for idêntico ao primeiro membro. Na segunda, não nos importamos com o que está no início (head) da lista se `X` for um membro do fim (tail).

3. O uso da plica (') nessa análise, embora ela não seja um caractere permitido em um nome de variável Prolog, ajuda a distinguir o uso de uma variável em uma regra em um nível de repetição do uso da mesma variável em outro nível.

Para ilustrar, suponha que temos as seguintes atribuições de lista em Prolog:

```
L = [my, dog, has, many, fleas]
M = [many, fleas, spoil, the, picnic]
```

Então as seguintes queries Prolog produzem as respectivas respostas mostradas abaixo de cada uma delas:

```
? - prefix(Answer, L).
Answer = [];
Answer = [my];
Answer = [my, dog];
...
Answer = [my, dog, has, many, fleas];
No
? - suffix(Answer, L), prefix(Answer, M).
Answer = [];
Answer = [many, fleas];
No
? - member(spoil, L).
No
? - member(spoil, M).
Yes
```

15.2.2 Aspectos Práticos de Prolog

Na prática, Prolog não é uma linguagem de programação lógica totalmente "pura". Em particular, ela tem algumas características projetadas para tornar a programação mais eficiente e prática. Nesta seção, discutimos várias características importantes, entre elas: o comando "cut," o operador `is` e a função `assert`.

Tracing Muitas implementações Prolog proporcionam os atributos `trace` e `untrace`, que são usados para ligar e desligar o tracing de outros atributos. Como pode haver diferentes atributos com o mesmo nome, e diferentes aridades (número de argumentos), o trace espera que o nome de um atributo seja seguido de uma barra diagonal e sua aridade. Por exemplo, considere o atributo `factorial` definido da seguinte forma:

```
factorial(0, 1).
factorial(N, Result) :- N > 0, M is N - 1,
              factorial(M, SubRes), Result is N * SubRes.
```

Esse atributo define a função fatorial repetidamente, com a primeira linha definindo o base case ($0! = 1$) e a segunda e a terceira linhas definindo o case repetitivo ($n! = n(n-1)!$). Note que precisamos introduzir a variável intermediária M para forçar a avaliação de $N - 1$ antes da chamada repetitiva.

Para rastrear (trace) essa função, podemos fazer o seguinte:

```
?- trace(factorial/2).
factorial/2: call redo exit fail

Yes
?- factorial(4, X).
Call: (  7) factorial(4, _G173)
Call: (  8) factorial(3, _L131)
Call: (  9) factorial(2, _L144)
Call: ( 10) factorial(1, _L157)
Call: ( 11) factorial(0, _L170)
Exit: ( 11) factorial(0, 1)
Exit: ( 10) factorial(1, 1)
Exit: (  9) factorial(2, 2)
Exit: (  8) factorial(3, 6)
Exit: (  7) factorial(4, 24)
X = 24
```

Na primeira chamada, o primeiro argumento está ligado a 4, e o segundo está ligado a uma variável anônima (`_G173`). Esse mesmo padrão é repetido para cada uma das próximas chamadas repetitivas até que ocorra o base case, e ao segundo argumento `_L170` seja finalmente atribuído o valor 1. Agora a repetição se desenvolve, atribuindo valores intermediários às variáveis anônimas e fazendo as multiplicações.

O atributo `listing`, como em `listing(factorial/2)`, mostrará todos os fatos e as regras correntes para o argumento de predicado:

```
?- listing(factorial/2).
factorial(0, 1).
factorial(A, B) :-
        A>0,
        C is A-1,
        factorial(C, D),
        B is A*D.
Yes
```

O atributo `listing` sem argumentos lista todas as funções do programa consultado correntemente.

***Cut* e Negação** Prolog tem uma função especial chamada *cut,* que força a avaliação de uma série de subobjetivos no lado direito de uma regra que não será julgada se o lado direito tiver sucesso uma vez. A função cut é escrita inserindo-se um ponto de exclamação (!) como subobjetivo no lugar em que a interrupção deve ocorrer.

Para ilustrar, considere o seguinte programa, que executa o algoritmo de *bubble sort* em uma lista:

```
?- bsort([5,2,3,1], Ans).
Call: (  7) bsort([5, 2, 3, 1], _G221)
Call: (  8) bsort([2, 5, 3, 1], _G221)
Call: (  9) bsort([2, 3, 5, 1], _G221)
Call: ( 10) bsort([2, 3, 1, 5], _G221)
Call: ( 11) bsort([2, 1, 3, 5], _G221)
Call: ( 12) bsort([1, 2, 3, 5], _G221)
Redo: ( 12) bsort([1, 2, 3, 5], _G221)
Exit: ( 12) bsort([1, 2, 3, 5], [1, 2, 3, 5])
Exit: ( 11) bsort([2, 1, 3, 5], [1, 2, 3, 5])
Exit: ( 10) bsort([2, 3, 1, 5], [1, 2, 3, 5])
Exit: (  9) bsort([2, 3, 5, 1], [1, 2, 3, 5])
Exit: (  8) bsort([2, 5, 3, 1], [1, 2, 3, 5])
Exit: (  7) bsort([5, 2, 3, 1], [1, 2, 3, 5])

Ans = [1, 2, 3, 5] ;

No
```

| Figura 15.6 ***Trace*** **de** `bsort` **para a Lista [5, 2, 3, 1]**

```
bsort(L, S) :- append(U, [A, B | V], L),
               B < A, !,
               append(U, [B, A | V], M),
               bsort(M, S).
bsort(L, L).
```

Esse programa primeiro divide uma lista `L` encontrando duas sublistas `U` e `[A, B | V]`, para as quais `B < A` é *verdadeiro*. Uma vez encontrada essa partição, é formada a lista `M` acrescentando as sublistas `U` e `[B, A | V]` e depois repetidamente, aplicando o bubble sort para formar a nova lista `S`.

Esse processo se repete até que não se possa encontrar mais partições de `L`:, até que não haja nenhuma sublista `[A, B | V]` de `L` na qual `B < A`. Essa é uma maneira compacta de dizer que a lista `L` está ordenada. Nesse ponto, a única regra aplicável que resta é `bsort(L,L)`, que retorna a lista ordenada como resposta. A Figura 15.6 mostra um *trace* do programa.

Se a instrução *cut* não estivesse presente nesse programa, o processo de pesquisa teria continuado com um `Redo` da regra no nível 11, pois o programa Prolog iria procurar todas as soluções:

```
Redo: ( 11) bsort([2, 1, 3, 5], _G221)
```

e isso levaria à primeira de uma série de respostas incorretas. A função *cut* é, portanto, útil quando queremos interromper o processo de pesquisa após encontrar a primeira série de instanciações para as variáveis no lado direito que satisfazem à regra, mas não outras.

```
factorial(0, 1).
factorial(N, Result) :- N > 0, M is N - 1,
                        factorial(M, P),
                        Result is N * P.
```

| Figura 15.7 A Função Fatorial em Prolog

```
?- factorial(4, X).                N  M   P      Result
Call: (  7) factorial(4, _G173)    4  3   _G173  4*P
Call: (  8) factorial(3, _L131)    3  2   _L131  3*P
Call: (  9) factorial(2, _L144)    2  1   _L144  2*P
Call: ( 10) factorial(1, _L157)    1  0   _L157  1*P
Call: ( 11) factorial(0, _L170)    0      _L170
Exit: ( 11) factorial(0, 1)                     1
Exit: ( 10) factorial(1, 1)                     1*1 = 1
Exit: (  9) factorial(2, 2)                     2*1 = 2
Exit: (  8) factorial(3, 6)                     3*2 = 6
Exit: (  7) factorial(4, 24)                    4*6 = 24
```

| Figura 15.8 *Trace* do Fatorial (4)

Os Operadores `is`, `not` e Outros O operador interfixado `is` pode ser usado para forçar a instanciação de uma variável:

```
?- X is 3+7.
X = 10
yes
```

Prolog tem operadores aritméticos (+, −, *, /, ^) e operadores relacionais (<, >, =, =<, >=, e \=) com suas interpretações usuais. Observe que, para manter os símbolos => e <= livres para serem usados como setas, a Prolog usa =< para comparações menor ou igual e >= para comparações maior ou igual.

Considere o atributo `factorial` definido na Figura 15.7. Esse atributo define a função fatorial repetidamente, com a primeira linha definindo o *base case* ($0! = 1$) e a segunda e a terceira linhas definindo o caso recursivo ($n! = n \times (n - 1)!$ quando $n > 0$). Observe que precisamos introduzir a variável intermediária M para forçar a avaliação de N − 1 antes da chamada repetitiva. O operador `is` faz o papel de um operador de atribuição intermediário, unificando um valor com sua variável.

Uma chamada a essa função gera uma série de atribuições a diferentes instanciações das variáveis N, M e P (mostradas com o auxílio de uma série de *trace*) na Figura 15.8. As variáveis anônimas `_G173`, `_L131`, . . . são geradas pelo interpretador a cada reinstanciação da regra repetitiva nessa definição. Na primeira chamada, o primeiro argumento está ligado a 4, e o segundo está ligado a uma variável anônima (`_G173`). Esse mesmo padrão é repetido para cada uma das próximas quatro chamadas repetitivas, até que ocorra o *base case* e ao

segundo argumento _L170 é finalmente atribuído o valor 1. Agora a recursão se desenvolve, atribuindo valores intermediários às variáveis anônimas e fazendo as multiplicações.

O operador not é implementado em Prolog em termos de falha em atingir o objetivo. Isto é, a cláusula not(P) tem sucesso quando a resolução de P falha. Assim, not pode ser usado para definir uma função em lugar de *cut*. Considere as seguintes definições alternativas da função fatorial dada na Figura 15.7:

```
factorial(N, 1) :- N < 1, !.
factorial(N, Result) :-   M is N - 1,
                          factorial(M, P),
                          Result is N * P.
factorial(N, 1) :- N < 1.
factorial(N, Result) :-   not(N < 1), M is N - 1,
                          factorial(M, P),
                          Result is N * P.
```

A primeira definição mostra como a função *cut* (!) pode ser usada para delinear o *base case* da chamada repetitiva. A segunda definição mostra como not remove a necessidade de usar a função *cut* (embora à custa de alguma eficiência, já que ela avalia a cláusula N < 1 duas vezes para cada valor diferente de N). Entretanto, um estilo melhor de programação é usar not em lugar de *cut*, pois ela melhora a clareza do programa. Melhor ainda seria inverter a condição, neste caso N >= 1, evitando assim o uso de not.

É importante destacar que o operador not nem sempre age como uma negação lógica. Às vezes, ele simplesmente irá falhar quando uma das variáveis contidas em seu argumento não estiver instanciada.

A Função assert As aplicações Prolog às vezes encontram situações nas quais o programa deve "atualizar-se" em resposta à query mais recente. Por exemplo, em uma aplicação de base de dados como aquela mostrada na Seção 15.2.1, alguém pode querer acrescentar um novo número a uma árvore genealógica e deixar que aquele elemento desempenhe um papel na resposta do programa nas próximas queries. Isso pode ser feito usando a função assert, que essencialmente permite que o programa se altere dinamicamente, acrescentando novos fatos e novas regras aos existentes. Por exemplo, suponha que queremos acrescentar no programa da Figura 15.3 a afirmativa de que Jane é a mãe de Joe. Faríamos isso com a seguinte instrução:

```
?- assert(mother(jane,joe)).
```

Esse fato agora será acrescentado a todos os outros no programa, e afetará queries como estas:

```
?- mother(jane, X).

X = ron ;
X = joe ;
No
```

Além disso, a função assert pode ser acrescentada ao próprio corpo de uma definição de função e dinamicamente acrescentar novos fatos e novas regras à base de dados.

Esse tipo de atividade é importante para programas que simulam aprendizado, no sentido de que eles podem armazenar novos conhecimentos à medida que interagirem com o usuário.

15.3 EXEMPLOS PROLOG

Nas próximas seções, apresentamos exemplos de aplicações Prolog por meio de uma vasta gama de aplicações de inteligência artificial: diferenciação simbólica, solução de palavras cruzadas, processamento de linguagem natural, semânticas e o problema das oito rainhas.

Cada um desses exemplos destaca vários pontos fortes da programação declarativa, e especialmente o mecanismo backtracking interno da Prolog e o não-determinismo resultante. Enquanto estudam esses exemplos, os leitores deverão pensar sobre o que seria necessário para resolver esses tipos de problemas em um paradigma de programação imperativo; normalmente, o trabalho para escrever o código será muito maior do que em Prolog.

15.3.1 Diferenciação Simbólica

O uso de Prolog para manipulação simbólica e prova de teoremas é amplo. Esse exemplo ilustra alguns dos poderes de dedução natural da Prolog na área de raciocínio lógico, executando diferenciação simbólica e simplificação de fórmulas de cálculo simples. A Figura 15.9 mostra as regras familiares para diferenciação.

Por exemplo, diferenciando-se a função $2 \cdot x + 1$ usando essas regras resulta na resposta (não simplificada) $2 \cdot 1 + x \cdot 0 + 0$, que simplificada resulta em 2.

A solução Prolog imita essas regras uma a uma. Elas são intrinsicamente repetitivas, e assim é a solução Prolog, como mostra a Figura 15.10.

As regras Prolog são escritas nessa ordem particular, de maneira que o processo de pesquisa analisará a expressão repetidamente antes de chegar ao base case, no qual residem os termos e os fatores individuais. A Figura 15.11 mostra uma árvore de pesquisa para a query `d(x, 2*x+1, Ans)` (não são mostrados os ramos que conduzem à falha).

Nesta ilustração, as variáveis temporárias `_G268`, `_G269`, `_G275` e `_G278` representam variáveis anônimas geradas pela Prolog à medida que ela encontra respostas para os termos intermediários na expressão original.

$\frac{dc}{dx} = 0$	c é uma constante
$\frac{dx}{dx} = 1$	
$\frac{d}{dx}(u + v) = \frac{du}{dx} + \frac{dv}{dx}$	u e v são funções de x
$\frac{d}{dx}(u - v) = \frac{du}{dx} - \frac{dv}{dx}$	
$\frac{d}{dx}(uv) = u\frac{dv}{dx} + v\frac{du}{dx}$	
$\frac{d}{dx}\left(\frac{u}{v}\right) = \left(v\frac{du}{dx} - u\frac{dv}{dx}\right)\Big/ v^2$	

| Figura 15.9 Regras de Diferenciação Simbólica

```
d(X, U+V, DU+DV) :- d(X, U, DU), d(X, V, DV).
d(X, U-V, DU-DV) :- d(X, U, DU), d(X, V, DV).
d(X, U*V, U*DV + V*DU) :- d(X, U, DU), d(X, V, DV).
d(X, U/V, (V*DU - U*DV)/(V*V)) :- d(X, U, DU), d(X, V, DV).
d(X, C, 0) :- atomic(C), C\=X.
d(X, X, 1).
```

| Figura 15.10 Diferenciador Simbólico em Prolog

Os leitores devem observar que a tarefa de simplificar uma expressão algébrica não é coberta pelas regras de diferenciação simbólica. Por exemplo, a seguinte query

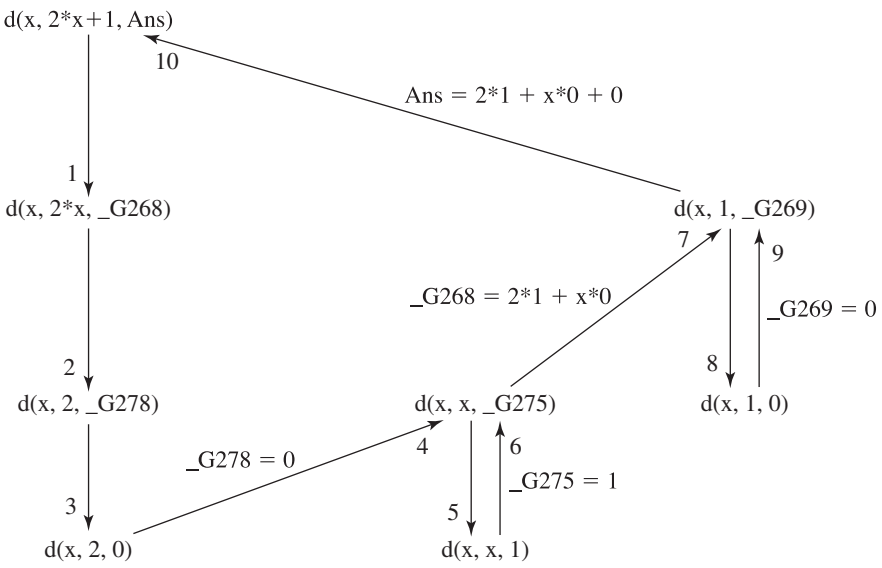
```
?- d(x, 2*x, 2).
```

não dá a resposta intuitiva `yes`, pois o resultado simbólico 2 * 1 + x * 0 + 0 obviamente não é equivalente a 2. A tarefa de simplificação depende de identidades como 1 * x = x e 0 + x = x e assim por diante. Um exercício no fim deste capítulo proporciona uma oportunidade de ampliar o programa da diferenciação simbólica dessa maneira.

15.3.2 Resolvendo Palavras Cruzadas

A lógica, às vezes, nos pede para resolver problemas que são palavras cruzadas, que são uma série de afirmações a partir das quais várias inferências podem ser feitas, e podem ser tiradas conclusões complexas. Veja aqui um exemplo simples:

> Baker, Cooper, Fletcher, Miller e Smith moram em um prédio de cinco andares. Baker não mora no quinto andar e Cooper não mora no primeiro. Fletcher não mora no



| Figura 15.11 Árvore de Pesquisa para a Query d(x, 2*x+1, Ans)

```
floors([floor(_,5),floor(_,4),floor(_,3),floor(_,2),floor(_,1)]).
building(Floors) :- floors(Floors),
        member(floor(baker, B), Floors), B \= 5,
        member(floor(cooper, C), Floors), C \= 1,
        member(floor(fletcher, F), Floors), F \= 1, F = 5,
        member(floor(miller, M), Floors), M > C,
        member(floor(smith, S), Floors), not(adjacent(S, F)),
        not(adjacent(F, C)),
        print_floors(Floors).
```

| Figura 15.12 Solução Prolog para o Problema do Prédio

último andar nem no térreo, e ele não mora em um andar adjacente a Smith ou Cooper. Miller mora em algum andar acima de Cooper. Quem mora em qual andar?

Em Prolog, podemos preparar uma lista para resolver esse tipo de problema, com uma entrada para cada andar no prédio. Como cada andar tem um ocupante e um número, a função `floor(Occupant, Number)` pode ser usada para caracterizá-lo. Podemos então enumerar os fatos que sabemos para cada andar usando essa lista e função em particular. A Figura 15.12 mostra uma solução para esse problema.

Cada linha separada na regra Prolog contém subobjetivos que representam uma das sentenças no enunciado do problema acima; todos esses subobjetivos precisam ser satisfeitos simultaneamente para que a regra toda tenha sucesso. As variáveis B, C, F, M e S representam os diferentes números dos andares das pessoas, que o programa procura instanciar. Seus valores estão restritos aos inteiros de 1 a 5 pela primeira instrução.

A função auxiliar `print_floors` é uma rotina repetitiva simples para mostrar os elementos de uma lista em linhas separadas. As cinco pessoas moram em algum lugar, assim a função `member` é usada para garantir isso.

```
print_floors([A | B]) :- write(A), nl, print_floors(B).
print_floors([]).

member(X, [X | _]).
member(X, [_ | Y]) :- member(X, Y).
```

A função `adjacent` tem sucesso sempre que seus dois argumentos X e Y diferem de 1, assim ela define o que é necessário para que dois andares sejam adjacentes um ao outro.

```
adjacent(X, Y) :- X =:= Y+1.
adjacent(X, Y) :- X =:= Y-1.
```

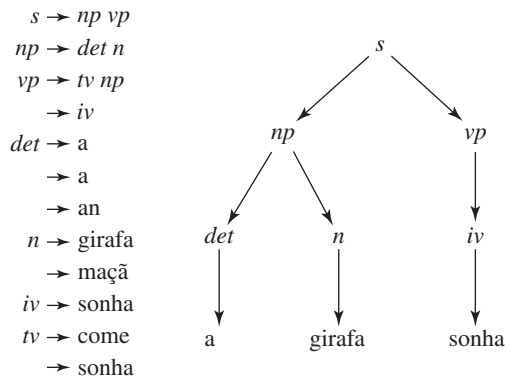


Figura 15.13
Uma gramática BNF simples e a árvore de análise para "A girafa sonha" (Giraffe Dreams)†

O quebra-cabeça é resolvido usando-se a seguinte query, que encontra as instanciações das cinco variáveis que, juntas, satisfazem a todas as restrições e atribuem uma lista de valores ao conjunto X:

```
? - building(X).
```

15.3.3 Processamento de Linguagem Natural

Podemos escrever programas Prolog que são efetivamente gramáticas BNF, que, quando executados, analisarão sentenças em uma linguagem natural. A Figura 15.13 mostra um exemplo de uma gramática dessas, juntamente a uma árvore de análise para a sentença "a girafa sonha" (the giraffe dreams).

Quando usamos a representação lista para sentenças, podemos escrever regras Prolog que dividem uma sentença em suas categorias gramaticais, usando a estrutura definida pelas próprias regras gramaticais. Por exemplo, considere a seguinte regra gramatical BNF, na qual *np* e *vp* representam as noções "sentença," "frase substantivo" e "frase verbo".

$s \rightarrow np\ vp$

Uma regra Prolog correspondente poderia ser:

```
s(X, Y) :- np(X, U), vp(U, Y).
```

As variáveis nessa regra representam listas. Em particular, X é a representação lista da sentença que está sendo analisada, e Y representa o fim resultante da lista que restará se essa regra tiver sucesso. A interpretação aqui espelha aquela da regra gramatical original: "X é uma sentença, restandoY, se o início de X pode ser identificado como um substantivo-frase, restando U, e o início de U pode ser identificado como um verbo-frase, restando Y".

† As categorias gramaticais mostradas aqui são: *s* = "sentença", *np* = "substantivo-frase", *vp* = "verbo-frase", *det* = "determinador" (ou "artigo"), *n* = "substantivo", *iv* = "verbo intransitivo" e *tv* = "verbo transitivo". Os símbolos terminais são representados por palavras não-itálicas. A análise ocorre nesses tipos de gramáticas exatamente como ocorre nas gramáticas que representam a sintaxe das linguagens de programação, conforme apresentamos no Capítulo 2.

O programa Prolog para a gramática da Figura 15.13 é mostrado a seguir:

```
s(X, Y) :- np(X, U), vp(U, Y).

np(X, Y) :- det(X, U), n(U, Y).

vp(X, Y) :- iv(X, Y).
vp(X, Y) :- tv(X, U), np(U, Y).

det([the | Y], Y).
det([a | Y], Y).

n([giraffe | Y], Y).
n([apple | Y], Y).

iv([dreams | Y], Y).
tv([dreams | Y], Y).
tv([eats | Y], Y).
```

Observe que os fatos que identificam símbolos terminais (girafa, come etc.) efetivamente retiram aqueles símbolos do início (head) da lista que está sendo passada por meio da gramática.

Para ver como isso funciona, considere a seguinte query Prolog, que pergunta se "a girafa sonha" é ou não uma sentença.

```
?- s([the, giraffe, dreams], [])
```

Aqui, `X` e `Y` são identificados com as duas listas dadas como argumentos, e a tarefa é encontrar uma lista U que satisfará, na ordem dada, a cada um dos seguintes objetivos (usando o lado direito da primeira regra no programa).

```
np([the, giraffe, dreams],U) vp(U,[])
```

Uma maneira de ver a dinâmica de todo o processo de análise gramatical é rodar um *trace* na própria query, que é mostrada na Figura 15.14. Por meio desse *trace*, podemos ver que as variáveis `U` e `Y` são instanciadas para `[dreams]` e `[]` respectivamente, para satisfazer o lado direito da primeira regra gramatical. Observe que, após sair de cada nível do *trace*, uma ou mais palavras são removidas do início da lista. Uma leitura cuidadosa desse *trace* revela uma correspondência direta entre as chamadas bem-sucedidas (aplicações da regra) e os nós da árvore de análise gramatical mostrados na Figura 15.13 para essa sentença. Assim, a leitura de um *trace* pode ajudar a desenvolver uma árvore de análise gramatical complexa para uma sentença.

O uso de Prolog para codificar gramáticas complexas dessa maneira às vezes mais atrapalha do que ajuda. Por essa razão, Prolog fornece uma notação muito compacta que imita diretamente a notação das próprias regras gramaticais livres de contexto. Essa notação é chamada *Definite Clause Grammar* (DCG) e é simples de assimilar.[4] O operador Prolog `-->`

4. DCGs são, em geral, LL(n) para um n arbitrário à frente. Assim, é necessário eliminar a repetição à esquerda nas regras gramaticais, para evitar laços infinitos durante a resolução.

```
?- s([the, giraffe, dreams],[]).
Call:  (  7) s([the, giraffe dreams], []) ?
Call:  (  8) np([the, giraffe, dreams], _L131) ?
Call:  (  9) det([the, giraffe, dreams], _L143) ?
Exit:  (  9) det([the, giraffe, dreams], [giraffe, dreams]) ?
Call:  (  9) n([giraffe, dreams], _L131) ?
Exit:  (  9) n([giraffe, dreams], [dreams]) ?
Exit:  (  8) np([the, giraffe, dreams], [dreams]) ?
Call:  (  8) vp([dreams], []) ?
Call:  (  9) iv([dreams], []) ?
Exit:  (  9) iv([dreams], []) ?
Exit:  (  8) vp([dreams], []) ?
Exit:  (  7) s([the, giraffe, dreams], []) ?
Yes
```

| Figura 15.14 Execução de *Tracing* de uma Query Prolog

é utilizado em lugar do operador :- em cada regra, e as variáveis de lista dentro da regra são, portanto, suprimidas. Por exemplo, a regra

```
s(X, Y) :- np(X, U), vp(U, V).
```

pode ser substituída pela seguinte versão simplificada equivalente:

```
s --> np, vp.
```

Ao fazer essa transformação, é importante enfatizar que não estamos mudando a aridade da função `s` (ainda 2) ou o significado da própria regra original. Essa notação foi introduzida como uma espécie de "macro" que permite que as regras Prolog sejam escritas de forma quase idêntica às regras gramaticais BNF que elas representam. Veja abaixo uma reescrita completa do programa Prolog para a gramática na Figura 15.13:

```
s --> np, vp.

np --> det, n.

vp --> iv.
vp --> tv, np.

det --> [the].
det --> [a].
n --> [giraffe].
n --> [apple].
iv --> [dreams].
tv --> [dreams].
tv --> [eats].
```

Um refinamento adicional para as regras de escrita gramatical proporciona capacidade para gerar uma árvore de análise gramatical diretamente a partir da gramática. Isto é, a gramática mostrada aqui pode ser modificada de maneira que uma query não dê apenas uma resposta `Yes` ou `No`, mas uma árvore de análise gramatical completa em forma funcional como resposta. Por exemplo, a forma funcional da árvore de análise gramatical da Figura 15.13 é:

```
s(np(det(the), n(giraffe)), vp(iv(dreams)))
```

Essa modificação é feita acrescentando-se um argumento adicional ao lado esquerdo de cada regra e variáveis apropriadas para conter os valores intermediários que são derivados nos estágios intermediários da execução. Por exemplo, a primeira regra na gramática acima seria ampliada da seguinte forma:

```
s(s(NP,VP)) --> np(NP), vp(VP).
```

Isso significa que a query precisa de um argumento extra, juntamente à sentença a ser analisada gramaticalmente e à lista vazia. Tal argumento, que aparece primeiro na query, é uma variável que conterá a árvore de análise gramatical, conforme mostramos abaixo:

```
?- s(Tree, [the, giraffe, dreams], []).
```

Deixamos como exercício uma revisão completa da gramática acima, que engloba esse refinamento.

É importante destacar que Prolog pode ser usada para gerar sentenças bem como para analisá-las gramaticalmente. Por exemplo, considere a aplicação da seguinte query geral ao programa Prolog acima (gramática):

```
? - s(Sentence, []).
```

Essa query, quando iniciada, solicita ao processo de pesquisa que encontre todas as instanciações da variável `Sentence` que terão sucesso com essa gramática. Na lista de respostas, encontraremos a instância a seguir, bem como todas as outras que podem ser geradas por essa gramática:

```
Sentence = [the, giraffe, dreams] ;
Sentence = [the, giraffe, eats, the, apple] ;
...
```

na qual o ponto-e-vírgula pode ser interpretado como um *ou*.

O processamento de linguagem natural tem sido uma área de pesquisa desde os anos 50. O exemplo apresentado aqui não começa a explorar a extrema dificuldade do assunto. Muitos pesquisadores em processamento de linguagem natural continuam a usar Prolog como um veículo para seu trabalho.

15.3.4 Semântica de Clite

Nesta seção implementamos grande parte das semânticas formais de Clite usando Prolog. Recorde-se do Capítulo 8, que, para a linguagem elementar Clite, o ambiente é estático, de forma que o estado pode ser representado simplesmente como uma coleção de pares variável-valor. Isso é expresso da seguinte maneira:

$$state = \{\langle var_1, val_1\rangle, \langle var_2, val_2\rangle, \ldots, \langle var_m, val_m\rangle\}$$

Aqui, cada var_i representa uma variável e cada val_i representa seu valor atribuído no momento.

O estado é uma espécie de *janela de observação* em um ambiente de desenvolvimento integrado (*integrated development environment* – IDE). Ele está sempre ligado a uma instrução particular do programa fonte e mostra para cada variável de programa seu valor corrente. Em nossa implementação Java, o estado foi implementado como uma tabela hash na qual o identificador de variável era a chave e o valor associado era o valor corrente da variável.

Um estado aqui é representado naturalmente como uma lista, com cada elemento da lista sendo um par que representa a ligação de uma variável ao seu valor. Assim, o estado Clite:

$\{\langle x, 1\rangle, \langle y, 5\rangle\}$

pode ser representado como a lista Prolog:

```
[[x,1], [y,5]]
```

Em seguida, implementamos as funções de acesso de estado denominadas `get` e `onion` (união de substituição) da implementação Java. Lembre-se de que a função `get` em Java foi usada para obter o valor de uma variável a partir do estado corrente. A função `get` toma uma variável de entrada e um estado de entrada e produz um valor de saída.

```
/* get(var, inState, outValue) */
```

Como o sistema de tipos Clite requer que todas as variáveis usadas em um programa sejam declaradas, não pode haver referência a uma variável que não esteja no estado. O base case é que o par variável-valor ocorre no início da lista de estado, caso em que o valor associado com a variável é o valor resultante desejado. Caso contrário, a pesquisa continua até o fim da lista.[5]

```
get(Var, [[Var, Val] | _], Val).
get(Var, [_ | Rest], Val) :- get(Var, Rest, Val).
```

Uma aplicação da função `get` é:

```
?- get(y, [[x, 5], [y, 3], [z, 1]], V).
V = 3.
```

A função `onion` toma uma variável de entrada, um valor de entrada, um estado de entrada e produz um novo estado com a parte valor do par correspondente a variável-valor substituída pelo novo valor.

```
/* onion(var, val, inState, outState) */
```

Lembre-se de que a função `onion` é capaz de gerar a hipótese simplificadora de que a variável que estamos procurando ocorre exatamente uma vez no estado. O *base case* é que a variável que está sendo correspondida ocorre no início da lista; o estado de saída é apenas um novo par variável-valor encadeado com o restante do estado de entrada. Caso contrário,

5. Um leitor astuto que já tenha lido o Capítulo 14 pode ter notado a semelhança dessa implementação com a implementação Scheme.

o novo estado é construído a partir do início encadeado com o estado de saída resultante da aplicação repetitiva de `onion` no fim do estado de entrada.

```
onion(Var, Val, [[Var, _] | Rest], [[Var, Val] | Rest]).
onion(Var, Val, [Xvar | Rest], [Xvar | OState]) :-
   onion(Var, Val, Rest, OState).
```

Uma aplicação de `onion` é:

```
?- onion(y, 4, [[x, 5], [y, 3], [z, 1]], S).
S = [[x, 5], [y, 4], [z, 1]].
```

Em seguida, considere a função de significado para a avaliação de expressão Clite somente para os inteiros. Para facilitar isso, escolhemos uma representação apropriada para uma expressão Clite em sintaxe abstrata. Uma possibilidade é usar listas; em lugar disso, preferimos usar estruturas:

```
value(val), where val is a number
variable(ident), where ident is a variable name
operator(term1, term2), where operator is one of:
    plus minus times div -- arithmetic
    lt, le, eq, ne, gt, ge -- relational
```

A implementação dessas funções de significado segue diretamente das regras dadas no Capítulo 8. Assumimos também que tenha sido executada uma verificação estática de semânticas. O significado de uma expressão Clite abstrata é implementado como uma série de regras, dependendo do tipo de expressão. Em Prolog, essas regras tomam uma expressão de entrada e um estado de entrada e retornam um valor:

```
/* mexpression(expr, state, val) */
```

O significado de uma expressão `value` é exatamente o próprio valor.

```
mexpression(value(Val), _, Val).
```

O significado de uma `variable` é o valor associado com a variável no estado corrente, obtido com a aplicação da função `get`.

```
mexpression(variable(Var), State, Val) :-
   get(Var, State, Val).
```

O significado de uma expressão binária é obtido aplicando-se o operador ao significado dos operandos; mostramos abaixo o significado para `plus`:

```
mexpression(plus(Expr1, Expr2), State, Val) :-
   mexpression(Expr1, State, Val1),
   mexpression(Expr2, State, Val2),
   Val is Val1 + Val2.
```

Essa definição diz que primeiro deve-se avaliar `Expr1` em `State` dando `Val1`, depois avaliar `Expr2` em `State` dando `Val2`. Depois, some os dois valores, e teremos o valor resultante. Os demais operadores binários são implementados de forma similar.

Finalmente, precisamos escolher uma representação para a sintaxe abstrata das instruções Clite. Embora pudéssemos usar uma representação lista (como fizemos para Scheme), preferimos usar estruturas:

```
skip
assignment(target, source)
block([s1, ... sn])
loop(test, body)
conditional(test, thenbranch, elsebranch)
```

O significado de uma *Instrução* abstrata é uma função de transformação de estado da forma que toma um *Estado* como entrada e produz um *Estado* como saída. A implementação dessas funções de significado segue diretamente das regras dadas no Capítulo 8 (e resumidas aqui). Assumimos também que tenha sido executada uma verificação estática de semânticas. A função de significado para uma instrução Clite pode ser escrita como uma seqüência de regras Prolog. Lembre-se de que a função de significado para uma instrução toma uma instrução de entrada e um estado de entrada e calcula um estado de saída:

```
/* minstruction(statement, inState, outState) */
```

Uma instrução *Skip* corresponde a uma instrução vazia. Como tal, ela deixa o estado inalterado; o estado de saída é uma cópia do estado de entrada.

```
minstruction(skip, State, State).
```

Uma instrução *Atribuição* consiste em uma *Variável* alvo e uma *Expressão* de origem. O estado de saída é computado a partir do estado de entrada, substituindo o *Valor* da *Variável* pelo valor computado da *Expressão* de origem, que é avaliada usando-se o estado de entrada. Todas as outras variáveis têm o mesmo valor no estado de saída que elas tinham no estado de entrada.

Assim, a implementação do significado de uma atribuição avalia a expressão de origem no estado atual, resultando em um valor, e então usa aquele valor para produzir um estado de saída (usando onion).

```
minstruction(assignment(Var, Expr), InState, OutState) :-
    mexpression(Expr, InState, Val),
    onion(Var, Val, InState, OutState).
```

As demais regras de significado ficam como exercício. Observe que essa solução é fundamentalmente funcional em sua natureza.[6] Em nenhum lugar nós requeremos o recurso de busca automática de Prolog.

6. Um leitor astuto que já tenha lido o Capítulo 14 pode querer comparar essa implementação com as implementações Scheme ou Haskell.

Esse desenvolvimento de uma pequena parte da semântica formal de Clite deve convencê-lo de que um modelo semântico completo para uma linguagem imperativa pode ser definido em Prolog. Portanto, via interpretação, a Prolog é capaz de computar qualquer função que possa ser programada em uma linguagem interpretativa. O inverso também é verdade, já que a maioria dos modernos computadores são fundamentalmente imperativos em sua natureza, e como os interpretadores Prolog existem nessas máquinas, qualquer função programada em Prolog pode ser computada por um programa imperativo. Assim, em teoria, as linguagens imperativas e as linguagens de programação lógica são equivalentes em seu poder de computação.

15.3.5 O Problema das Oito Rainhas

Finalmente, retornamos ao problema backtracking da Seção 13.4.2. Como o backtracking é o mecanismo de controle natural da Prolog, podemos passar ao desenvolvimento de uma solução para o problema das oito rainhas sem nos preocupar em desfazer movimentos de teste.

Em geral, o problema é colocar N rainhas mutuamente antagônicas em um tabuleiro de xadrez $N \times N$ de maneira que nenhuma rainha possa capturar qualquer outra rainha em um único movimento. Ao desenvolver a solução, usaremos as mesmas codificações das diagonais que usamos na Seção 13.4.2.

Desenvolveremos a solução de baixo para cima. Um grande problema no uso de Prolog é que não há estruturas globais de dados. Assim como na programação funcional, quaisquer informações necessárias devem ser passadas como argumentos a quaisquer atributos necessários. Como antes, trabalharemos coluna por coluna, procurando uma fileira na qual possamos colocar com segurança a próxima rainha. Se for encontrada uma fileira segura, passaremos para a próxima coluna, caso contrário, o programa Prolog volta e desfaz o último movimento.

As fileiras ocupadas são armazenadas como uma lista, com o número da fileira mais recente armazenado no início da lista. Assim, para um tabuleiro 4×4, a lista `[2, 0]` representa a configuração do tabuleiro:

	0	**1**	**2**	**3**
0	Q			
1				
2		Q		
3				

Como na solução Java, numeramos as fileiras e as colunas começando em 0. Assim, para formar um tabuleiro de xadrez $N \times N$:

$$0 \leq \textit{fileira}, \textit{coluna} < N$$

Primeiro determinamos se um *trial row move* é seguro. Isso é feito passando como argumentos a lista de fileiras ocupadas, diagonais sudoeste e diagonais sudeste. Para tirar vantagem do atributo `member`, tudo isso é passado como três listas separadas. Um *trial row move* é válido se o número da fileira não estiver na lista de fileiras ocupadas

e se suas diagonais associadas sudoeste e sudeste não forem membros das listas de diagonais associadas:

```
/* valid(TrialRow, TrialSwDiag, TrialSeDiag,
         Rowlist, SwDiagList, SeDiagList) */
valid(_, _, _, [ ]).

valid(TrialRow, TrialSwDiag, TrialSeDiag,
         RowList, SwDiagList, SeDiagList) :-
     not(member(TrialRow, RowList)),
     not(member(TrialSwDiag, SwDiagList)),
     not(member(TrialSeDiag, SeDiagList)).
```

Em seguida, dadas uma fileira e uma coluna, precisamos computar as diagonais sudoeste e sudeste. Da Seção 13.4.2, lembramos que a primeira é a soma dos números de fileira e coluna, enquanto a última é sua diferença:

```
/* compute SeDiag, SwDiag */
getDiag(Row, Col, SwDiag, SeDiag) :-
     SwDiag is Row + Col, SeDiag is Row - Col.
```

Em seguida, para determinada coluna, tentamos encontrar uma fileira segura para colocar a próxima rainha. Isso é feito por meio de uma iteração sobre a seqüência de números de fileira 0 ... $N - 1$:

```
/* for current col, find safe row */
place(N, Row, Col, RowList, SwDiagList, SeDiagList, Row) :-
  Row < N,
  getDiag(Row, Col, SeDiag, SwDiag),
  valid(Row, SeDiag, SwDiag, RowList, SwDiagList, SeDiagList).

place(N, Row, Col, RowList, SwDiagList, SeDiagList, Answer) :-
  NextRow is Row + 1,
  NextRow < N,
  place(N, NextRow, Col, RowList, SwDiagList, SeDiagList, Answer).
```

O último argumento é o número da fileira em que a rainha foi colocada com segurança, se o atributo `place` tiver sucesso. O primeiro atributo para `place` tem sucesso se a fileira corrente for segura. Caso contrário, é usado o segundo atributo para avançar para a próxima fileira.

Basicamente, a mesma lógica é aplicada à iteração sobre as colunas. Nesse caso, se o atributo `solve` tem sucesso, o último argumento é a lista de posicionamento de fileiras em ordem inversa:

```
/* iterate over columns, placing queens */
solve(N, Col, RowList, _, _, RowList) :-
    Col >= N.

solve(N, Col, RowList, SwDiagList, SeDiagList, Answer) :-
    Col < N,
    place(N, 0, Col, RowList, SwDiagList, SeDiagList, Row),
    getDiag(Row, Col, SwDiag, SeDiag),
    NextCol is Col + 1,
    solve(N, NextCol, [Row | RowList], [SwDiag | SwDiagList],
        [SeDiag | SeDiagList], Answer).
```

Finalmente, precisamos do próprio driver principal, que nos permite resolver para um tabuleiro arbitrário $N \times N$, para $N \geq 0$:

```
queens(N, Answer) :- solve(N, 0, [ ], [ ], [ ], Answer).
```

O segundo argumento é o resultado que contém a lista das posições de fileiras em ordem inversa. Abaixo está um exemplo da execução desse programa para $N = 0$, 1, 2, 3 e 4:

```
| ?- queens(0, R).
R = [].
no
| ?- queens(1, R).
R = [0].
no
| ?- queens(2, R).
no
| ?- queens(3, R).
no
| ?- queens(4, R).
R = [2,0,3,1];
R = [1,3,0,2];
no
| ?-
```

15.4 RESUMO

Linguagens de programação lógica como Prolog proporcionam uma maneira diferente de pensar sobre a solução de um problema. Após alguns sucessos iniciais, especialistas previram que as linguagens lógicas substituiriam amplamente as linguagens imperativas (Kowalski, 1988). Por exemplo, Prolog foi a base para o projeto Japonês do Computador de Quinta Geração (*Fifth Generation Computer*) (Shapiro, 1983) que começou no início dos anos 80. Porém, esse projeto foi abandonado após cerca de 10 anos (Fuchi *et al.*, 1993).

Apesar do fracasso dessas expectativas, foram usadas muitas outras linguagens declarativas para criar aplicações bem-sucedidas. Por exemplo, o sistema de avaliação na universidade de um dos autores era baseado em regras bem-sucedidas por muitos anos, para certificar os estudantes para a graduação. Outros sistemas bem-sucedidos baseados em regras incluem programas para identificar interações de drogas, diagnóstico de doenças a partir de sintomas e configuração de instalações de computadores.

Fora da área de inteligência artificial, as pesquisas continuam no desenvolvimento de linguagens declarativas. Um exemplo é a linguagem Datalog (Ullman, 1989), que foi desenvolvida para sistemas de bases de dados. Datalog também foi usada recentemente na área de otimização de compilador de código (Waley e Lam, 2004). A simplicidade e a clareza das regras para otimização de código em Datalog, comparadas com C ou Java, ajudam a explicar o contínuo interesse no desenvolvimento de linguagens baseadas em regras.

EXERCÍCIOS

15.1 Identifique por uma variável lógica cada uma das cláusulas nas seguintes instruções, e depois reescreva essas instruções na forma de cláusulas de Horn.

(a) Se o filme *Fantasma* estiver sendo exibido, e os preços dos ingressos forem razoáveis, então iremos ao cinema.

(b) Se a economia local estiver boa ou se Webber estiver na cidade, então os preços dos ingressos serão razoáveis.

(c) Webber está na cidade.

15.2 (a) Escreva as seguintes instruções como uma série de fatos e regras Prolog: mamíferos têm quatro pernas e não têm braços, ou dois braços e duas pernas. Uma vaca é um mamífero. Uma vaca não tem braços.

(b) Pode a Prolog chegar à conclusão de que uma vaca tem quatro pernas? Explique.

15.3 Considere a árvore genealógica definida na Figura 15.4. Desenhe uma árvore de pesquisa no estilo daquela mostrada na Figura 15.2 para a query `grandparent(Who, ron)`.

15.4 Reconsiderando a Figura 15.4, defina uma nova relação "primo" que represente a relação entre duas pessoas quaisquer cujos pais são irmãos e irmãs. Escreva uma query para esse programa expandido que identifique todas as pessoas que são primas de Ron.

15.5 Escreva um programa Prolog para encontrar o máximo, o mínimo e o intervalo dos valores em uma lista de números.

15.6 Escreva um programa Prolog `remdup` que remova todas as duplicatas de uma lista. Por exemplo, a query `remdup([a,b,a,a,c,b,b,a], X)` deve retornar `X = [a,b,c]`.

15.7 Amplie o programa de diferenciação simbólica de maneira que ele diferencie funções com expoentes, bem como somas e produtos. Essa ampliação deve se basear no seguinte conhecimento:

$$\frac{du^n}{dx} = nu^{n-1}\frac{du}{dx} \qquad \text{para inteiros } n > 0$$

Ao resolver esse problema, use o símbolo `^` para representar a exponenciação. Isto é, a expressão x^2 deve ser digitada como `x ^ 2`.

15.8 Use o seu programa ampliado de diferenciação simbólica `d` para diferenciar as seguintes funções:
(a) $x^2 + 2x + 1$
(b) $(5x - 2y)/(x^2 + 1)$

15.9 Considere o problema de simplificar uma expressão algébrica como, por exemplo, o resultado da diferenciação simbólica. Sabemos que identidades do tipo $x + 0 = x$ e $1 \cdot x = x$ são usadas na simplificação de expressões.
(a) Proponha uma série de regras para simplificação, com base nas propriedades de 0 e 1 quando somados ou multiplicados por outra expressão, e então escreva uma função repetitiva Prolog `simp` que simplifica uma expressão algébrica arbitrária.
(b) Mostre como `simp` pode ser usada para simplificar o resultado da diferenciação de expressões *a* e *b* na questão anterior.

15.10 Considerando as funções `append`, `prefix` e `suffix` definidas neste capítulo, desenhe uma árvore de pesquisa para cada uma das seguintes queries Prolog:
(a) `suffix([a], L), prefix(L, [a, b, c]).`
(b) `suffix([b], L), prefix(L, [a, b, c]).`

15.11 Considere o programa Prolog para calcular o fatorial de um número n, dado na Seção 15.2.2. Desenhe a árvore de pesquisa de subobjetivos que a Prolog usa para calcular o fatorial de 4.

15.12 Considere adicionar o seguinte fato e a seguinte regra ao programa da árvore genealógica discutido na Seção 15.2.1:

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

(a) Explique a resposta de Prolog à query `ancestor(bill, X)` usando uma árvore de pesquisa de subobjetivos.
(b) Descreva as circunstâncias gerais sob as quais um laço infinito pode ocorrer em um programa Prolog.
(c) Sugira uma pequena revisão do fato e da regra acima que evitará o problema que você descobriu na parte (a) dessa questão.

15.13 (a) Execute o programa de processamento de linguagem natural da Seção 15.3.3 para determinar se cada uma das sentenças a seguir é ou não uma sentença válida (instância de símbolo não-terminal `s`):

```
The giraffe eats the apple.
The apple eats the giraffe.
The giraffe eats.
```

(b) Sugira uma pequena alteração no programa que possa tornar as três sentenças acima válidas.

15.14 Revise a gramática Prolog da Figura 15.13 de maneira que ela produza uma árvore de análise gramatical completa para a query
`s(Tree, [the, giraffe, eats, the, apple], []).`

15.15 Complete as funções semânticas Clite iniciadas na Seção 15.3.4.

15.16 (Trabalho de Equipe) Projete um interpretador completo para Clite em Prolog, começando com a sintaxe concreta e incluindo regras Prolog para as funções *V* e *M* que definem o sistema de tipo e as semânticas de Clite.

15.17 Use Prolog para encontrar todas as soluções (zero ou mais) para o seguinte problema: Mason, Alex, Steve e Simon estão enfileirados em uma delegacia de polícia. Um deles é loiro, bonitão e sem cicatrizes. Dois deles, que não são loiros, estão de pé um de cada lado de Mason. Alex é o único que está de pé exatamente ao lado de um bonitão. Steve é o único que não está exatamente ao lado de um dos homens com cicatrizes. Quem é o loiro, bonitão e sem cicatrizes?

15.18 Escreva uma função Prolog "intersecção" que retorna uma lista que contenha somente uma instância de cada átomo (parte muito pequena) que é membro de suas duas listas de argumentos. Por exemplo, a query:

```
?- intersection([a,b,a], [c,b,d,a], Answer)
```

deve retornar a resposta `[b,a]` ou `[a,b]`. Dica: tente usar uma função `member` e outras funções que você tenha escrito para ajudar na solução deste problema.

15.19 Escreva um programa Prolog que resolva o seguinte problema:

> O senhor e a sra. Astor, o senhor e a sra. Blake, o senhor e a sra. Crane e o senhor e a sra. Davis estavam sentados ao redor de uma mesa redonda. A sra. Astor foi insultada pelo senhor Blake, que estava sentado à esquerda dela. O senhor Blake foi insultado pela sra. Crane, que estava sentada no lado oposto a ele em relação ao centro da mesa. A sra. Crane foi insultada pela anfitriã, sra. Davis. A anfitriã era a única pessoa que estava sentada entre os dois casais. A anfitriã foi insultada pela única pessoa que se sentava entre dois homens. Quem insultou a anfitriã? Descreva também a ordem em que as pessoas estavam sentadas, começando pela anfitriã.

15.20 Escreva um programa Prolog que resolva o problema a seguir. Como parte da solução, ele deverá imprimir **todas** as travessias, com o remador listado primeiro.

> Tom, Elliott, Bill e Vic tinham de cruzar um rio usando uma canoa que podia levar apenas duas pessoas. Em cada uma das três travessias da margem esquerda para a margem direita, a canoa tinha duas pessoas, e em cada uma das duas travessias da margem direita para a margem esquerda, a canoa tinha uma pessoa. Tom não era capaz de remar quando havia mais alguém na canoa com ele. Elliott não era capaz de remar quando qualquer um, exceto Bill, estava na canoa com ele. Cada pessoa remou pelo menos em uma travessia. Quem remou duas vezes?

15.21 Discuta as diferenças básicas entre os paradigmas da programação orientada a objetos e da programação lógica. Em quais circunstâncias cada uma delas é particularmente forte, e em quais circunstâncias cada uma delas é particularmente frágil?

15.22 Por que a coleta de lixo é importante para as linguagens de programação lógica? Que estratégias de coleta de lixo são usadas em interpretadores Prolog? Você pode determinar que estratégias são usadas no interpretador Prolog particular que você está usando?

15.23 (Trabalho de Equipe) Defina e implemente a sintaxe concreta e abstrata e o sistema de tipo de um pequeno subconjunto de Prolog usando o estilo notacional e a implementação Java desenvolvida para Clite.

15.24 Projete um programa Prolog que resolva (facilmente) quebra-cabeças Sudoku (**www.sudoku.com**).

Programação Orientada a Eventos

16

"De todas as misérias do ser humano, a mais amarga é esta, saber tanto e ter controle sobre nada."

Heródoto (484–432 a.C.)

VISÃO GERAL DO CAPÍTULO

Um modelo convencional de computação faz o programa prescrever a ordem exata de entrada. Esse modelo é suportado pelos quatro paradigmas de programação – imperativo, orientado a objetos, funcional e lógico. Nesse modelo, o programa controla a seqüência de passos que ocorrem em tempo de execução, e a ordenação da entrada tem um papel passivo no controle de como esses passos são executados. Além disso, esse modelo suporta algoritmos que terminam assim que a entrada é esgotada.

A programação orientada a eventos põe esse modelo às avessas.

> **Definição**: Programas orientados a eventos não controlam a seqüência na qual ocorrem eventos de entrada; em vez disso, eles são escritos para reagir a qualquer seqüência razoável de eventos.

Nesse modelo de controle, os dados de entrada governam a seqüência particular de operações que são executadas pelo programa, mas o programa não determina a ordem de seqüência de entrada. Além disso, um programa acionado por eventos usualmente é projetado para executar indefinidamente, ou pelo menos até que o usuário selecione o botão de saída.

O exemplo mais comum de programa acionado por eventos é a *interface gráfica de usuário* (*graphical user interface* – GUI), encontrada na maioria dos computadores pessoais e laptops usados nos dias atuais. Programas acionados por eventos também controlam aplicações baseadas na Internet. Por exemplo, um sistema de registro de estudante on-line deve ser preparado para interagir com um estudante, não importa qual seja sua próxima ação: acrescentar um curso, cancelar um curso, determinar a classe onde assistirá às aulas, e assim por diante. Um sistema de reservas de uma empresa aérea deve estar preparado para responder a várias seqüências de eventos do usuário, como mudar a data da viagem, a cidade de destino ou a poltrona preferida.

A programação orientada a eventos é usada também em aplicações embutidas em dispositivos como telefones celulares, motores de automóveis, sistemas de navegação em aviões e sistemas de segurança residencial. Em sistemas de navegação de aviões, os eventos que causam respostas programadas incluem mudança na direção, velocidade do vento ou temperatura; por sua própria natureza, esses eventos também não ocorrem em qualquer ordem determinada ou previsível.

Para proporcionar um suporte eficaz para a programação orientada a eventos, algumas linguagens acrescentaram suporte para o padrão de projeto modelo-visualização-controle – MVC (Gamma et al., 1995). Java tem classes que suportam um projeto MVC, embora outras linguagens, como Visual Basic e Tcl/Tk, também suportem programação orientada a eventos. Neste capítulo, usamos Java como veículo principal para ilustrar os princípios e as práticas de programação orientada a eventos.

16.1 CONTROLE ACIONADO POR EVENTOS

Stein (1999, p. 1) explica a programação orientada a eventos comparando-a com a visão tradicional da computação controlada por entradas:

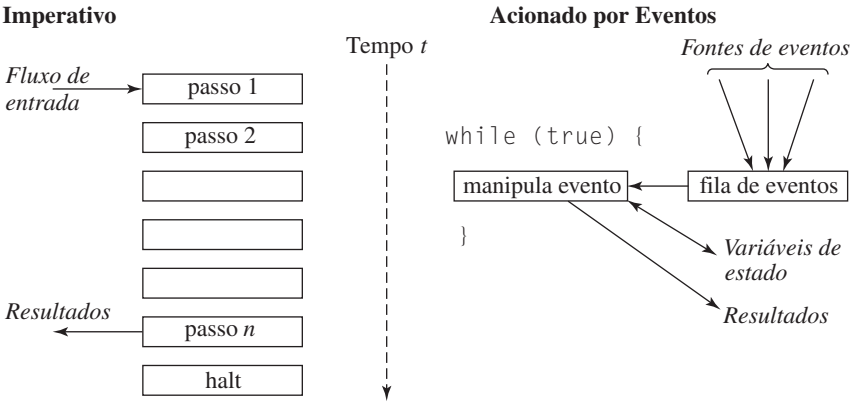
> A computação é uma função de sua entrada para sua saída. Ela é composta de uma seqüência de passos funcionais que produzem – no final – algum resultado como objetivo . [...] Esses passos são combinados por uma seqüência no tempo.

Stein argumenta que muitas computações modernas são embutidas em ambientes físicos nos quais a seqüência dos eventos no tempo é imprevisível e (potencialmente) sem um fim explícito. Para lidar com essa imprevisibilidade, Stein argumenta que a computação deverá ser modelada como uma *interação* (Stein, 1999, p. 8):

> A computação é uma comunidade de entidades persistentes acopladas entre si por seu contínuo comportamento interativo. [...] Início e término, quando presentes, são casos especiais que, muitas vezes, podem ser ignorados.

Essa visão é usada por uma ampla gama de aplicações para as quais os programas de computador são agora projetados, incluindo robótica, videogames, sistemas de posicionamento global (GPS) e sistemas de segurança e alarme domésticos.

O controle de um programa acionado por eventos é diferente do controle de um programa tradicional acionado por entradas, conforme está resumido na Figura 16.1. Aqui nós vemos que no modelo imperativo tradicional o programa é projetado de maneira que a ordem



| Figura 16.1 Comparando os Modelos Imperativo e Acionado por Eventos

de entradas determina o projeto computacional. Em uma variante desse modelo, a entrada é reunida no início da computação e os resultados são fornecidos próximo ao fim. Em outra variante, a computação toma os dados de entrada e produz resultados em um laço contínuo. Em qualquer dos casos, os programadores determinam a ordem de entrada dos dados, que, por sua vez, determina o projeto computacional.

De outro lado, a entrada para um programa acionado por eventos vem de *fontes de eventos* autônomas distintas, que podem ser sensores em um robô ou botões em uma aplicação interativa. Esses eventos ocorrem de forma assíncrona, e cada um deles entra em uma fila de eventos quando ocorre. À medida que o tempo passa, um laço de controle simples recupera o próximo evento dessa fila e o "manipula". No processo de manipular o evento, o programa pode consultar e/ou alterar o valor de uma variável ou produzir *resultados* intermediários.

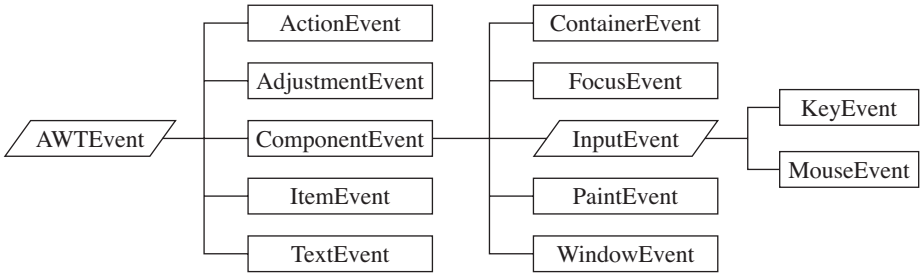
O importante é que vemos aqui duas propriedades dos programas acionados por eventos:

1. Um programa acionado por eventos tipicamente não tem um ponto de parada predeterminado, como chegar ao fim do arquivo lendo os dados.
2. O laço explícito ler-avaliar-imprimir presente nos programas tradicionais controlados por entrada não aparece explicitamente em programas acionados por evento. Dessa forma, na Seção 16.3.1, veremos uma aplicação que não contém laços.

16.1.1 Modelo-Visualização-Controle

(Gamma et al., 1995) discute uma série de padrões de projeto para programas orientados a objeto. Um padrão de aplicação, por exemplo, Modelo-Visualização-Controle (MVC), é particularmente útil no projeto de aplicações GUI acionadas por eventos.

> **Definição**: Em MVC o modelo é o objeto real que está sendo implementado, o controlador é o mecanismo de entrada (botões, menus, caixas de seleção etc.) e a visualização é a representação de saída do modelo.



| Figura 16.2 Classe Java AWTEvent e suas Subclasses[1]

Ao aplicar o padrão MVC, é uma boa prática desacoplar o modelo, a visualização e o controlador tanto quanto for possível. Em uma dada aplicação, um modelo pode ter múltiplas visualizações e múltiplos controladores.

Por exemplo, no jogo-da-velha, o jogo é o modelo. É responsabilidade do jogo determinar de quem é a vez, quando uma partida termina, quem ganhou etc. Assim, o estado da classe `Game` consiste em determinar:

- Que jogador tem o próximo movimento, `X` ou `O`.
- O estado atual do tabuleiro, isto é, o conteúdo de cada quadrado.

A não ser pelo fato de retornar uma representação visual do movimento de um jogador, a classe do jogo (e as suas classes auxiliares) não está preocupada com os aspectos visuais do jogo. Também não está preocupada com o modo como os jogadores indicam que quadrado eles escolhem para marcar. Um exemplo do jogo-da-velha está implementado na Seção 16.3.3.

A classe `Controller` é responsável por preparar a aparência visual do jogo e cuida da entrada dos jogadores. Os aspectos visuais constam de uma `TextArea` para mostrar mensagens, um botão new game para iniciar um novo jogo e o próprio tabuleiro. A função primária do controlador é a de manipular as entradas de cada jogador.

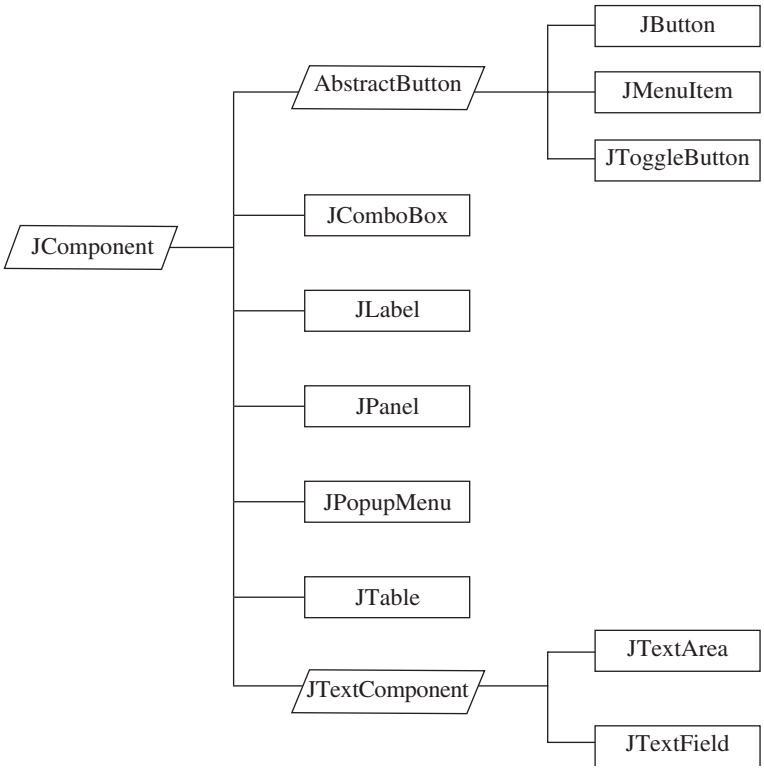
Controlar a representação visual do tabuleiro de jogo e mostrar as mensagens para os jogadores é responsabilidade do *view*. Se desejássemos ter múltiplas visualizações do jogo, a visualização seria desacoplada do modelo/jogo e do controlador. No entanto, em nossa implementação, a visualização está altamente acoplada ao controlador, tão acoplada que é muito difícil separar os dois.

Uma implementação MVC completa para o jogo-da-velha foi desenvolvida na Seção 16.3.3.

16.1.2 Eventos em Java

Java suporta programação orientada a eventos fornecendo certas classes e alguns métodos que podem ser usados para projetar uma interação. Tal projeto precisa classificar os eventos que podem ocorrer, associar as ocorrências daqueles eventos com objetos específicos na estrutura e depois manipular cada evento efetivamente quando ele ocorrer.

1. Nesses diagramas de classe, as classes abstratas são colocadas dentro de paralelogramos, enquanto as classes não-abstratas são colocadas dentro de retângulos. Essa notação é usada por Flanagan (Flanagan, 1996) em sua série de livros denominada *Java in a Nutshell*.



| Figura 16.3 Hierarquia Parcial de Classes para Fontes de Eventos

Os tipos de eventos que podem ocorrer em Java são definidos pelas subclasses da classe abstrata predefinida `AWTEvent`. Essas subclasses estão resumidas na Figura 16.2.

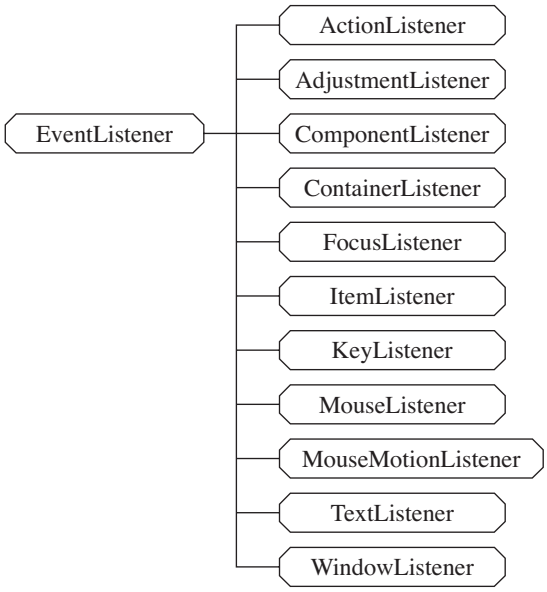
Cada fonte de evento em uma interação pode gerar um evento que é um membro de uma dessas classes. Por exemplo, se um botão é a fonte de um evento, ele gera eventos que são membros da classe `ActionEvent`. Discutiremos os detalhes dessa relação na Seção 16.2.

Os objetos que podem, eles próprios, ser fontes de eventos no *framework* Swing são subclasses da classe abstrata `JComponent`. Na Figura 16.3 está um resumo de algumas dessas classes. Aqui nós vemos, por exemplo, que qualquer botão que pode ser selecionado pelo usuário em uma interação é declarado como uma instância da classe `JButton`.

Para que um programa manipule um evento, ele deve ser equipado com "ouvidores" apropriados que reconhecerão quando um evento particular, como um clique de mouse, ocorreu em um objeto que é uma fonte de eventos. A classe `Event` tem subclasses que desempenham esse papel para cada uma das classes evento identificadas acima. Essas subclasses estão resumidas na Figura 16.4.

Por exemplo, para equipar um botão de forma que o programa possa "detectar" uma ocorrência da seleção daquele botão, o botão precisa chamar seu método `addActionListener`. Se isso não for feito, os eventos do botão não serão detectados pelo programa. Isso é discutido na Seção 16.2.

Figura 16.4
Classes Java EventListener[2]



Finalmente, para responder a eventos iniciados por objetos nessas classes, precisamos implementar métodos especiais chamados de *handlers (manipuladores).* Cada classe de eventos predefine o(s) nome(s) do(s) manipulador(es) que podem ser escritos para ela. Na Tabela 16.1 apresentamos um resumo dos manipuladores que são pré-identificados para botões, caixas de seleção, digitação de texto e eventos de mouse.

Tabela 16.1 Componentes e Seus Manipuladores de Eventos

Widget	Listener	Métodos de Interface
JButton	ActionListener	actionPerformed(ActionEvent e)
JComboBox	ActionListener	actionPerformed(ActionEvent e)
JLabel		
	MouseListener	mouseClicked(MouseEvent e)
		mouseEntered(MouseEvent e)
		mouseExited(MouseEvent e)
		mousePressed(MouseEvent e)
		mouseReleased(MouseEvent e)
	MouseMotion-	mouseDragged(MouseEvent e)
	Listener	mouseMoved(MouseEvent e)
JTextArea	ActionListener	actionPerformed(ActionEvent e)
JTextField	ActionListener	actionPerformed(ActionEvent e)

2. A inclusão de uma classe dentro de um hexágono a distingue como uma classe interface (Flanagan, 1996), em vez de uma classe regular. Lembre-se da distinção entre classes e interfaces que foi discutida no Capítulo 13.

16.1.3 Aplicações GUI Java

Java fornece recursos para programadores que criam programas acionados por eventos em que a origem dos eventos é uma interface gráfica de usuário.

> **Definição**: Uma aplicação baseada em uma *interface gráfica de usuário (graphical user interface – GUI)* é um programa que roda dentro de sua própria janela e comunica-se com os usuários utilizando elementos gráficos como botões e menus.

Pelo fato de elas serem projetadas para reagir a eventos em vez de iniciá-los, as aplicações GUI têm uma estrutura ligeiramente diferente das outras aplicações Java; essa estrutura está esquematizada na Figura 16.5.

As primeiras linhas mostradas nessa figura são diretivas `import` que possibilitam acesso às bibliotecas de classes Java tanto para os elementos gráficos quanto para os manipuladores de eventos. O `<instance variable declarations>` define vários objetos que podem ser colocados na aplicação bem como outros valores que ajudam o programa a lembrar o que ocorreu até então. Como acontece com qualquer outra classe, essas variáveis definem coletivamente o *estado* da interação. Elas podem ser variáveis de dados primitivos, objetos e conjuntos, como em qualquer outro programa de aplicação Java ou outra classe. Algumas variáveis podem representar cores, botões, menus, campos de texto e outros objetos que têm funções especiais à medida que o programa e o usuário interagem. O método `main` pode ser incluído na classe ou em uma classe separada.

O construtor tem a responsabilidade de colocar objetos no quadro, tais como botões, caixas de seleção, rótulos e campos de texto. O construtor também anexa ouvidores (listeners) de eventos específicos a diferentes objetos na estrutura, de forma que esses objetos podem ser equipados para reconhecer quando foram selecionados pelo usuário. Ouvidores de evento (event listeners) não são nada mais do que objetos que implementam uma interface particular; eles podem ser a própria aplicação GUI, conforme indicado acima, ou podem ser classes separadas. Conforme veremos, essas classes às vezes precisam de acesso a diversas variáveis da aplicação GUI, de maneira que elas comumente são implementadas como classes internas.

Uma interface gráfica de usuário muitas vezes tem um método *paint* (pintar) sempre que o aplicativo precisa redesenhar sua própria janela. Um redesenho (*repaint*) pode ocorrer por várias razões: outra janela encobre parcialmente a janela do aplicativo por

```
import java.awt.*;
import java.awt.event.*;
import java.swing.*;

public class <classname> extends JPanel
                         implements <listeners> {
   <instance variable declarations>
   public <classname> ( ) {
     <code to initialize the GUI>
   }
   <event handlers>
}
```

| Figura 16.5 Estruturas Gerais de uma Aplicação GUI em Java

algum tempo, a janela do aplicativo é minimizada e depois maximizada, e assim por diante. Componentes como botões, campos de texto etc. se redesenharão a si mesmos. No entanto, qualquer coisa feita diretamente no quadro usando métodos da classe `Graphics` será perdida durante uma operação de redesenho se não for feita no método paint.

Para cada tipo específico de evento (como a seleção de um botão, um clique do mouse ou seleção de uma caixa de seleção), o programa deve implementar um método especial chamado de `<event handler>`. A finalidade do manipulador (handler) é mudar o estado da interação de maneira que ele se "lembre" de que tal evento ocorreu. Um manipulador pode responder ao usuário que pressiona o botão do mouse, enquanto outro pode responder ao usuário que seleciona um botão no quadro. Sempre que um evento desses realmente ocorre, seu manipulador associado é executado uma vez.

O `<listeners>` que aparece no cabeçalho da classe identifica os *tipos* de eventos para os quais a aplicação está preparada para responder. Quatro tipos diferentes de eventos iniciados pelo usuário podem ser manipulados:

- Eventos de movimento do mouse (manipulados pelo `MouseMotionListener`).
- Eventos do mouse (manipulados pelo `MouseListener`).
- Seleções de botão e campo de texto (manipuladas pelo `ActionListener`).
- Seleções de uma caixa de seleção (manipuladas pelo `ActionListener`).

O importante é que o programa não pode saber, ou prever, a ordem na qual ocorrerão os diferentes eventos ou o número de vezes que cada um será repetido; ele tem de estar preparado para todas as possibilidades. Essa é a essência da programação orientada a eventos. Outra diferença importante é a de que o programador escreve cada método para executar sua tarefa e sair. O laço que espera pelos eventos e os despacha faz parte dos recursos fornecidos por Java.

16.2 MANIPULAÇÃO DE EVENTOS

Nesta seção descrevemos as considerações de programação Java básica para responder aos vários tipos de eventos iniciados pelo usuário que ocorrem enquanto uma aplicação acionada por evento está rodando – eventos do mouse, seleção de botões, entradas em campos de texto e opções em caixas de seleção.

16.2.1 Cliques do Mouse

Para que o programa manipule cliques do mouse, a interface `MouseListener` deve ser implementada pela própria aplicação ou por uma classe manipuladora de mouse. Se o listener estiver sendo manipulado diretamente pela classe, então a classe deve especificar o listener em sua cláusula `implements` e ativar o listener, usualmente dentro do construtor:

```
public class MyApplication extends JPanel implements MouseListener {

     public MyApplication( ) {
         ...
         addMouseListener(this);
         ...
     }
```

A alternativa é usar uma classe separada para manipular cliques do mouse. Geralmente, essa classe é uma classe interna à aplicação, de forma que o manipulador do mouse tem acesso a todos os métodos da aplicação, particularmente o contexto gráfico:

```
public class MyApplication extends JPanel{

    public MyApplication( ) {
        ...
        addMouseListener(new MouseHandler());
        ...
    }

    private class MouseHandler implements MouseListener {
        ...
    }
```

Se for usada uma classe externa, é comum a classe passar a si própria na chamada do construtor do manipulador do mouse; o objeto pode então ser salvo em uma variável de instância da classe manipuladora do mouse.

Qualquer que seja a alternativa usada, todos os métodos a seguir devem ser acrescentados à classe que implementa o `MouseListener`, e pelo menos um deles deve ter algumas instruções que respondam ao evento que ele representa:

```
public void mousePressed(MouseEvent e) { }
public void mouseReleased(MouseEvent e) { }
public void mouseClicked(MouseEvent e) { }
public void mouseExited(MouseEvent e) { }
public void mouseEntered(MouseEvent e) { }
```

Uma vantagem do uso de uma classe separada é que Java fornece a classe `MouseAdapter`, que é precisamente a implementação trivial da classe `MouseListener` dada acima. Isso significa que a classe separada pode estender a classe `MouseAdapter` (o que uma aplicação não pode fazer), anulando exatamente os métodos para os quais devem ser providenciadas ações. Em muitas instâncias, isso usualmente é apenas o método `mouseClicked` ou `mousePressed`.

```
public class MyApplication extends JPanel {

    public MyApplication( ) {
    ...
    addMouseListener(new MouseHandler());
    ...
  }

  private class MouseHandler extends MouseAdapter {
      public void mouseClicked(MouseEvent e) {
          <action>
      }
  }
```

Uma resposta típica para um evento do mouse é capturar as coordenadas x-y do pixel em que aquele evento ocorre na janela do programa. Para fazer isso são usados os métodos `getX` e `getY` da classe `MouseEvent`. Por exemplo, o seguinte manipulador responde a um clique do mouse armazenando as coordenadas x-y do clique nas variáveis de instância `x` e `y` da classe externa.

```
public void mouseClicked(MouseEvent e) {
    x = e.getX();
    y = e.getY();
}
```

16.2.2 Movimento do Mouse

Similar aos cliques do mouse, para manipular o movimento do mouse a interface `MouseMotionListener` deve ser implementada pela própria aplicação ou por uma classe manipuladora do movimento do mouse. Para criar o listener devem ser colocadas as seguintes chamadas dentro do construtor:

```
addMouseMotionListener(<listener>);
```

A classe que implementa o listener deve implementar todos os métodos a seguir; pelo menos um deles deve ter algumas instruções que respondam ao evento que o listener representa:

```
public void mouseDragged(MouseEvent e) { }
public void mouseMoved(MouseEvent e) { }
```

Uma vantagem do uso de uma classe separada é que Java fornece a classe `MouseMotionAdapter`, que é precisamente a implementação trivial de `MouseMotionListener` dada acima. Isso significa que a classe separada pode estender a classe `MouseMotionAdapter` (que uma classe que estende `JPanel` não pode fazer), anulando exatamente os métodos para os quais devem ser providenciadas ações. Em muitas situações, esse é usualmente o método `mouseDragged`.

16.2.3 Botões

Um *botão* é um objeto com nome na tela que pode ser selecionado com um clique do mouse. Como muitas variáveis podem ser declaradas com a classe `JButton` e colocadas na aplicação, os manipuladores de botões usualmente são implementados por meio de uma classe separada, e não pela própria aplicação. Um botão é declarado e inicializado da seguinte forma:

```
JButton <variable> = new JButton(<string>);
```

Veja aqui um exemplo:

```
JButton clearButton = new JButton("Clear");
```

Um botão é colocado na aplicação incluindo o seguinte dentro do construtor no seu programa:

```
add(<variable>);
```

Por exemplo:

```
add(clearButton);
```

Para ser útil, um botão precisa ter um listener anexado a ele, de maneira que o botão responda aos cliques do mouse. Isso normalmente é feito com a inclusão do seguinte, dentro do construtor da aplicação:

```
<variable>.addActionListener(<listener>);
```

Por exemplo:

```
clearButton.addActionListener(new ClearButtonHandler());
```

A classe que manipula o botão selecionado pelo usuário deve implementar a interface `ActionListener`. A classe listener deve implementar um método `actionPerformed` para manipular o evento de seleção do botão:

```
public void actionPerformed (ActionEvent e) {
    if (e.getSource() == <variable>) {
       <action>
    }
}
```

Aqui, `<variable>` refere-se ao nome da variável `JButton` conforme declarada e inicializada, e `<action>` define o que fazer sempre que ocorre um evento. Se for criado um único manipulador para cada botão, o teste para determinar qual botão foi selecionado pode ser omitido; isso normalmente é preferível. A seguir, por exemplo, está um manipulador escrito como uma classe interna à classe da aplicação que limpa a tela sempre que o `clearButton` é clicado pelo usuário:

```
private class ClearButtonHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        repaint();
    }
}
```

Observe que o método `repaint` é um método `JPanel`; portanto, essa classe funciona como está escrita somente se ela for uma classe interna à classe da aplicação que estende `JPanel`. O uso de uma classe externa requer a escrita de um construtor que toma a aplicação como um argumento. Usando nosso *clear button* como exemplo, o código de `addActionListener` no construtor da aplicação apareceria assim:

```
clearButton.addActionListener(new ClearButtonHandler(this));
```

A classe manipuladora de botão seria então assim:

```
public class ClearButtonHandler implements ActionListener {
     JPanel app;
     public ClearButtonHandler(JPanel a) { app = a; }
     public void actionPerformed(ActionEvent e) {
          app.repaint( );
     }
}
```

Observe que o uso de uma classe externa torna mais difícil para a classe listener determinar qual botão foi selecionado, já que os próprios botões são declarados dentro da classe da aplicação.

16.2.4 Rótulos, Áreas de Texto e Campos de Texto

Um objeto `JLabel` é um objeto cujo valor *string* pode ser colocado dentro de uma janela para rotular outro objeto, como um `JTextField`. Ele pode ser acrescentado à janela a partir do construtor. Por exemplo, a instrução

```
add(new JLabel("Fahrenheit"));
```

colocaria a mensagem "Fahrenheit" na janela. Rótulos são como botões, exceto pelo fato de os rótulos não poderem ter um listener anexado a eles.

Um objeto `JTextArea` é um objeto na tela que tem um nome e pode conter mensagens de texto de múltiplas linhas. É um objeto que pode ser rolado, de forma que pode conter mais texto do que normalmente seria visível na tela de uma só vez. Um objeto `JTextField` é um objeto que contém uma única linha de texto. Qualquer um dos dois pode ser usado para capturar texto do teclado ou simplesmente mostrar mensagens do aplicativo. No primeiro caso, um objeto `ActionEvent` é acionado quando o usuário pressiona a tecla Enter ou Return. `JTextArea` e `JTextField` são declarados da seguinte forma:

```
JTextArea <variable>;
JTextField <variable>;
```

Por exemplo:

```
JTextArea echoArea;
JTextField typing;
```

Os objetos `JTextArea` e `JTextField` normalmente são colocados na aplicação como parte do construtor no programa. A inicialização requer que sejam especificados o número de linhas de texto (para `JTextArea`) e o número de caracteres por linha.

```
<variable1> = new JTextArea(<lines>, <chars>);
add(<variable1>);
<variable2> = new JTextField(<chars>);
add(<variable2>);
```

Por exemplo:

```
echoArea = new JTextArea(5, 40);
add(echoArea);
typing = new JTextField(40);
add(typing);
```

Neste exemplo, declaramos e colocamos um `JTextArea` de 5 linhas por 40 caracteres e um `JTextField` de 40 caracteres no painel atual.

Quando o usuário digita um `JTextField` ou um `JTextArea` e pressiona a tecla Return, a aplicação pode manipular aquele evento por meio de um código adicional em seu manipulador de evento `actionPerformed`:

```
public void actionPerformed (actionEvent e) {
   if (e.getSource() == <variable>) {
     String s = <variable>.getText();
     <action>
   }
}
```

Aqui, <`variable`> refere-se ao nome da variável `JTextArea` ou `JTextArea` que foi declarada e colocada na estrutura pelo construtor, como `typing`, no exemplo acima. Quando ocorre o evento, o valor *string* digitado na área de texto é atribuído à variável `s`, e a <`action`> é então executada.

Uma solução melhor é usar uma classe interna que atende especificamente ao `JTextField typing`:

```
private class TextHandler implements ActionListener {
   public void actionPerformed (actionEvent e) {
     String s = <variable>.getText( );
     <action>
   }
}
```

Neste caso, o manipulador não precisa verificar a origem do evento causador; ele deve ser o usuário pressionando a tecla Return (ou Enter) no `JTextField typing`.

Como exemplo de uma <`action`>, a digitação do usuário pode ser ecoada imediatamente na área de texto (TextArea) encadeando-a com o texto que já está lá. Os métodos `getText` e `setText` são úteis para essa finalidade, conforme mostramos a seguir:

```
echoArea.setText(echoArea.getText() + s + "\n");
```

Se essa linha for acrescentada à <`action`> no código acima, a digitação do usuário será ecoada em uma nova linha dentro do objeto `JTextArea` denominado `echoArea`.

16.2.5 Caixas de Seleção

Uma *caixa de seleção* é um objeto na tela que oferece várias opções para serem selecionadas pelo usuário; é como um menu desdobrável, mas ela pode ser colocada em qualquer lugar dentro da aplicação, não somente no topo. Uma caixa de seleção é

declarada da seguinte forma:

```
JComboBox <variable> ;
```

Por exemplo:

```
JComboBox combo;
```

A caixa de seleção é nomeada e colocada na janela como parte do construtor do seu programa. As diferentes seleções são atribuídas à variável `JComboBox` como parte do construtor ou usando o método `addItem`. A interação é disparada adicionando-se um listener à estrutura para a caixa de seleção.

```
<variable> = new JComboBox( );
<variable>.addItem(<string1>);
...
add(<variable>);
<variable>.addActionListener(<listener>);
```

Por exemplo:

```
combo = new JComboBox( );
combo.addItem("North");
combo.addItem("East");
combo.addItem("South");
combo.additem("West");
add(combo);
combo.addItemListener(new ComboHandler());
```

No caso acima, assume-se que uma classe interna à própria aplicação esteja manipulando o evento caixa de seleção. O manipulador de evento do listener da aplicação deve implementar `ActionListener`. Quando o usuário seleciona uma das caixas de seleção, o evento é manipulado por um método `actionPerformed`:

```
private class ComboHandler implements ActionListener {
   public void actionPerformed (ActionEvent e) {
     String s = (String)combo.getItem( );
     if (s.equals(<string1>)) {
       <action1> in response to a selection of <string1>
     }
     else if (s.equals(<string2>)) {
       <action2> in response to a selection of <string2>
     }
     ...
   }
}
```

Quando ocorre o evento de seleção de uma caixa de seleção, esse manipulador é executado. A String `s` apanha o valor da escolha que o usuário selecionou, que é acessível ao

manipulador por meio da chamada de método `combo.getItem( )`. Essa escolha é usada em uma série de testes para selecionar a <`action`> apropriada.

16.3 TRÊS EXEMPLOS

Nesta seção, ilustramos três problemas que utilizam programação orientada a eventos. O primeiro é o projeto de uma interface GUI simples. O segundo é um Java applet, e o terceiro envolve o projeto de um jogo interativo usando o padrão de projeto modelo–visualização-controle (MVC).

16.3.1 Uma Interface GUI Simples

Considere o projeto de uma interface simples em que o usuário pode inserir retângulos e colocar textos, em localizações arbitrárias, no quadro. O usuário deve ser capaz de fazer isso da maneira mais simples possível, e, portanto, é essencial que a tela contenha botões, menus e áreas de digitação de texto, bem como manipular as ações do mouse. Na Figura 16.6, mostramos o esboço de um projeto para suportar essa atividade.

Essa janela tem quatro objetos: um botão clear, um menu de caixa de seleção, uma área de texto para se comunicar com o usuário, à medida que os eventos são iniciados, e um campo de texto no qual o usuário pode colocar mensagens. Assim, podemos começar nosso projeto definindo o estado da computação com os seguintes objetos e as seguintes interpretações:

`combo`	O usuário pode selecionar Nothing, Rectangle ou Message nesse menu.
`echoArea`	Uma área de texto para relatar o evento mais recente que ocorreu.
`typing`	Uma área de texto para entrada do usuário.

O botão não precisa ser parte do estado global, já que sua função é meramente limpar a tela, e isso pode ser feito completamente dentro do manipulador de evento para o botão. Nenhum outro objeto ou manipulador precisa estar ciente sobre o botão clear.

O estado da computação para essa aplicação também precisa manter controle de quaisquer informações que sejam relevantes para executar a próxima tarefa, seja ela desenhar um retângulo ou colocar um texto em algum lugar na janela. Para desenhar um retângulo, o sistema precisa de duas informações: as coordenadas x-y do canto superior esquerdo do retângulo e as coordenadas x-y do canto inferior direito. Esses dois pontos podem ser acessados por

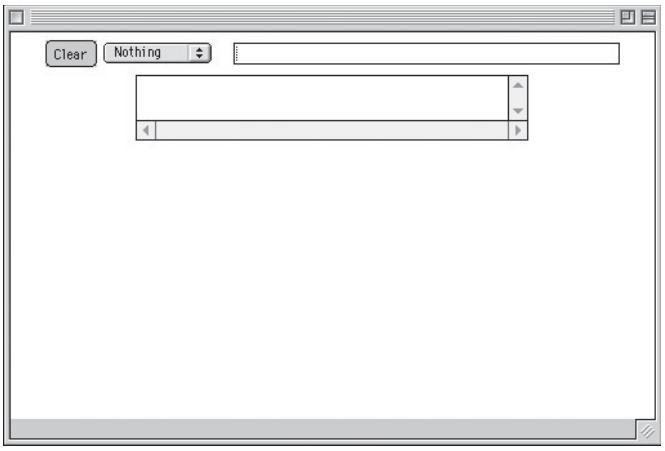


Figura 16.6
Projeto Inicial para uma Ferramenta de Desenho Gráfico

meio de um evento de mouse e do manipulador `mouseClicked`, mas o primeiro par de coordenadas x-y deve ser armazenado globalmente dentro do estado da aplicação. Além disso, também deve ser armazenada globalmente uma contagem dos cliques do mouse, de maneira que o primeiro clique possa ser distinguido do segundo na determinação dos vértices do retângulo. Assim, as informações adicionais de estado globais incluem:

```
lastX, lastY      X-Y coordinates of last mouse click
clickNumber       The number of mouse clicks (odd or even)
```

As coordenadas x e y de um clique de mouse são informadas no `JTextArea` pelo programa sempre que ocorrer um evento de clique do mouse. As mensagens de informação ao usuário também são mostradas aqui.

O código que define o estado é mostrado na Figura 16.7, e o código que inicializa a interação é mostrado na Figura 16.8.

O projeto de manipuladores de evento é um processo mais complicado. Para cada evento que ocorre, um manipulador apropriado deve conter código para distinguir aquele evento dos restantes e então mudar o estado e/ou gerar saída na estrutura de forma apropriada. Considere o seguinte cenário:

> Primeiro, o usuário seleciona "Rectangle" no menu da caixa de seleção. Para manipular corretamente o evento, o sistema deve armazenar aquela seleção e então solicitar ao usuário que dê um clique com o mouse no ponto onde deverá ser traçado o canto superior esquerdo do retângulo, como mostra a Figura 16.9.

Que código escrevemos para responder a essa seleção? Escrevemos código dentro de um método `actionPerformed` da classe manipuladora de evento `ComboHandler`, pois o objeto selecionado é uma `JComboBox`. O manipulador completo para esse problema é mostrado na Figura 16.10.

Embora esse código seja um pouco obscuro, ele revela que o manipulador `ActionListener` deve estar preparado para *qualquer* evento de menu que possa ocorrer. Ele faz a distinção entre as possibilidades, verificando o parâmetro `e` para o item selecionado, que é atribuído à variável de estado `combo`. Se a escolha for um `Rectangle`, o manipulador solicita respeitosamente ao usuário que dê um clique para definir a localização do canto superior esquerdo do retângulo no quadro. Se a escolha for uma `Message`, o usuário receberá uma solicitação diferente. Em qualquer dos casos, esse manipulador ecoa a natureza desse evento na `echoArea`.

Supondo-se que o usuário clique o mouse no quadro, o sistema deve responder armazenando as coordenadas x-y daquele clique nas variáveis de estado `lastX`, `lastY` da aplicação e solicitar ao usuário que clique o mouse para definir o canto inferior direito do retângulo desejado. Mas isso deve ser feito *somente* se a escolha atual for

```
private int lastX = 0; // Click inicial das coordenadas x-y
private int lastY = 0;
private int clickNumber = 0;
private JComboBox combo;
private String[ ] choices = {"Nothing", "Rectangle", "Message"};
private JTextArea echoArea;
private JTextField typing;
```

| Figura 16.7 Definindo o Estado para Interação

```
public Skeleton( ) {
   // Definir a cor do fundo da tela e manipular comandos do mouse
   setBackground(Color.white);
   addMouseListener(new MouseHandler());

   // Criar um botão e adicioná-lo ao Painel.
   JButton clearButton = new JButton("Clear");
   clearButton.setForeground(Color.black);
   clearButton.setBackground(Color.lightGray);
   add(clearButton);
   clearButton.addActionListener(new ClearButtonHandler());

   //Criar um menu utilizando uma caixa de seleção e adicioná-lo ao Painel
   combo = new JComboBox(choices);
   add(combo);
   combo.addActionListener(new ComboHandler());

   // Criar um campo de texto e uma área de texto e adicioná-los ao Painel
   typing = new JTextField(20);
   add(typing);
   typing.addActionListener(new TextHandler());
   echoArea = new JTextArea(2, 40);
   echoArea.setEditable(false);
   add(echoArea);
  }
```

Figura 16.8 Código para Inicializar a Interação

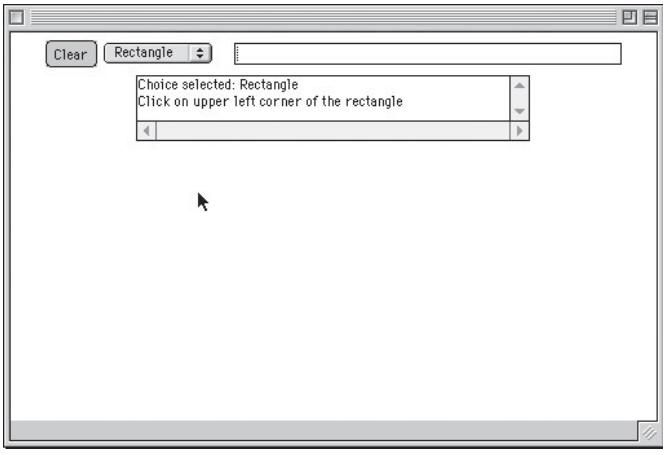


Figura 16.9 Primeiro Passo da Interação: O Usuário Seleciona um Retângulo

```
private class ComboHandler implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        String c = (String) (combo.getSelectedItem());
        echoArea.setText("Combo selected: " + c);
        // Preparar para manipular o primeiro clique do mouse
        clickNumber = 0;
        if (c.equals("Rectangle"))
            echoArea.append("\nClick to set upper"
                + "left corner of the rectangle");
        else if (c.equals("Message"))
            echoArea.append(
                "\nEnter a message in the text area");
    }
}
```

| Figura 16.10 Manipulador `ActionListener` para esta Interação

um `Rectangle`. No instante em que ocorre o evento do clique do mouse, a única fonte de informação sobre que evento o precede imediatamente vem do estado da aplicação. Isto é, o clique poderia ter sido precedido de um evento diferente, que provocaria uma resposta diferente da interação. Onde tudo isso é selecionado? No manipulador `mouseClicked`, como mostra a Figura 16.11.

Esse manipulador também deve estar preparado para qualquer coisa, já que ele não sabe implicitamente que eventos ocorreram imediatamente antes desse clique do mouse em particular. A variável de estado `clickNumber` ajuda a organizar as coisas, pois seu valor atualizado terá um número ímpar para o primeiro clique e um número par para o segundo. Assim, o canto superior esquerdo de um retângulo é indicado para valores ímpares, e o desenho de um retângulo completo, usando as coordenadas x e y do clique atual juntamente com as coordenadas x e y do clique anterior (armazenadas em `lastX` e `lastY`), é indicado para cliques pares.

O restante desse manipulador de evento deverá ser razoavelmente claro. O efeito de desenhar um retângulo após o usuário ter clicado duas vezes é mostrado na Figura 16.12. Aqui, a seta na figura mostra a localização do segundo clique, cujas coordenadas x e y são respectivamente 215 e 204.

A próxima tarefa para projetar essa interação é implementar o manipulador de eventos que responde ao usuário que seleciona o botão Clear ou digita texto na área de texto. O método `actionPerformed` para esse evento é mostrado na Figura 16.13. Observe aqui a simplicidade de se colocar um único manipulador no botão Clear; ele não tem de testar o que dispara o evento (no caso, o botão ou a tecla Enter na área de texto); ele limpa a tela via uma operação de *repaint*.

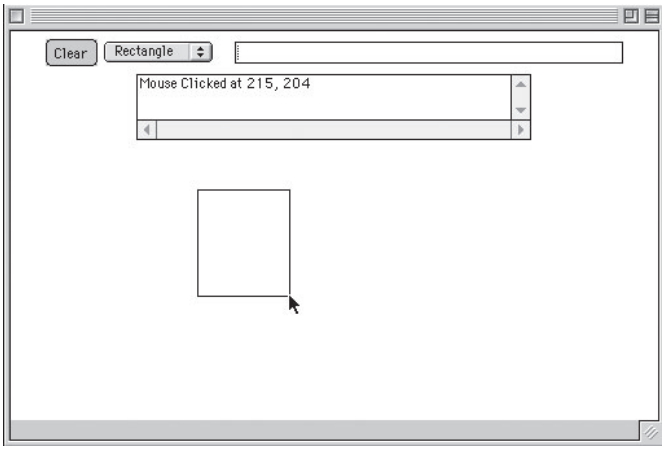
Responder a um texto digitado pelo usuário é coisa simples; precisamos apenas armazenar o texto para uso posterior e solicitar ao usuário que clique o mouse para localizar o texto no quadro, como mostra a Figura 16.14.

Observe que as classes `TextHandler` e `ClearButtonHandler` implementam a interface `ActionListener` e ambas têm métodos `actionPerformed`. Nenhum manipulador precisa ter ciência do outro, já que cada um deles está atendendo a eventos separados.

```
private class MouseHandler extends MouseAdapter {
  public void mouseClicked(MouseEvent e) {
    int x = e.getX();
    int y = e.getY();
    echoArea.setText("Mouse Clicked at " +
                e.getX() + ", " + e.getY() + "\n");
    Graphics g = getGraphics();
    if (combo.getSelectedItem().equals("Rectangle")) {
        clickNumber = clickNumber + 1;
        // Este é o primeiro clique?
        if (clickNumber % 2 == 1) {
            echoArea.append("Click to set lower right"
                + "corner of the rectangle");
            lastX = x;
            lastY = y;
            }
            // ou o segundo?
            else g.drawRect(lastX, lastY,
                Math.abs(x-lastX), Math.abs(y-lastY));
    }
    else if (combo.getSelectedItem().equals("Message"))
        // mostrar mensagem
        g.drawString(typing.getText(), x, y);
  } // mouseClicked
}
```

| Figura 16.11 Detalhes para o Manipulador `mouseClicked`

| Figura 16.12 Efeito quando se Seleciona a Opção Rectangle e Clica-se o Mouse Duas Vezes



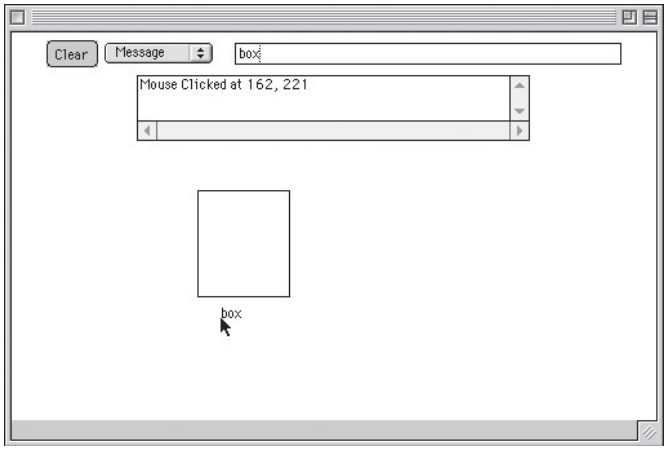
```
private class ClearButtonHandler implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        echoArea.setText("Clear button selected ");
        repaint();
    }
}
```

Figura 16.13 Manipulador ActionPerformed para o Botão Clear

```
private class TextHandler implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        echoArea.setText("Text entered: " + typing.getText());
        if (combo.getSelectedItem().equals("Message"))
            echoArea.append("\nNow click to place this message");
    }
}
```

Figura 16.14 Manipulador ActionPerformed para a Tecla Enter na Área de Digitação

Figura 16.15 Efeito Resultante Quando o Usuário Coloca Texto no Quadro



O efeito resultante quando se digita texto e clica o mouse para localizar o texto abaixo do retângulo que foi colocado no quadro é mostrado na Figura 16.15. A localização do clique do mouse é indicada pela seta na figura, que está nas coordenadas x-y 162 e 221.

Finalmente, mostramos o código para o método `main` na Figura 16.16. Esse código estabelece a janela externa como uma `JFrame` e adiciona uma instância da classe Skeleton (panel) ao seu painel de conteúdo. Depois ele estabelece um tamanho para o quadro e faz o quadro aparecer.

Esse esboço fornece um exemplo concreto do processo usado para criar um programa acionado por eventos. O programa executa indefinidamente – o usuário pode passar horas desenhando retângulos, colocando mensagens e apagando o quadro, e somente o usuário decide que seqüência de eventos ocorrerá em determinado instante no tempo.

```
public static void main(String args[]) {
   JFrame frame = new JFrame();
   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

   Skeleton panel = new Skeleton( );

   frame.getContentPane().add(panel);
   frame.setLocation(100, 100);
   // Necessário para alguns gerenciadores de janelas
   frame.setSize(500, 500);
   //frame.pack( );
   frame.show();
}
```

| Figura 16.16 O Método `main`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class <classname> extends JApplet {
                          implements <listeners> {
   <state variable declarations>
   public void init ( ) {
      <code to initialize the applet>
   }
  <event handlers>
}
```

| Figura 16.17 Estrutura Geral de um Applet Java

Nenhum dos métodos que escrevemos tem um laço. Assim, todos os métodos parecem encerrar suas respectivas execuções. No entanto, o programa continua a rodar, respondendo aos eventos. Como se consegue isso? O controle está no laço de evento discutido na Seção 16.1 e fornecido pelo Java Swing framework.

O leitor atencioso deve ter notado que essa aplicação não tem um método *paint*. Assim, minimizar a janela ou sobrepor outra janela sobre essa necessitará eventualmente de um redesenho, fazendo desaparecer todos os retângulos e as mensagens. Se a aplicação precisa redesenhar o que foi desenhado na tela, então a aplicação tem de se lembrar do que foi desenhado. Veremos exemplos disso nos próximos programas.

16.3.2 Criando um Applet Java

Nesta seção, converteremos o programa da seção anterior em um *applet*. Um *applet* é um programa Java criado para rodar dentro de um *web browser*, ou navegador para a Internet. Muitos professores já desenvolveram interessantes applets que ajudam a ilustrar conceitos difíceis em vários campos.

Nesta seção descrevemos como converter a simples aplicação da seção anterior em um applet. O código está disponível no site do livro. Ele não é mostrado aqui, porque os detalhes do código são os mesmos que foram mostrados na seção anterior. Somente a estrutura é ligeiramente diferente, como mostra a Figura 16.17.

As principais diferenças entre um applet e uma aplicação GUI são:

1. A classe principal estende `JApplet` em vez de `JPanel` ou `JFrame`.
2. Um applet não tem um método `static main`.
3. Um applet não tem um construtor. Em vez disso, o applet framework proporciona uma funcionalidade equivalente chamando o método `init` do applet, que é usado para inicializar o estado do applet, incluindo elementos GUI na tela.
4. Quando a classe estende `JPanel`, podemos acrescentar componentes diretamente no painel. Observe que no método `main` o próprio painel tem de ser acrescentado ao painel de conteúdo do quadro via chamada de método `getContentPane( ).add(component)`. Em um applet, o método `init` deve criar um `JPanel` e adicionar todas as coisas ao painel. O painel, por sua vez, é adicionado ao `contentPane` do applet.
5. O método `paintComponent` de um painel deve ser convertido para o método `paint` de um applet. A classe `Skeleton` não anula o `paintComponent` do painel.

Assim, para converter a aplicação GUI (no caso a classe `Skeleton`) em um applet `SkeletonApplet`, executamos o processo acima. O resultado, com a maior parte do código omitida, está na Figura 16.18. Os outros detalhes do código são idênticos nos dois programas.

Outros métodos applet importantes são:

- `start`: este método é invocado quando o site é visitado. Geralmente ele é usado para iniciar quaisquer *threads* (seqüência de execução) de animação.
- `stop`: este método é chamado quando se sai do site do applet ou quando o *web browser* é *iconified*. Ele é usado para interromper quaisquer *threads* (seqüência de execução) de animação.
- `destroy`: este método é o oposto do `init`. Ele é usado para liberar recursos usados pelo applet.

Portanto, utilizando o Swing framework de Java, converter uma aplicação GUI em um applet é um processo simples. A conversão oposta também é simples.

16.3.3 Jogos Interativos Acionados por Eventos

Considere a criação de um programa acionado por eventos que monitora um jogo interativo entre duas pessoas. Desenvolvemos aqui uma implementação do jogo-da-velha e discutimos a implementação do jogo de Nim, ambos na forma de programas acionados por evento.

Jogo-da-Velha No jogo-da-velha há um tabuleiro 3 × 3 e dois jogadores jogam alternadamente. Cada jogada força o tabuleiro para um novo estado que difere do anterior pelo conteúdo de um quadrado. O jogo termina quando o tabuleiro está cheio ou um dos jogadores colocou três marcas (X ou O) em uma linha na horizontal ou na vertical ou na diagonal. A interação deve rodar indefinidamente, de maneira que os jogadores podem zerar o tabuleiro e iniciar um novo jogo a qualquer instante. Portanto, o *estado* de um jogo-da-velha inclui naturalmente:

- O estado do tabuleiro.
- De quem é a vez.
- Se alguém já ganhou o jogo.

Nesse cenário, um *evento* pode ser um movimento de um jogador (clicando o mouse em um quadrado não ocupado para colocar o próximo X ou O) ou um jogador selecionando o botão "new game". A ordem de ocorrência desses eventos é totalmente imprevisível pelo programa – qualquer um desses eventos pode ocorrer enquanto o jogo estiver rodando.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class SkelApplet extends JApplet {
    // Estado global da interação
    private int lastX = 0;    // primeiro click das coordenadas x-y
    private int lastY = 0;
    ...

    public void init( ) {
        JPanel panel = new JPanel( );
        ...
        // Criar um botão e adicioná-lo ao Painel.
        JButton clearButton = new JButton("Clear");
        clearButton.setForeground(Color.black);
        clearButton.setBackground(Color.lightGray);
        panel.add(clearButton);
        clearButton.addActionListener(new ClearButtonHandler());
        ...
        getContentPane( ).add(panel);
    }

    private class ClearButtonHandler implements ActionListener {
        public void actionPerformed (ActionEvent e) {
         ...
        }
    }
    ...
}
```

| Figura 16.18 O Programa Skeleton como um Applet

Os resultados mostrados pelo programa são o próprio tabuleiro e uma área de mensagens que é usada para mostrar de quem é a vez, quem é o ganhador e outras informações à medida que o jogo continua.

O projeto geral dessa aplicação GUI segue o padrão Modelo-Visualização-Controle introduzido na Seção 16.1.1.

Modelo A classe `Game` tem um estado muito simples. Ela deve controlar de qual jogador é a próxima vez e o estado do tabuleiro. Os jogadores são representados por inteiros, e assim o tabuleiro é um *array* unidimensional de inteiros, no qual –1 representa uma célula vazia, 0 representa um `X` e 1 representa um `O`.

```
private int player = -1;    // Jogador atual: EX, ou OH
private int[ ] board = new int[9];
```

Embora visualmente o tabuleiro seja bidimensional, é mais simples usar uma representação unidimensional. Assim, os quadrados são numerados de zero até 8, percorrendo-se cada linha da esquerda para a direita.

A função principal do jogo é registrar os movimentos dos jogadores, o que é feito pelo método `move(square)`. Esse método primeiro muda para o próximo jogador (usando aritmética de módulo) e então registra o movimento. Finalmente, ele retorna uma representação visual do movimento (que é discutida a seguir).

```
public CellIcon move(int square) {
    player = (player + 1) % 2;
    board[square] = player;
    return cell[player];
}
```

Outra função do jogo é descobrir quando um jogador ganhou o jogo. Isso é implementado usando-se uma abordagem de força bruta. O *array* `lines` tem uma linha com índices para cada linha, cada coluna e cada diagonal de ganhador no jogo. O método `won( )` verifica cada um desses índices para saber se o jogador atual tem uma combinação ganhadora.

```
private int[ ][ ] lines = {
    {0, 1, 2}, {3, 4, 5}, {6, 7, 8},    //linhas
    {0, 3, 6}, {1, 4, 7}, {2, 5, 8},    //colunas
    {0, 4, 8}, {2, 4, 6}                //diagonais
};
public boolean won( ) {
    for (int i = 0; i < lines.length; i++)
        if (board[lines[i][0]] == player
            && board[lines[i][1]] == player
            && board[lines[i][2]] == player)
            return true;
    return false;
} // venceu
```

Outra função importante do jogo é limpar o tabuleiro, quando solicitado, e zerar o ganhador. Lembre-se de que um quadrado vazio é representado pelo inteiro –1.

```
public CellIcon[ ] clear( ) {
    for (int i = 0; i < board.length; i++)
        board[i] = -1;
    player = -1;
    return initial;
};
```

O método final da classe `Game` é `player( )`, que retorna uma `String` representando o jogador atual, isto é, `X` ou `O`.

As outras classes que fazem parte do modelo são a classe `CellIcon` e suas subclasses. Uma instância da classe `CellIcon` é retornada pela classe `Game` de forma que a posição do tabuleiro escolhida pelo jogador pode ser identificada corretamente com um `X` ou um `O`.

Os botões e os títulos podem ser identificados com *strings* de texto ou com ícones. Nesse último caso podemos usar imagens ou qualquer classe que implemente a interface `Icon`. Uma classe dessas deve implementar o método `paintIcon` para renderizar o ícone. Usamos ícones para que os botões tenham Xs e Os grandes.

A classe `CellIcon` é uma classe abstrata que implementa a funcionalidade básica e inclui a definição de cores e tamanhos. O único método não implementado é o método `paintIcon`, que é único para cada ícone que está sendo renderizado. O código para a classe `CellIcon` é dado na Figura 16.19.

São fornecidas três subclasses: a célula vazia, um `X` e um `O`. O código para a classe que fornece um ícone `X` é:

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class CellXIcon extends CellIcon {
    public CellXIcon(int size) { super(size); }

    public void paintIcon(Component c, Graphics g,
                int x, int y) {
       int size = size( );
       g.setColor(Color.white);
       g.fillRect(x, y, size, size);
       g.setColor(getColor( ));
       g.drawLine(x, y, x+size, y+size);
       g.drawLine(x+size, y, x, y+size);
       g.setColor(Color.black);
    }
}
```

Observe que a classe `CellXIcon` não sabe (ou não se importa) que ela está associada com um `X`; ela simplesmente se desenha quando solicitada. As duas classes restantes, bem como a classe `Game` completa, estão no site do livro.

```
import java.awt.*;
import javax.swing.*;

public abstract class CellIcon implements Icon {

    private Color color = Color.black;
    private int size;

    public CellIcon (int size) { this.size = size; }

    public int size( ) { return size; }
    public Color getColor( ) { return color; }
    public void setColor(Color c) { color = c; }

    public int getIconWidth( ) { return size; }
    public int getIconHeight( ) { return size; }
}
```

| Figura 16.19 A Classe CellIcon

As Classes de Controle Reconhecendo que muitos jogos de tabuleiro requerem um *array* retangular, ou uma "grade" de quadrados ou "células" em que são registrados os movimentos, a classe `Keypad` foi criada para suportar essas atividades. Vamos primeiro considerar o conceito de uma célula.

Uma *célula* é basicamente um quadrado capaz de se mostrar na tela. Como tal, ela tem coordenadas x-y e um tamanho; ela pode ser implementada usando a classe `Rectangle` de `java.awt`. Uma célula tem um valor; para acomodar uma variedade de jogos, os valores constantes de células incluem OFF (sem valor ou vazia), ON, EX (para o jogo-da-velha) e OH (um simples círculo). Uma célula também pode ter uma cor.

Um projetista de jogo pode ser tentado a implementar uma classe célula. No entanto, qualquer um que esteja familiarizado com a biblioteca Swing de Java perceberia que uma `JButton` tem todas as propriedades desejadas. Além disso, uma vez selecionada, ela pode ser desabilitada de maneira que não responda mais aos cliques do mouse representando movimentos ilegais. Isso simplifica a lógica de programação, já que a presença de um movimento ilegal não precisa ser verificada.

Um aspecto importante do controlador é o código de inicialização que cria um *array* retangular de botões identificados. Como parte do estado construtor ele deixa para trás botões que respondem a cliques do mouse. Esse código de inicialização para o método `keypad` é dado na Figura 16.20. Observe que, como na classe game, é usado um *array* unidimensional para representar o *array* bidimensional visual.

Qualquer célula na grade pode ser definida por um simples clique do mouse usando o ícone retornado pelo método `move` do jogo. Por exemplo, suponha que queremos colocar um X na linha 2, coluna 1 do tabuleiro do jogo-da-velha. Isso está ilustrado na Figura 16.21. Observe que o jogo "pensa" que o quadrado 7 foi selecionado.

O mouse listener atribuído aos botões da grade ou do tabuleiro é usado para identificar o quadrado selecionado. O código central acionado por eventos aparece nesse manipulador para cliques do mouse. À medida que cada botão é clicado, o método `actionPerformed` chama o método de movimento do jogo e marca o botão com o resultado. O botão é então desabilitado. Depois é feita uma verificação para saber se alguém ganhou o jogo. Em caso afirmativo, é anunciado o vencedor, e todos os botões são desabilitados. A classe listener para o mouse, no que diz respeito aos botões do tabuleiro, é dada na Figura 16.22.

Agora, com o tabuleiro e suas classes listener para o mouse em mãos, podemos completar o projeto das partes visual e manipuladora de eventos do programa jogo-da-velha. O principal elemento de projeto visual é um tabuleiro 3×3, como mostra a Figura 16.21.

```
public JPanel keypad (int rows, int cols, Icon[ ] label) {
    JPanel panel = new JPanel( );
    panel.setLayout( new GridLayout(rows, cols, 2, 2) );
    square = new JButton[label.length];
    for (int i = 0; i < label.length; i++) {
        square[i] = new JButton(label[i]);
        square[i].addActionListener( new ActionHandler(i) );
        panel.add( square[i] );
    }
    return panel;
}
```

| Figura 16.20 A Classe Keypad

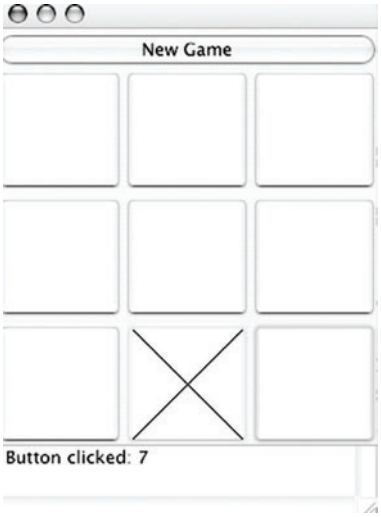


Figura 16.21
Colocação de um X na Grade

Cada jogador faz a sua vez clicando o mouse para colocar um X ou um O em um dos quadrados não ocupados; o jogador X joga primeiro. O ganhador é o jogador que primeiro colocar três Xs ou três Os em uma linha na horizontal ou na vertical ou na diagonal. Ocorre um empate no jogo quando o tabuleiro estiver cheio e ninguém tiver três Xs ou três Os em uma linha.

O programa usa uma `TextArea` para mostrar as mensagens apropriadas (por exemplo, quando um jogador ganhou). Ele permite que cada jogador em sua vez clique em um quadrado vazio e substitua aquele quadrado por um X ou um O. O programa

```
private class ActionHandler implements ActionListener {
     private int button;

     public ActionHandler(int button) {
          this.button = button;

     }

     public void actionPerformed(ActionEvent e) {
          Icon icon = game.move(button);
          square[button].setIcon(icon);
          square[button].setEnabled(false);
          if (game.won( )) {
               setButtons(false);
               echoArea.setText(game.player( ) + "wins!");
          } else
               echoArea.setText("Button clicked: " + button);
     }
}
```

Figura 16.22 A Classe Mouse Listener para os Botões do Tabuleiro

então controla de quem é a vez e coloca essas informações na `TextArea` no início de cada jogada. Então, as variáveis de estado para esse jogo incluem:

- Uma variável para os 3 × 3 botões.
- Uma variável TextArea para enviar mensagens aos jogadores.
- Uma variável que representa o jogo.

Essas variáveis são:

```
private TextArea echoArea;
private Game game;
private JButton[ ] square;
```

O construtor inicializa a tela adicionando primeiro um botão "New Game" e anexando um mouse listener a ele. Em seguida ele acrescenta o tabuleiro. Finalmente, ele acrescenta a área de texto para mensagens. Esse código de inicialização é dado na Figura 16.23.

Sempre que um jogador decide sair do jogo atual antes que ele termine, o programa deve responder apropriadamente limpando o tabuleiro e reabilitando todos os botões do tabuleiro. O manipulador de eventos deve responder quando um jogador seleciona o botão "novo jogo" sempre que isso ocorrer. Ele deve reinicializar o tabuleiro. Esse manipulador de eventos é dado na Figura 16.24.

A aplicação completa está disponível no site do livro. Ela contém o que foi apresentado aqui, as classes `Controller` e `Game` completas, bem como o programa principal.

Deve-se notar que a própria classe `Controller` e suas classes manipuladoras de eventos não possuem conhecimento sobre os detalhes do jogo. Elas não sabem sobre os

```
public Controller(Game game) {
     this.game = game;
     setLayout(new BorderLayout( ));

     // Adiciona botão para novo jogo
     Button newGame = new Button("New Game");
     newGame.addActionListener(new NewGameHandler());
     add(newGame, BorderLayout.NORTH);

     // Insere um tabuleiro 3x3 ao painel
     Icon[ ] label = game.clear( );
     JPanel board = keypad(3, 3, label);
     add(board, BorderLayout.CENTER);

     // Adiciona uma área de eco
     echoArea = new TextArea(2, 25);
     echoArea.setEditable(false);
     echoArea.setText("X goes first");
     add(echoArea, BorderLayout.SOUTH);
} // Controlador
```

| Figura 16.23 Inicialização do Controlador

```
private class NewGameHandler implements ActionListener {
     public void actionPerformed (ActionEvent e) {
          echoArea.setText("New Game button selected");
          newGame( );
          repaint( );
     }
}
```

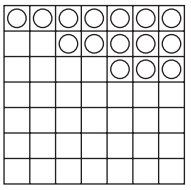
| Figura 16.24 O Manipulador de Eventos do Novo Jogo

Xs e os 0s, o tamanho do tabuleiro, quais ícones serão colocados para os botões e assim por diante. O controlador é grandemente pelo jogo.

Nim A criação de outros jogos interativos envolve considerações similares, seja o jogo Othello, Connect-4 ou Nim. Cada um deles requer que o programa mantenha controle do estado do jogo definindo variáveis e manipuladores de eventos apropriados de forma que o estado seja alterado da maneira exatamente correta quando cada evento ocorre.

Considere o jogo Nim de dois jogadores, que pode ser jogado em um tabuleiro que tem três fileiras de pedras, conforme mostrado abaixo:

Cada jogador, na sua vez, remove uma ou mais pedras de qualquer uma das três fileiras. O ganhador é o jogador que remover a última pedra. Um programa pode monitorar esse jogo usando um tabuleiro 7×7 no qual as três primeiras fileiras contêm Os nas localizações onde as pedras estão inicialmente colocadas, como mostramos abaixo:



O programa pode conduzir um diálogo com os dois jogadores usando uma área de texto para mostrar mensagens apropriadas. Ele deve permitir que cada jogador possa clicar em uma ou mais pedras em particular para serem removidas durante a sua jogada, mas não deve permitir que sejam tiradas pedras em duas fileiras diferentes na mesma jogada. Para marcar uma pedra "removida", o programa pode substituir o O por um espaço (OFF) na mesma posição no tabuleiro.

Para monitorar o jogo, o seu programa deverá ser capaz de:

1. Controlar de quem é a vez (Jogador 1 ou Jogador 2) e colocar essas informações na área de texto no início de cada jogada;
2. Aceitar uma série de cliques de mouse que juntos constituem um movimento permitido;

3 Determinar quando o jogo termina (e qual dos jogadores é o ganhador) e relatar esse evento na área de texto;

4 Permitir o início de um novo jogo a qualquer instante, talvez fornecendo uma caixa de seleção ou um botão que fará o manipulador restaurar o tabuleiro ao estado mostrado acima.

O programa deverá ter um método chamado `GameOver` que determina se o jogo terminou (o tabuleiro contém apenas espaços). Esse método deverá ser chamado pelo manipulador `mouseClicked` para dizer se deve ou não anunciar o vencedor.

O estado do programa, incluindo o jogo e o controlador, deverá conter o seguinte:

- Uma variável para o tabuleiro 7×7.
- Uma variável TextArea para enviar mensagens aos jogadores.
- Uma variável que determina de quem é a vez; essa variável alterna seu valor sempre que um jogador completar uma jogada permitida.
- Uma variável que registra a fileira em que o jogador pegou sua primeira pedra para um único movimento (pois todas as pedras subseqüentes devem ser apanhadas da mesma fileira).

O término do projeto dessa iteração é deixado como exercício.

16.4 OUTROS APLICATIVOS ACIONADOS POR EVENTOS

Programas acionados por eventos ocorrem em muitos outros domínios. Veja a seguir três exemplos:

- Um caixa eletrônico de banco.
- Um sistema de segurança residencial.
- Um caixa de supermercado.

Nesta seção, descrevemos rapidamente as duas primeiras interações, focalizando nossa atenção em suas variáveis de estado e nos tipos de eventos que serão manipulados no laço de eventos para manter a integridade entre as variáveis de estado.

16.4.1 Caixa Eletrônico de Banco

Um caixa eletrônico de banco funciona com um programa que roda sete dias por semana, 24 horas por dia. O programa deve ser capaz de executar as transações do usuário. Os elementos essenciais de um caixa eletrônico típico são mostrados na Figura 16.25. (Na prática, os detalhes são diferentes, mas os elementos mostrados aqui são adequados para caracterizar o que acontece durante uma transação em um caixa eletrônico.)

O estado dessa interação é capturado parcialmente pelos objetos nessa tela e parcialmente pelas outras informações relacionadas com o usuário particular do caixa eletrônico. Uma coleção básica de variáveis de estado necessárias para uma transação de caixa eletrônico inclui:

- O número da conta do usuário – `account`.
- O tipo de transação (depósito, retirada etc.) – `type`.
- O valor da transação – `amount`.
- Uma mensagem do banco para o usuário – `message`.
- O saldo disponível do usuário – `balance`.

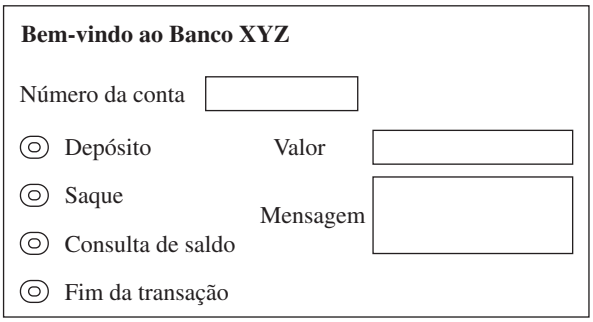


Figura 16.25
Elementos de uma Interface de Usuário de um Caixa Eletrônico

A última variável nesta lista, a variável saldo, mostra uma nova dimensão que não tínhamos encontrado em nossos outros exemplos de programação orientada a eventos. Nesse caso, o programa deve interagir não apenas com o usuário, mas também com a base de dados do banco de todas as suas contas e dos saldos atuais. Um programa que interage com uma base de dados como essa, que pode estar em um computador inteiramente diferente ou "servidor" na rede do banco, é chamado de aplicativo *cliente-servidor*. Os aplicativos cliente-servidor existem em uma grande variedade de sistemas, incluindo os sistemas de reservas de passagens aéreas, sistemas on-line para encomenda de livros e sistemas de inventário. Na Seção 17.4.2 há um exemplo cliente-servidor totalmente desenvolvido.

Retornando ao nosso exemplo, podemos agora caracterizar os diferentes eventos que podem ocorrer e também como eles devem ser manipulados (isto é, o efeito que eles deverão ter no estado da interação).

1 *Evento*: O usuário entra com um número de conta (passa o cartão).
Manipulado por: O programa verifica que o número da conta é válido, acessa o saldo e exibe a mensagem "Escolha uma transação".

2 *Evento*: O usuário seleciona um botão (depósito, saque etc.).
Manipulado por: O programa verifica se a conta é válida.

Em caso afirmativo, salva o tipo de botão selecionado.

- Se for depósito ou saque, mostra a mensagem "Digite o valor".
- Se o cliente pedir o saldo, mostra o saldo.
- Se não houver mais transações, encerra a operação.

Se não, mostra a mensagem "Número da conta".

3 *Evento*: O usuário digita o valor.
Manipulado por: O programa verifica se o usuário selecionou depósito ou saque.

Se for depósito, acrescenta o valor ao saldo da conta.

Se for saque, então:

- Se o saldo for maior do que aquele valor, subtrai a quantia do saldo.
- Caso contrário, mostra a mensagem "Saldo insuficiente".

Caso contrário, mostra a mensagem "Selecione um tipo de transação (depósito ou saque)".

O detalhe principal desse projeto é que o sistema não sabe com antecedência o tipo de transação ou a ordem em que os eventos ocorrerão. Ele responde a todas as diferentes possibilidades e atualiza o estado da interação corretamente.

16.4.2 Sistema de Segurança Doméstica

Os sistemas de segurança doméstica proporcionam ao dono da residência um simples aviso de que ocorreu um evento indesejado, como um arrombamento, um incêndio ou um alagamento. Esses sistemas são programados para reagir a sensores que podem detectar fumaça, água ou movimento, os quais são posicionados em lugares estratégicos em toda a casa. Eles podem ser programados pelos proprietários das residências de forma que a movimentação destes na casa não seja considerada como um arrombamento. Eles podem também ser conectados ao quartel local dos bombeiros e à delegacia de polícia, de maneira que a notificação de um evento indesejado possa receber uma resposta imediata.

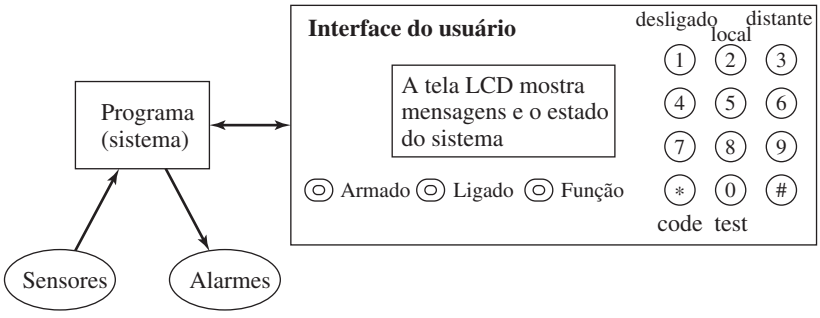
O projeto geral para um sistema desses está esboçado na Figura 16.26. Aqui, os sensores e a interface de usuário fornecem eventos ao programa, enquanto os alarmes e a interface de usuário recebem respostas do programa. Respostas na interface de usuário são mostradas em uma tela LCD. Como esse sistema deve ser capaz de manipular eventos paralelamente (por exemplo, sinais de dois sensores diferentes podem ocorrer simultaneamente), o programa deve manipular ambos os eventos (conforme discutimos neste capítulo) e a concorrência (discutida no Capítulo 17). Ignoramos as dimensões da programação concorrente desse problema na discussão atual.

As variáveis de estado para esse programa incluem:

- A senha do usuário – `password`.
- O estado do usuário (aqui ou distante) – `user`.
- O estado do sistema (armado ou desarmado) – `armed`.
- O estado de cada sensor (ativo ou inativo) – `sensors`.
- O estado de cada alarme (ativo ou inativo) – `alarms`.
- Uma mensagem do sistema para o painel de controle – `message`.

Generalizamos ao extremo essa descrição para simplificar a discussão e focalizar nossa atenção nas idéias principais sobre a programação orientada a eventos relacionada a esse problema. Em todo caso, aqui estão alguns dos eventos que podem ocorrer, com um esboço do que poderia acontecer com o estado em resposta a cada evento.

- *Evento*: O usuário digita a senha.
  Manipulado por: O programa verifica se a senha é válida e mostra uma mensagem.
- *Evento*: O usuário entra na função "away" (distante).
  Manipulado por: O programa verifica se a senha foi fornecida e muda o estado de todos os sensores para "ativo", o estado do sistema para "armado" e mostra uma mensagem.



| Figura 16.26 Projeto Geral de um Sistema de Segurança Doméstica

- *Evento*: O sensor recebe o sinal.
  Manipulado por: Se o sistema estiver armado, o programa dispara o alarme apropriado e mostra a mensagem.
- *Evento*: O usuário entra na função "teste".
  Manipulado por: O programa desarma o sistema.
- *Evento*: O usuário entra na função "local".
  Manipulado por: O programa desabilita os sensores de movimento, habilita todos os outros e muda o estado do sistema para "armado".

16.5 RESUMO

A programação orientada a eventos tornou-se difundida nas aplicações modernas, principalmente devido ao aparecimento das interfaces gráficas de usuário (GUIs). Essa técnica de projeto difere dos paradigmas convencionais porque ela permite que a entrada ocorra em seqüências imprevisíveis e porque seus programas são projetados para rodar continuamente.

Java proporciona um grande suporte para a programação orientada a eventos, embora ela não seja a única linguagem usada para essa finalidade. Várias outras linguagens, como Tcl/Tk, também têm recursos para suportar a manipulação de eventos.

Programas GUI acionados por evento podem ser escritos para uma ampla variedade de aplicações, como jogos interativos e dispositivos de pesquisa on-line. O padrão de projeto modelo–visualização–controle (MVC) é particularmente útil para o projeto desses programas.

EXERCÍCIOS

16.1 A variável `TextArea` é rolável, de forma que todas as ações na execução desse programa podem ser ecoadas sem perda de informação. Como você pode refinar a interação na Seção 16.3.1 para que isso aconteça?

16.2 Amplie a interação simples acionada por eventos dada na Seção 16.3.1 de maneira que o menu da caixa de seleção inclua linhas e ovais, além de retângulos e mensagens.

16.3 Considere a ampliação desse programa novamente de forma que ele tenha uma função para apagar. Discuta a diferença entre apagar a forma desenhada mais recentemente na tela e o apagamento de objetos na ordem inversa em que eles foram desenhados. Que mudanças são necessárias na definição do estado do programa para conseguir esse objetivo mais ambicioso?

16.4 Escreva um aplicativo que implemente o seguinte:

- Contenha caixas de menu popup com três opções.
- Uma opção defina a cor: vermelho, verde, amarelo, azul.
- Outra defina a figura a ser desenhada: círculo ou quadrado.
- Outra defina o tamanho do círculo ou quadrado: 10, 20, 30 pixels. Tamanho é o diâmetro de um círculo ou a altura e a largura de um quadrado.
- Quando o mouse for clicado, seja desenhado um círculo ou um quadrado, e seja preenchido com o tamanho e a cor apropriados nas coordenadas (x, y) do clique do mouse, se a figura a ser traçada for um círculo ou um quadrado.

- As figuras desenhadas deverão sobreviver ao redesenho, isto é, o aplicativo que está sendo minimizado e maximizado. Dica: você precisa de uma `List` (lista) de figuras gráficas; o seu método `paintComponent` deve solicitar que cada figura na `List` se redesenhe.

16.5 Considere a seguinte configuração para o tabuleiro do jogo-da-velha:

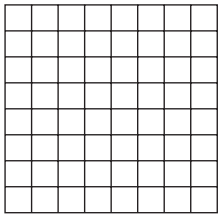
Descreva a seqüência de eventos que devem ocorrer para que isso seja conseguido, começando com um tabuleiro vazio. Para cada evento, identifique o manipulador que é chamado e a alteração que ocorre entre as variáveis de estado que são afetadas por aquele evento.

16.6 Pense em uma estratégia pela qual o seu programa pode suportar um jogo-da-velha de um ou dois jogadores. Se o usuário escolher a opção de um jogador (por exemplo, por meio de um menu), o programa deverá assumir o papel do Jogador X e fazer um jogo competitivo contra o usuário – isto é, o Jogador X nunca deve perder.

16.7 Implemente o programa Nim acionado por eventos declarando as variáveis de estado e implementando os métodos `init`, `mouseClicked`, `actionPerformed` e `gameOver`.

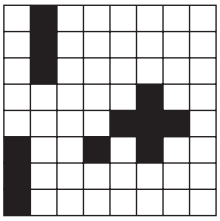
16.8 (Difícil) Pense em uma estratégia pela qual o seu programa Nim assume o papel do Jogador 1 e faz um jogo competitivo do Nim. Você pode pesquisar esse problema antes de começar a escrever o código. (Só resolva esse problema depois que você tiver completado a questão anterior.)

16.9 O jogo Connect-4 tem dois jogadores que, alternadamente, colocam Xs e Os em um tabuleiro 8×8, como esse mostrado abaixo.



No entanto, cada movimento deve ser feito no quadrado vazio que estiver mais abaixo de uma coluna. O vencedor é aquele que conseguir colocar quatro *tokens* (X ou O) em uma fileira, que pode ser horizontal, vertical ou diagonal. Projete uma interação acionada por evento para esse jogo. Ao fazer isso, comece projetando as variáveis de estado, os objetos no quadro e, finalmente, os manipuladores de evento que são necessários para monitorar efetivamente o jogo. Você precisa ter um método que determine quando o jogo termina (um jogador colocou quatro *tokens* em uma fileira ou o tabuleiro ficou cheio).

16.10 (Trabalho de Equipe) Considere o projeto de uma interação controlada por eventos que demonstra o crescimento da população. Essa interação apresenta um modelo de crescimento e uma queda populacional denominado Jogo da Vida, que foi criado por John Conway em 1970. Ele usa uma grade para mostrar uma população de indivíduos em uma única geração, na forma da grade 8×8 mostrada a seguir.



Um quadrado preenchido na grade indica um indivíduo vivo e fica vazio para indicar a falta de um indivíduo vivo. Cada quadrado da grade tem até oito "vizinhos" imediatos, um em cada lado e um em cada diagonal. Por exemplo, os oito vizinhos da célula na posição (2, 3) – isto é, fileira 2 e coluna 3 – são as células nas posições (1, 2), (1, 3), (1, 4), (2, 2), (2, 4), (3, 2), (3, 3) e (3, 4).

A interação controla o que acontece na grade, de uma geração para a próxima, por meio de uma série de gerações. Começamos com uma geração inicial como aquela mostrada acima. Para prever quais as células que representarão indivíduos vivos na próxima geração, aplicam-se as seguintes regras a cada célula:

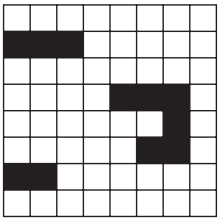
Sobrevivente. Um indivíduo sobrevive para a próxima geração se ele tiver dois ou três indivíduos vizinhos vivos na geração atual.

Nascimento. Um indivíduo nasce na próxima geração se (a) sua célula está vazia na geração atual e (b) a sua célula tem exatamente três indivíduos vivos vizinhos na geração atual.

Morte por solidão. Um indivíduo morre na próxima geração se ele tiver menos de dois vizinhos vivos na geração atual.

Morte por superpopulação. Um indivíduo morre na próxima geração se ele tiver quatro ou mais vizinhos vivos na geração atual.

A grade abaixo mostra o resultado quando se aplicam essas quatro regras a cada uma das células da grade acima.



Como você pode ver, várias células sobreviveram (por exemplo, aquela na posição (1, 1)), várias nasceram (por exemplo, aquela na posição (1, 0)), morreram de solidão (por exemplo, aquela na posição (3, 5)) ou morreram por superpopulação (por exemplo, aquela na posição (5, 4)). Outras células permaneceram desocupadas da primeira geração para a segunda.

Representando esse processo por uma série de gerações, obtemos alguns padrões interessantes de crescimento e diminuição populacional. Por exemplo, o padrão no canto superior esquerdo da grade se repetirá por uma série de gerações, enquanto aquele no canto inferior esquerdo desaparecerá completamente após a segunda geração. O padrão no meio da tela mudará sua forma e sua posição de uma maneira interessante através de uma série de gerações.

Para visualizar essas mudanças populacionais, a sua interação deverá ter duas grades lado a lado no tabuleiro, e então pedir ao usuário que defina a primeira geração clicando em uma série de

quadrados na grade da esquerda. O término da atividade pode ser sinalizado pelo usuário clicando fora da grade ou talvez selecionando uma alternativa diferente em uma caixa de seleção na tela. Em seguida, o usuário deverá poder pedir ao programa que calcule e mostre a segunda geração na grade da direita, a terceira geração na grade da esquerda, a quarta geração à direita, e assim por diante. O fim da simulação ocorre com um ou mais dos seguintes estados:

(1) Três grades inteiras ficam vazias (toda a geração morreu);
(2) Duas gerações sucessivas são idênticas nas suas populações; ou
(3) O usuário seleciona um botão que interrompe a simulação.

O usuário deverá também ter a oportunidade de iniciar uma nova simulação; nesse evento, o programa deverá limpar ambas as grades e permitir que o usuário inicie um novo jogo selecionando uma configuração inicial diferente na grade da esquerda.

16.11 Faça uma pesquisa na Internet e aprenda sobre a programação de um sistema real de segurança residencial. Que linguagem de programação é usada na implementação de um sistema desses? Como é usado o modelo acionado por eventos no programa que controla esse sistema?

17 Programação Concorrente

"Duas estradas divergiam por um bosque amarelo,
E lamentavelmente eu não pude pegar ambas . . ."

Robert Frost, *The Road Not Taken*

VISÃO GERAL DO CAPÍTULO

Superficialmente, a idéia de duas partes de um programa, ou mesmo de dois programas separados, executando *concorrentemente* parece bastante simples. A programação para concorrência pode ocorrer em muitos níveis de linguagem – desde o nível mais baixo de lógica digital até o nível de aplicativo (por exemplo, um navegador para a Internet). A concorrência também ocorre nos quatro paradigmas de programação – imperativa, orientada a objetos, funcional e lógica.

Não importa qual seja a perspectiva, uma introdução da concorrência em uma aplicação complexa torna-a mais realística no sentido de que ela modela a realidade melhor do que se a concorrência fosse descartada. Além disso, a introdução da concorrência em um programa pode poupar uma quantidade enorme de recursos de computação, tanto em espaço quanto em velocidade. Sem dúvida, muitas aplicações importantes da computação – como a modelagem oceânica, navegadores para Internet ou mesmo um sistema operacional – não seriam concebíveis se a concorrência não fosse um elemento central de seus projetos.

No entanto, a programação concorrente em qualquer nível ou domínio de aplicação traz complexidades especiais e fundamentais. Em particular, se dois elementos de programas concorrentes precisam se comunicar um com o outro, como essa comunicação será coordenada? E mais: se dois desses elementos precisam compartilhar o valor comum de um dado, com a possibilidade de que nenhum dos dois possa sobrescrever aquele valor, como esse compartilhamento será racionalizado?

Tradicionalmente, o estudo da concorrência ocorre no ambiente de um curso de sistemas operacionais, no qual o gerenciamento de eventos simultâneos em diferentes processos ativos é uma preocupação principal. No entanto, o projeto de aplicações com a concorrência embutida tem se tornado cada vez mais importante, já que os projetistas de software agora rotineiramente modelam sistemas com componentes que interagem de forma assíncrona e compartilham dados. Portanto, é importante nos cursos de linguagem de programação aprender sobre as etapas que os projetistas de linguagem tiveram de percorrer para satisfazer a essa necessidade agora tão difundida na programação.

Devido à vasta gama de tópicos que ocorrem quando o assunto é a concorrência, este capítulo focalizará somente a concorrência nos níveis de linguagem e projeto de software, deixando outros aspectos da concorrência para outros cursos. Além disso, este capítulo trata da concorrência em apenas dois cenários:

- Um que utiliza um único processador;
- Um que utiliza a comunicação interprocessos (IPC).

No primeiro caso, um programa roda em um único processador, mas de tempos em tempos ele pode dinamicamente se dividir em *threads* (seqüências de execução do programa) concorrentes de controle. No segundo caso, um programa é visto como uma coleção de processos em cooperação que rodam em uma rede e compartilham dados. Um tipo de aplicação IPC é chamado aplicativo *client-server* (cliente-servidor), no qual um único processo-servidor compartilha informações com vários processos-cliente e roda simultaneamente e de forma autônoma em máquinas separadas na Internet.

Este capítulo ilustra cada um desses cenários junto com a coordenação e as estratégias de compartilhamento de dados que estão por trás de sua efetiva implementação. Para proporcionar um foco para essa discussão, nossos exemplos usam Java como linguagem para a ilustração. As características da concorrência em várias outras linguagens estão resumidas no fim do capítulo.

17.1 CONCEITOS DE CONCORRÊNCIA

O Capítulo 13 sugeria que uma boa maneira de pensar em um programa orientado a objetos é uma coleção de objetos interagindo que se comunicam pelo uso da *passagem de mensagens* via métodos. Assim, cada objeto é visto como uma máquina separada que encapsula tanto os dados quanto as operações sobre aqueles dados. Diferentes objetos que são instâncias da mesma classe compartilham o mesmo conjunto de operações, mas seus dados diferem.

Por exemplo, um programa usando a classe *stack* (pilha) pode conter vários objetos diferentes que são *stacks* (pilhas), cada uma com sua própria coleção de dados, porém compartilhando o mesmo conjunto em comum de operações específicas de uma pilha

(push, pop etc.). Levando esse modelo um pouco mais adiante, podemos agora explorar a maneira como todos os objetos podem estar executando simultaneamente, enviando e recebendo mensagens dos outros objetos no programa.

Um exemplo mais prático de um programa concorrente orientado a objetos é o moderno *web browser*. A concorrência em um *web browser* ocorre quando o *browser* começa a renderizar uma página. A página que está sendo renderizada é um recurso compartilhado que deve ser gerenciado cooperativamente pelas várias *threads* (seqüências) envolvidas em fazer o download e a renderização de seus diferentes aspectos. Por exemplo, enquanto o *browser* ainda está fazendo o download de um arquivo de imagem ou um arquivo gráfico, ele pode estar renderizando outra imagem em uma localização diferente na tela.

No entanto, todas essas várias *threads* não podem escrever simultaneamente imagens na tela, especialmente se a imagem ou o gráfico que está sendo baixado causa o redimensionamento do espaço alocado para mostrar a imagem, afetando assim o arranjo do texto. Ao mesmo tempo em que faz tudo isso, vários botões, como o botão `Stop`, ainda estão ativos e podem ser clicados pelo usuário.

Nesta seção, exploramos como diferentes *threads* podem cooperar para executar uma tarefa como a renderização de uma página da Internet. Em certos momentos, as *threads* devem ter acesso exclusivo a um recurso compartilhado, como a tela de um monitor, para impedir que outras *threads* interfiram nela. Por exemplo, a *thread* que redimensiona uma página da Internet na tela precisa ter acesso exclusivo àquela página durante todo o tempo necessário para sua tarefa de redimensionamento.

17.1.1 História e Definições

A execução concorrente dos processos em um programa pode ocorrer pela distribuição desses processos para processadores separados ou intercalando-os em um único processador, usando fatias de tempo. O fatiamento de tempo divide o tempo em pequenos blocos e distribui esses blocos entre os processos de uma maneira uniformizada. Os primeiros computadores às vezes tinham um único processador principal e vários outros processadores menores para fazer a entrada e a saída separadamente. Os programas nessas primeiras máquinas eram executados em lote, cada programa era executado até o fim e só depois era executado o próximo.

Mais tarde, foi introduzida a programação múltipla, para aumentar a eficiência. Na *programação múltipla*, vários programas são carregados na memória e executados de uma forma intercalada, sendo usado um *scheduler* (escalonador) para mudar o controle de um programa para outro. À medida que o custo dos computadores diminuiu em relação ao custo do trabalho, foi introduzido o conceito de *tempo compartilhado interativo*. O compartilhamento do tempo permite que duas ou mais pessoas usem teclados e monitores para se comunicar simultaneamente com o computador. O sistema operacional Unix foi desenvolvido nos anos 1960 para suportar operações em tempo compartilhado em computadores relativamente baratos.

A programação concorrente foi usada nos primeiros sistemas operacionais para suportar o paralelismo no hardware e para suportar a programação múltipla e o tempo partilhado. Mas, por longo tempo, a programação concorrente foi considerada como muito difícil e suscetível a erros, para ser usada por aplicações comuns. No entanto, o nosso conhecimento de programação concorrente e os métodos de linguagem de programação usados para suportá-la evoluíram de maneira que muitas aplicações modernas são *multithread*. Em computadores com dois ou mais processadores, essas aplicações podem ter cada *thread* executada por um processador separado, conseguindo-se assim uma significativa aceleração de processamento.

As definições e os conceitos a seguir possuem uso bastante difundido. Enquanto um *programa* representa a execução de uma seqüência de instruções, um *processo* representa uma única seqüência de instruções em execução. Como tal, cada processo tem seu próprio estado independente do estado de qualquer outro processo, incluindo processos do sistema operacional.

Um processo também tem recursos anexados como arquivos, memória e portas da rede. O estado de um processo inclui sua memória e a localização da instrução corrente que está sendo executada. Essa noção estendida de estado é chamada de *contexto de execução.*

> **Definição**: Um *programa concorrente* é um programa projetado para ter dois ou mais contextos de execução. Chamamos um programa desses de *multithread,* porque mais de um contexto de execução pode estar ativo simultaneamente.

Um *programa paralelo* é um programa concorrente no qual vários contextos de execução, ou *threads*, estão ativos *simultaneamente*. Para nossa finalidade, não há diferença entre um programa concorrente e um programa paralelo.

Um *programa distribuído* é um programa concorrente projetado para ser executado simultaneamente em uma rede de processadores autônomos que não compartilham a memória principal, com cada *thread* rodando em seu próprio processador separado. Em um sistema operacional distribuído, por exemplo, o mesmo programa (como um editor de texto) pode ser executado por vários processadores, com cada instância tendo seu próprio contexto de execução separado dos outros (ou seja, sua própria janela). Isso não é o mesmo que um programa *multithread*, no qual os dados podem ser compartilhados entre diferentes contextos de execução.

O termo *concorrência* então representa um programa em que dois ou mais contextos de execução podem estar ativos simultaneamente. Cada um dos programas que temos visto até agora tem um único contexto de execução e, portanto, são chamados de programas *single-threaded.* Um programa com múltiplos contextos de execução é chamado de *multi-threaded.* Em um programa *multi-threaded*, parte do estado do programa é compartilhado entre as *threads*, enquanto parte do estado (incluindo o controle de fluxo) é único para cada *thread*.[1]

17.1.2 Controle de *Thread* e Comunicação

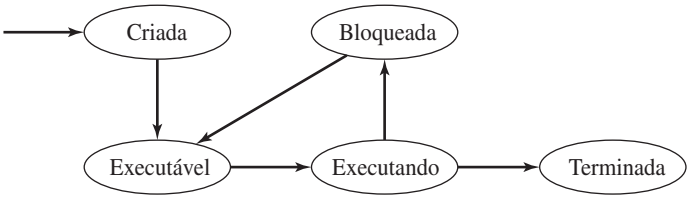
Tanto em Java quanto em Ada, uma *thread* está associada a um método separado (função), em vez de a uma única instrução. No entanto, para iniciar uma *thread* são necessárias ações adicionais, além de chamar uma função. Primeiro, quem chama uma nova *thread* não espera que ela se complete antes de continuar; ele passa à execução das instruções que seguem a chamada da *thread*. Segundo, quando a *thread* termina, o controle não retorna para quem a chamou.

Para entender mais concretamente o controle de *threads*, devemos introduzir a idéia de *estado* de uma *thread*. Uma *thread* pode estar em um dos seguintes estados:

1. *Criada*: a *thread* foi criada, mas ainda não está pronta para rodar.
2. *Executável*: a *thread* está pronta para rodar (às vezes, esse estado é chamado de *pronto*). A *thread* aguarda por um processador no qual possa ser executada.[2]
3. *Executando*: a *thread* está sendo executada em um processador.
4. *Bloqueada*: a *thread* está esperando para entrar em uma seção de seu código que requer acesso exclusivo a um recurso compartilhado (variável), ou então voluntariamente ela cedeu o uso do seu processador.
5. *Terminada*: a *thread* já parou e não pode ser reiniciada.

---

1. Se dois ou mais componentes de software de comunicação rodam de forma concorrente, o resultado é um programa concorrente, se as peças formam um todo conceitual. Caso contrário, a situação é vista como dois programas autônomos que podem se comunicar por meio de um protocolo comum como um sistema de arquivos ou um protocolo de rede. Se os programas se comunicam, eles formam um sistema concorrente.
2. Do ponto de vista de uma linguagem de programação, *Executável* e *Executando* podem ser combinados em um único estado.



| Figura 17.1 Estados de uma *Thread*

Uma seção de código que requer acesso exclusivo a uma variável compartilhada é chamada de *seção crítica* de uma *thread.*

Esses estados e as possíveis transições entre eles são representados na Figura 17.1. Repare, em particular, que uma *thread* pode ir e voltar entre os estados *Bloqueada* e *Executando* várias vezes durante seu tempo de vida. Por exemplo, uma *thread* pode estar mandando vários documentos para uma fila de impressão, mas pode ter de esperar até que um documento seja impresso para depois mandar outro.

Programas concorrentes requerem comunicação ou interação *interthreads*. A comunicação ocorre pelas seguintes razões:

1. Uma *thread*, às vezes, requer acesso exclusivo a um recurso compartilhado, como uma fila de impressão, por exemplo, uma janela de um terminal ou um registro em um arquivo de dados.
2. Uma *thread*, às vezes, precisa trocar dados com outra *thread*.

Em ambos os casos, as duas *threads* em comunicação devem sincronizar sua execução para evitar conflito ao adquirir recursos ou para fazer contato ao trocar dados. Uma *thread* pode se comunicar com outras *threads* por meio de:

1. Variáveis compartilhadas: esse é o mecanismo primário usado em Java, e também pode ser usado em Ada.
2. Passagem de mensagens: esse é o mecanismo primário usado em Ada.[3]
3. Parâmetros: são usados em Ada em conjunto com a passagem de mensagens.

Threads normalmente cooperam umas com as outras para resolver um problema. No entanto, é altamente desejável manter a comunicação entre *threads* em um nível mínimo; isso torna o código mais fácil de entender e deixa cada *thread* rodar em sua própria velocidade sem ser retardada por protocolos complexos de comunicação.

17.1.3 Corridas e *Deadlocks*

Dois problemas fundamentais que podem ocorrer ao executar duas diferentes *threads* assincronicamente são *condições de corrida* e *deadlocks*.

> **Definição**: Uma *condição de corrida* (às vezes chamada *corrida crítica*) ocorre quando o valor resultante de uma variável pode ser diferente, quando duas diferentes *threads* de um programa estão modificando essa mesma variável e dependendo de qual *thread* altera primeiro essa variável.

3. Ada usa um mecanismo de comunicação interthread chamado *rendezvous*. O projeto desse mecanismo foi fortemente influenciado pelo trabalho de Hoare sobre Communicating Sequential Processes (CSP) (Hoare, 1978), (Hoare, 1985). CSP fornece uma teoria completa sobre a concorrência, e atualmente estão sendo feitos esforços no sentido de adicionar características de programação no estilo CSP para Java.

Isso significa que o resultado é determinado por qual das *threads* ganha a "corrida" (race) entre elas, já que isso depende da ordem em que as operações individuais são intercaladas no tempo.

Por exemplo, considere duas *threads* compartilhando uma variável c, e ambas tentando executar a seguinte instrução assincronicamente:

```
c = c + 1;
```

Em uma máquina estilo JVM, o código para essa instrução poderia ser:

```
1. load   c
2. add    1
3. store  c
```

Suponha que cada uma das instruções acima seja atômica, mas que uma *thread* pode ser parada entre quaisquer duas delas.

Se o valor inicial de `c` for 0, então qualquer um dos valores finais 1 ou 2 para `c` é possível. Por exemplo, suponha que as duas *threads* sejam A e B. Então o valor resultante de `c` depende criticamente de A ou B completar o passo 3 antes de a outra começar o passo 1. Em um caso, o valor resultante de `c` é 2, no outro, o valor resultante de `c` é 1.

De fato, à medida que aumenta o número de *threads* tentando executar esse código, o número resultante de valores distintos computados para `c` pode variar entre 1 e o número de *threads*! Essa é uma situação computacional inaceitável.

Uma *thread* que quer adquirir um *recurso compartilhado*, como um arquivo ou uma variável compartilhada (como `c` no exemplo acima), primeiro tem de adquirir acesso ao recurso. Quando o recurso não é mais necessário, a *thread* deve renunciar o acesso ao recurso para que as outras *threads* possam acessá-lo. Se uma *thread* não é capaz de adquirir um recurso, sua execução é normalmente suspensa até que o recurso se torne disponível. A aquisição de um recurso deve ser administrada de forma que nenhuma *thread* seja atrasada indevidamente nem seja negado o acesso a um recurso de que ela precisa. Uma ocorrência desse último tipo às vezes é chamada de *lockout* ou *inanição*. Na próxima seção, veremos as estratégias para evitar o *lockout*.

Erros que ocorrem em um programa concorrente podem aparecer como *erros transientes*. São erros que podem ocorrer ou não, dependendo dos caminhos de execução das várias *threads*. A localização de um erro transiente pode ser extremamente difícil porque a seqüência de eventos que causou a ocorrência da falha pode não ser conhecida ou reproduzível. Diferentemente da programação seqüencial, rodar novamente o mesmo programa com os mesmos dados pode não reproduzir a falha. A geração de informações de *debug* pode por si só alterar o comportamento do programa concorrente de forma a evitar que a falha ocorra novamente. Assim, uma habilidade importante no projeto de um programa concorrente é a possibilidade de expressá-lo de forma que garanta a ausência de corridas críticas.

O código dentro de uma *thread* que acessa uma variável compartilhada ou outro recurso é chamado de *seção crítica.* Para uma *thread* executar seguramente uma seção crítica, ela precisa ter acesso a um mecanismo de travamento; esse mecanismo deve permitir que um bloqueio seja testado ou definido como uma única instrução atômica. Mecanismos de travamento são usados para assegurar que somente uma única *thread* esteja executando uma seção crítica (portanto, acessando uma variável compartilhada) de cada vez; isso pode eliminar condições de corrida crítica, como aquela ilustrada acima. Um mecanismo de bloqueio como esse é chamado de *semáforo* e será ilustrado mais tarde.

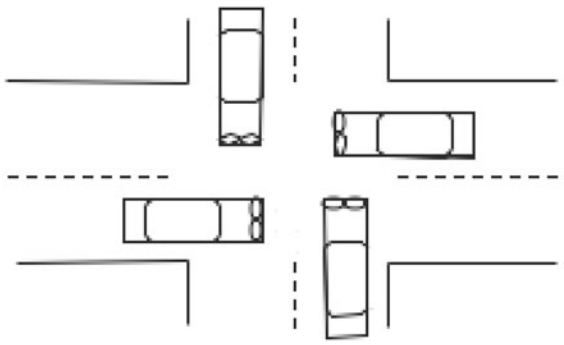


Figura 17.2
***Deadlock* em uma Intersecção**

O segundo problema fundamental que pode ocorrer ao executar duas *threads* diferentes de forma assíncrona é chamado de *deadlock*.

> **Definição**: Um *deadlock* ocorre quando uma *thread* estiver esperando por um evento que nunca acontecerá.

Um *deadlock* normalmente envolve várias *threads*, cada uma delas esperando para ter acesso a um recurso retido por uma outra *thread*. Um exemplo clássico de *deadlock* é um engarrafamento de tráfego em uma intersecção onde cada carro que está entrando é bloqueado por outro, como mostra a Figura 17.2.

Quatro condições necessárias devem ocorrer para que exista um *deadlock* (Coffman et al., 1971):

1. As *threads* precisam reclamar direitos exclusivos aos recursos.
2. As *threads* devem reter alguns recursos enquanto esperam por outros; isto é, elas adquirem recursos gradativamente e não todos de uma só vez.
3. Os recursos não podem ser removidos das *threads* que estão esperando (não há apropriação antecipada).
4. Existe uma cadeia circular de *threads* na qual cada *thread* retém um ou mais recursos necessários para a próxima *thread* na cadeia.

As técnicas para se evitar ou recuperar-se de *deadlocks* têm como base a negação de pelo menos uma dessas condições. Uma das melhores técnicas para se evitar o *deadlock*, embora nada prática, é o Algoritmo do Banqueiro (Banker's Algorithm) (Dijkstra, 1968a). Dijkstra também imaginou um problema conhecido como o Problema do Jantar dos Filósofos – em inglês, Dining Philosophers' Problem (Dijkstra, 1971) –, cuja solução se tornou uma ilustração clássica da prevenção do *deadlock*. Esse problema está colocado como um exercício.

Dizemos que uma *thread* é *adiada indefinidamente* se ela for atrasada esperando por um evento que pode nunca ocorrer. Essa situação pode ocorrer se o algoritmo que aloca recursos para as *threads* que os requisitam não tiver nenhuma consideração pelo tempo de espera de uma *thread*. A alocação de recursos pelo critério "primeiro que entra, primeiro que sai" é uma solução simples, que elimina o adiamento indefinido.

Análogo ao adiamento indefinido é o conceito de *injustiça.* Nesse caso, não é feita nenhuma tentativa para garantir que as *threads* de mesmo *status* tenham o mesmo progresso ao adquirir recursos. Justiça em um sistema concorrente deve ser considerada como um projeto. Um critério simples de justiça é aquele que ocorre quando se podem fazer várias escolhas para uma ação, cada alternativa deverá ser igualmente provável. O descuido com a justiça ao projetar um sistema concorrente pode levar ao adiamento indefinido, tornando o sistema não-utilizável.

17.2 ESTRATÉGIAS DE SINCRONIZAÇÃO

Foram desenvolvidos dois dispositivos principais que suportam programação para concorrência: semáforos e monitores. Eles serão discutidos separadamente em cada uma das seções a seguir.

17.2.1 Semáforos

Os semáforos originalmente foram definidos por Dijkstra (1968a). Basicamente, um semáforo é uma variável inteira e um mecanismo associado de enfileiramento de *threads*. Duas operações atômicas, tradicionalmente chamadas de *P* e *V*, são definidas para um semáforo `s`:

- *P(s)* – se $s > 0$ então atribui `s = s - 1`; caso contrário, bloqueia (enfileira) a *thread* que chamou *P*.
- *V (s)* – se uma *thread T* é bloqueada no semáforo `s`, então acorda *T*; caso contrário, atribui `s = s + 1`.

As operações *P* e *V* são atômicas no sentido de que elas não podem ser interrompidas, uma vez que tenham sido iniciadas. Se o semáforo somente assume os valores 0 e 1, ele é chamado de semáforo *binário*. Caso contrário, ele é chamado de *semáforo de contagem.*

Uma aplicação de um semáforo binário ocorre quando duas tarefas usam semáforos para sinalizar uma à outra, quando há trabalho para a outra fazer; esse processo, às vezes, é chamado *sincronização cooperativa*. Um exemplo clássico ocorre no caso da cooperação produtor-consumidor, em que uma única tarefa do produtor deposita informações em um *buffer* compartilhado, de entrada única, para que essas informações sejam recuperadas por uma única tarefa consumidora. A *thread* do produtor espera (por meio de uma operação *P*) que o *buffer* fique vazio, deposita a informação, depois sinaliza (por meio de uma operação *V*) que o *buffer* está cheio. A *thread* do consumidor espera (por meio de uma operação *P*) que o *buffer* fique cheio, então remove as informações do *buffer* e sinaliza (por meio de uma operação *V*) que o *buffer* está vazio. O código para esse exemplo em Concurrent Pascal está na Figura 17.3.

Um caso mais complicado ocorre quando temos vários produtores e consumidores compartilhando um *buffer* de múltiplas entradas (mas de tamanho finito). Nesse caso, é insuficiente para um produtor, por exemplo, saber que o *buffer* não está cheio. Ele precisa também impedir o acesso de outros produtores enquanto ele deposita suas informações. O protocolo usual para isso é dado na Figura 17.4, na qual o produtor primeiro testa (por meio de uma operação *P*) se o *buffer* não está cheio e depois bloqueia a seção crítica (por meio de outra operação *P* no semáforo de bloqueio). Se o produtor tivesse executado esses dois passos na ordem inversa, ele teria produzido um *deadlock,* pois todas as outras *threads* estariam impedidas de rodar. Repare que semáforos não-cheios/não-vazios são semáforos de contagem, enquanto o semáforo de bloqueio é um semáforo binário.

O produtor primeiro produz informações localmente. Depois ele se certifica de que o *buffer* está não-cheio, fazendo uma operação *P* no semáforo `nonfull`; se o *buffer* não estiver cheio, um ou mais produtores continuarão. Em seguida, o produtor precisa de acesso exclusivo às variáveis do *buffer* compartilhado. Para obter esse acesso, o produtor executa uma operação *P* no semáforo binário `lock`. Passar além desse ponto garante acesso exclusivo às diversas variáveis do *buffer* compartilhado. O produtor deposita suas informações e sai da seção crítica executando uma operação *V* no semáforo de bloqueio. Finalmente, o produtor sinaliza às *threads* consumidoras que o *buffer* não está vazio por meio de uma operação *V* no semáforo `nonempty`. O código para o consumidor é similar.

Embora o semáforo seja um mecanismo elegante, de baixo nível, para o controle da sincronização, não iríamos querer criar um sistema grande, multitarefa, como um sistema

```
program SimpleProducerConsumer;
var buffer : string;
    full : semaphore = 0;
    empty : semaphore = 1;

procedure Producer;
var tmp : string
begin
    while (true) do begin
        produce(tmp);
        P(empty); { begin critical section }
        buffer := tmp;
        V(full); { end critical section }
    end;
end;

procedure Consumer;
var tmp : string
begin
    while (true) do begin
        P(full); { begin critical section }
        tmp := buffer;
        V(empty); { end critical section }
        consume(tmp);
    end;
end;

begin
    cobegin
        Producer; Consumer;
    coend;
end.
```

| Figura 17.3 Cooperação Simples Produtor-Consumidor Usando Semáforos

operacional, usando semáforos. Isso resulta do fato de que a omissão de uma única operação *P* ou *V* poderia ser catastrófica.

17.2.2 Monitores

Os *monitores* (Hoare, 1974) proporcionam um dispositivo alternativo para gerenciar a concorrência e evitar o *deadlock*. Os monitores proporcionam a base para a sincronização em Java. O conceito de um monitor é baseado no monitor ou kernel dos primeiros sistemas operacionais; lá ele era usado como um método de comunicação entre *threads* do sistema operacional. Esses primeiros monitores rodavam em modo privilegiado e eram não-interrompíveis.

```
program ProducerConsumer;
const size = 5;
var buffer    :  array[1..size] of string;
    inn       :  integer = 0;
    out       :  integer = 0;
    lock      :  semaphore = 1;
    nonfull   :  semaphore = size;
    nonempty  :  semaphore = 0;

procedure Producer;
var tmp : string
begin
    while (true) do begin
        produce(tmp);
        P(nonfull);
        P(lock); { begin critical section }
        inn := inn mod size + 1;
        buffer[inn] := tmp;
        V(lock); { end critical section }
        V(nonempty);
    end;
end;

procedure Consumer;
var tmp : string
begin
    while (true) do begin
        P(nonempty);
        P(lock); { begin critical section }
        out = out mod size + 1;
        tmp := buffer[out];
        V(lock); { end critical section }
        V(nonfull);
        consume(tmp);
    end;
end;
```

| Figura 17.4 Múltiplos Produtores e Consumidores

O monitor desenvolvido por Hoare é uma versão descentralizada dos primeiros monitores de sistemas operacionais. Sua finalidade é encapsular uma variável compartilhada com operações primitivas (`signal` e `wait`) sobre aquela variável, e então proporcionar um mecanismo de bloqueio automático sobre essas operações de maneira que no máximo uma *thread* possa estar executando uma operação a cada vez. Nosso exemplo produtor-consumidor está remodelado na Figura 17.5, usando um monitor em vez de semáforos.

```
monitor Buffer;
const size = 5;
var buffer    : array[1..size] of string;
    in        : integer = 0;
    out       : integer = 0;
    count     : integer = 0;
    nonfull   : condition;
    nonempty  : condition;

procedure put(s : string);
begin
    if (count = size) then
       wait(nonfull);
    in := in mod size + 1;
    buffer[in] := tmp;
    count := count + 1;
    signal(nonempty);
end;

function get : string;
var tmp : string
begin
    if (count = 0) then
        wait(nonempty);
    out = out mod size + 1;
    tmp := buffer[out];
    count := count - 1;
    signal(nonfull);
    get := tmp;
end;
end;
```

| Figura 17.5 Monitor Produtor-Consumidor

O bloqueio da seção crítica na versão semáforo é proporcionado automaticamente pelo monitor em cada função ou procedimento. Isso significa que um produtor tem que tentar uma operação `put` para poder determinar se há ou não espaço no *buffer*. Nesse caso, após a operação `put`, um produtor deve verificar o valor da variável `count` que controla o número de entradas de *buffer* que estão em uso. Se o *buffer* estiver cheio, então o produtor espera que ocorra a condição `nonfull`. Assim, o semáforo de contagem geral anterior foi transformado em uma variável inteira e uma condição. As mudanças na função `get` são similares. Repare que quando uma *thread* é forçada a esperar por uma condição, é liberado o bloqueio sobre o monitor.

O código para as *threads* do produtor e do consumidor não estão na Figura 17.5; a figura mostra apenas as operações que estão associadas com o *buffer* compartilhado.

Monitores e semáforos são equivalentes em poder, pois qualquer monitor pode ser implementado usando semáforos (Hoare, 1974) e vice-versa (Ben-Ari, 1994).

17.3 SINCRONIZAÇÃO EM JAVA

Nesta seção, exploramos uma implementação moderna do controle da concorrência, usando Java como linguagem para ilustrar. Outras linguagens, como Ada, proporcionam recursos similares que serão apresentados resumidamente mais adiante.

17.3.1 *Threads* em Java

Lembre-se, da Figura 17.1, que uma *thread* pode estar em um dentre cinco estados: criada, executável, executando, bloqueada ou terminada. Nesta seção e na próxima, discutimos como uma *thread* faz transições de um estado para outro, basicamente ignorando a transição do estado executável para executando, já que isso é manipulado pela máquina virtual Java subjacente. A Figura 17.6 resume os estados de uma *thread* Java e as transições entre elas.

Em Java, toda *thread* é uma implementação da interface `Runnable`, e assim ela deve conter um método `run()`. A maneira mais simples de criar uma *thread* é definir uma subclasse que herda da classe Java `Thread`:

```
public class MyThread extends Thread {
    public MyThread( ) { ... }
    ...
}
```

Um objeto dessa subclasse pode então ser criado declarando-o e construindo-o por meio de uma operação `new`:

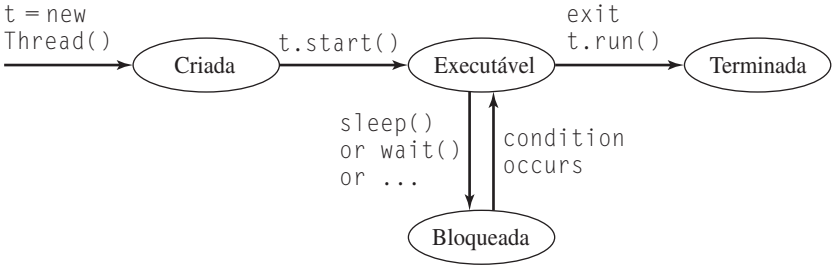
```
Thread thread = new MyThread( );
```

Para tornar essa *thread* executável, seu método `start` deve ser invocado:

```
thread.start( );
```

Iniciar uma *thread* significa transferir controle para seu método `run` e então continuar a execução. Cada classe que estende a classe `Thread` deve então providenciar um método `run`. No entanto, ela não precisa fornecer um método `start`, pois já foi providenciado um pela classe `Thread`.

Um método `run` de uma *thread* normalmente contém um laço, já que saindo do método `run` termina-se a *thread*. Por exemplo, em uma animação gráfica, o método `run`



| Figura 17.6 Estados de uma *Thread* Java

move repetidamente os objetos gráficos, redesenha a tela e então entra no estado *sleep* (para retardar a animação). Assim, uma animação gráfica típica seria:

```
public void run () {
    while (true) {
        moveObjects( );
        repaint( );
        try { Thread.sleep(50);
        } catch (InterruptedException exc) { return; }
    }
}
```

Observe que o método `sleep` potencialmente gera uma `InterruptedException`, que deve ser capturada pelo uso de uma instrução `try-catch`.

A chamada do método `sleep` muda a *thread* do estado executando para o estado bloqueada, no qual a *thread* fica esperando por um intervalo de tempo. *Sleeping* (dormir) freqüentemente é feito em aplicações visuais para evitar que a visualização se realize muito rapidamente. Outras formas de bloqueio incluem acesso a variáveis compartilhadas, que será discutido a seguir.

Finalmente, o estado terminado é alcançado quando o método de execução termina. Uma maneira de alcançar esse estado é a *thread* sair de seu método `run`, que terminará a *thread*, por exemplo:[4]

```
public void run () {
    while (continue) {
        moveObjects( );
        repaint( );
        try { Thread.sleep(50);
        } catch (InterruptedException exc) { return; }
    }
}
```

Para parar essa *thread*, outra *thread* pode chamar um método para definir o valor da variável de instância `continue` para *false*. Essa variável não é considerada como compartilhada; qualquer condição potencial de corrida pode ser seguramente ignorada, pois, na pior hipótese, ela apenas causa uma iteração extra do laço.

Às vezes é inconveniente tornar subclasse a classe `Thread`; por exemplo, podemos querer que um *applet* seja uma *thread* separada. Nesses casos, uma classe meramente tem de implementar a interface `Runnable` diretamente; ou seja, implementar um método `run`. O esboço de uma classe dessas é:

```
public class MyClass extends SomeClass implements Runnable {
   ...
   public void run( ) { ... }
}
```

4. Antes de Java 1.2, uma *thread* podia também ser terminada chamando seu método `stop`; no entanto, por razões complicadas, isso se tornou problemático, e o método `stop` na classe `Thread` foi desaprovado na versão 1.2 de Java.

Cria-se uma instância de `MyClass` em uma *thread* usando:

```
MyClass obj = new MyClass();
Thread thread = new Thread(obj);
thread.start( );
```

Observe que aqui o uso de interfaces em Java elimina a necessidade de herança múltipla.

A sincronização é obtida em Java por meio do conceito de monitor e associando um bloqueio com cada objeto que pode ser compartilhado. Para implementar uma variável compartilhada, criamos uma classe contendo a variável compartilhada e representamos cada método (que não o construtor) como `synchronized`:

```
public class SharedVariable ... {
    public SharedVariable (...) { ... }

    public synchronized ... method1 (...) { ... }
    public synchronized ... method2 (...) { ... }
    ...
}
```

Para realmente compartilhar uma variável, criamos uma instância da classe e a tornamos acessível às *threads* separadas. Uma maneira de conseguir isso é passar o objeto compartilhado como um parâmetro para o construtor de cada *thread*. No caso de uma variável compartilhada produtor-consumidor, pode ser assim:

```
SharedVariable shared = new SharedVariable( );
Thread producer = new Producer(shared);
Thread consumer = new Consumer(shared);
```

Aqui, assumimos que tanto `Producer` quanto `Consumer` estendem `Thread`. O exemplo completo do *bounded buffer* (*buffer* delimitado) aparece na próxima seção.

17.3.2 Exemplos

Nesta seção, são desenvolvidos e ilustrados vários exemplos de concorrência em Java. O primeiro exemplo, chamado "Bouncing Balls" (Bolas que saltam), usa *threads* para rodar uma simples animação, mas sem sincronização. O segundo exemplo reimplementa o problema do *bounded buffer* em Java, destacando o uso de primitivas de sincronização para controlar o acesso a variáveis compartilhadas e evitar o *deadlock*. O terceiro exemplo, chamado "Crivo de Eratóstenes", ilustra o uso da concorrência na computação científica.

Bouncing Balls Neste exemplo consideramos o caso simples quando se usa uma *thread* para rodar uma animação. O problema é representar uma ou mais bolas saltando em uma janela. Quando uma bola alcança a borda da janela, ela inverte a direção de seu movimento; no entanto, não é feita nenhuma tentativa de fazer a coisa parecer fisicamente realística.

Inicialmente, criamos duas classes, uma das quais encapsula a bola e a outra o próprio aplicativo. Vamos primeiro considerar a classe `Ball`.

Quais as informações que se deve ter sobre uma bola em movimento? Se a bola for pintada na tela como um círculo, então deveremos saber:

- Sua localização, na forma de um par de coordenadas (*x*, *y*) dentro da janela ou do quadro de animação.
- Sua direção e velocidade (*delta*), na forma de uma alteração de suas coordenadas (*x*, *y*) como (*dx*, *dy*).
- Seu tamanho ou seu diâmetro em pixels.
- Sua cor (para tornar a animação mais divertida).

Além do construtor, dois outros métodos são fornecidos:

- Um método `move`, que move a bola um passo (*delta*). Se a bola colidir com a borda da janela ou do quadro, sua direção será invertida.
- Um método `paint`, que desenha a bola na tela.

Ao construtor é passada somente a posição inicial da bola na tela. A velocidade, a direção do movimento e a cor são escolhidas como simples funções pseudo-aleatórias dessas coordenadas. O diâmetro é fixo; todas as bolas têm o mesmo tamanho. O código completo para a classe *ball* (bola) é dado na Figura 17.7.

A criação de uma versão inicial da classe de aplicação, chamada `BouncingBalls`, é igualmente simples. Recorde-se do Capítulo 16, em que uma classe de aplicação gráfica deve estender a classe `JPanel` e deve ter um método `main` similar àquele dado na Figura 16.16. Nessa versão inicial são fornecidos dois outros métodos:

- O construtor, que faz algumas arrumações e define a largura e a altura do quadro e coloca uma única bola no quadro.
- Um método `paintChildren`, que solicita à bola que pinte a si própria, de uma maneira tipicamente orientada a objetos.

Essa versão simples e inanimada é mostrada na Figura 17.8. Aqui, a largura e a altura do painel no qual as bolas estão saltando são declaradas como constantes públicas.

Restam dois problemas: acrescentar movimento e acrescentar mais bolas. Há muitas maneiras de se acrescentar mais bolas à tela. Uma delas é gerar aleatoriamente bolas em coordenadas aleatórias. Uma alternativa igualmente simples e mais divertida é inserir uma bola usando um clique de mouse. Por mais que sejam acrescentadas mais bolas, cada bola tem de ser registrada, e o método `paintChildren`, revisado, para pedir a cada bola que se desenhe. Uma solução é inserir cada bola em um `Vector`.

A maneira-padrão de acrescentar movimento a gráficos em computador é usar uma *thread*. Uma abordagem é tornar cada bola uma *thread* separada, cujo método `run` move e desenha a bola. No entanto, para tornar o movimento visível na tela, a bola deve chamar o método *repaint* (redesenhar) do aplicativo. Depois da reflexão, parece exagerado querer um redesenho de todo o aplicativo após o movimento de cada bola.

Uma abordagem melhor é ter uma classe *thread* cujo método `run` move cada bola um passo e só então redesenha a tela. Essa classe é uma classe interna ao próprio aplicativo. Para tornar mais visível o movimento das bolas em um processador rápido, a *thread* entra no estado *sleep* (dormir) após cada redesenho.

O conjunto expandido de variáveis de instância e o construtor associado para esse *applet* revisado das bolas que saltam estão na Figura 17.9.

Como as bolas estão agora todas em um vetor, o método `paintChildren` (mostrado na Figura 17.10) deve ser revisado. Em vez de meramente pintar uma única bola, nós agora fazemos uma iteração sobre todos os objetos no vetor `list`, pedindo a cada bola que se redesenhe.

```
import java.awt.*;

public class Ball {

    Color color = Color.red;
    int x;
    int y;
    int diameter = 10;
    int dx = 3;
    int dy = 6;

    public Ball (int ix, int iy) {
       super( );
       x = ix;
       y = iy;
       color = new Color(x % 256, y % 256, (x+y) % 256);
       dx = x % 10 + 1;
       dy = y % 10 + 1;
    }

    public void move () {
       if (x < 0 || x >= BouncingBalls.width)
          dx = - dx;
       if (y < 0 || y >= BouncingBalls.height)
          dy = - dy;
       x += dx;
       y += dy;
    }

    public void paint (Graphics g) {
       g.setColor(color);
       g.fillOval(x, y, diameter, diameter);
    }
} // Ball
```

| Figura 17.7 Classe Ball (Bola)

A classe do aplicativo contém também uma classe interna `MouseHandler` (mostrada na Figura 17.11), cujo método `mousePressed` é chamado quando é detectado um clique do mouse. Esse método cria uma nova bola nas coordenadas (x, y) do clique e insere a bola no vetor.

Finalmente, a classe `BallThread` é mostrada na Figura 17.12. Essa classe é formada por apenas um método de execução, que executa um laço continuamente. Em cada iteração do laço, cada bola é movida, todo o quadro é redesenhado, e então a *thread* entra no estado *sleep* (para retardar a animação).

```
import java.awt.*;

public class BouncingBalls extends JPanel {
    public final static int width = 500;
    public final static int height = 400;
    private Ball ball = new Ball(128, 127);

    public BouncingBalls ( ) {
       setPreferredSize(new Dimension(width, height));
    }

    public synchronized void paintChildren(Graphics g) {
       ball.paint(g);
    }

    public static void main(String[] args) {
       JFrame frame = new JFrame("Bouncing Balls");
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       frame.getContentPane().add(new BouncingBalls( ));
       frame.setLocation(50, 50);
       frame.pack();
       frame.setVisible(true);
    }
}
```

| Figura 17.8 Classe Inicial do Aplicativo

```
public final static int width = 500;
public final static int height = 400;
private Ball ball = new Ball(128, 127);
private Vector<Ball> list = new Vector();

public BouncingBalls ( ) {
    setPreferredSize(new Dimension(width, height));
    list.add(ball);
    addMouseListener(new MouseHandler());
    BallThread bt = new BallThread();
    bt.start( );
}
```

| Figura 17.9 Construtor Final das Bolas que Saltam

```
public synchronized void paintChildren(Graphics g) {
    for (Ball b : list) {
        b.paint(g);
    }
}
```

| Figura 17.10 Método Final paintChildren das Bolas que Saltam

```
private class MouseHandler extends MouseAdapter {
  public void mousePressed(MouseEvent e) {
      Ball b = new Ball(e.getX(), e.getY());
      list.add(b);
  } // mousePressed
} // MouseHandler
```

| Figura 17.11 Manipulador do Mouse das Bolas que Saltam

```
private class BallThread extends Thread {
    public boolean cont;
    public void run( ) {
        cont = true;
        while (cont) {
        for (Ball b : list) {
            b.move();
        }
        repaint( );
        try { Thread.sleep(50);
        } catch (InterruptedException exc) { }
      }
    } // run
} // BallThread
```

| Figura 17.12 *Thread* das Bolas que Saltam

Um leitor atencioso, que leu as seções anteriores deste capítulo, deve estar preocupado com possíveis condições de corrida. É improvável ocorrer uma condição de corrida nesse aplicativo simples, mas pode ocorrer, dependendo da implementação da classe `Vector`. Teoricamente, é possível que a implementação do método `add` da classe Vector crie um elemento vazio no vetor e então insira as informações apropriadas no elemento. Uma verificação da Java API para a classe `Vector` mostra que o método `add` é sincronizado, e bloqueia, portanto, o acesso do objeto por outra *thread* (por exemplo, a *thread* que chama `paintChildren`).

Vários comandos *import* adicionais devem agora ser acrescentados ao cabeçalho desta classe:

```
import javax.swing.*;
import java.awt.event.*;
import java.util.*;
import java.awt.*;
```

Fica como exercício uma variedade de possíveis modificações para essa classe.

Bounded Buffer Nesta seção, examinamos novamente o exemplo do *bounded buffer*, desta vez em Java. Lembre-se de que ele admite pelo menos duas *threads*, um produtor e um consumidor. No entanto, nossa solução acomodará múltiplos produtores e múltiplos consumidores.

Recorde-se da Seção 17.2, em que o produtor e o consumidor se comunicam por meio de um *buffer* compartilhado de tamanho finito. O *buffer* efetivamente é colocado dentro de um monitor declarando cada um de seus métodos públicos como `synchronized`. Nesse caso, Java cria um bloqueio associado com cada instância ou objeto da classe *`Buffer`*; somente uma única *thread* pode executar qualquer método sincronizado para determinado objeto.

Tanto o produtor quanto o consumidor acessam o *buffer* de maneira circular. O produtor deposita informações (neste exemplo, *strings* de datas) usando um método `put`, enquanto o consumidor remove informações por meio de um método `get`.

Na versão monitor, após iniciar a execução do método, o produtor pode ter que esperar (renunciar ao monitor) na presença da condição `nonfull`, enquanto o consumidor pode ter que esperar na presença da condição `nonempty`. Em ambos os casos, após depositar/receber informações, o produtor/consumidor tem que sinalizar o outro. Java oferece suporte tanto para a espera (método `wait()`) quanto para a sinalização (método `notify()`), mas não em uma condição específica. Uma chamada a `notify()` sinaliza uma *thread* em espera, que pode ser um produtor ou um consumidor. Assim, as simples instruções `if` da solução do monitor se tornam laços `while` em Java. Caso contrário, o código Java para a classe *`Buffer`* dado na Figura 17.13 imita a solução do monitor razoavelmente bem.

O produtor e o consumidor são muito similares na construção. Ambas as classes são subclasses de `Thread` e tomam três argumentos: uma instância do *buffer* compartilhado, um inteiro informando à *thread* por quanto tempo ela deve permanecer no estado *sleep* (dormir) e um limite de iteração. Cada classe consiste em apenas um construtor e um método `run` que são dados na Figura 17.14 e na Figura 17.15, respectivamente.

O aplicativo principal (veja a Figura 17.16) é muito simples. Primeiro, ele constrói um *buffer*, depois um produtor e um consumidor. O *buffer* é passado como um argumento para as duas *threads*, junto com o tempo em milissegundos e uma contagem de iteração. Nesse exemplo, o produtor está configurado para produzir uma marcação de tempo a cada segundo, mas o consumidor consome a mensagem somente a cada três segundos. Devido ao fato de o *buffer* ter um tamanho finito, eventualmente o produtor tem sua velocidade diminuída até àquela do consumidor.

Crivo de Eratóstenes Uma área de importância crescente na programação concorrente/paralela é a computação científica. A idéia é tomar os algoritmos numéricos e reformulá-los de maneira que partes do problema possam ser resolvidas em paralelo. O desenvolvimento de conjuntos poderosos de estações de trabalho de baixo custo tornou muito atraente essa abordagem para resolver problemas complexos como a análise estrutural dos aviões, dinâmica dos fluidos, modelagem das condições meteorológicas, entre outros.

```
class Buffer {
    private String[] buffer;
    private int in = -1;
    private int out = -1;
    private int count = 0;

    public Buffer(int size) { buffer = new String[size]; }

    public synchronized void put(String s) {
        while (count >= buffer.length)
        try{ wait(); } catch(InterruptedException e)
        { return; };
        count++;
        buffer[++in % buffer.length] = s;
        notifyAll();
    }

    public synchronized String get() {
        while (count == 0)
           try { wait(); }
             catch(InterruptedException e) { return; };
        count--;
        String s = buffer[++out % buffer.length];
        notifyAll();
        return s;
    }
} // Buffer
```

| Figura 17.13 Classe *Buffer*

Neste exemplo, calculamos todos os números primos menores do que um determinado número usando o *Crivo de Eratóstenes*. Usando conjuntos, o algoritmo abstrato seqüencial funciona da seguinte maneira:

```
Set s = {2..N}; // inicializa s
while (|s| > 0) { // s não está vazia
   int p = x in s      // x é o valor mínimo em s
   print(p);           // p é primo
   for (int i = p; i <= N; i += p) // cada múltiplo de p
     s = s - {i}; // exclui de s
}
```

Inicializamos uma série `s` para todos os inteiros de 2 até `N` (nosso valor de entrada). Depois, enquanto a série `s` não está vazia, selecionamos primeiro o valor mínimo de `s` e o chamamos de `p`. O valor `p` deve ser primo, assim ele é um dos números do nosso resultado. Então, removemos da série `s` todos os múltiplos de `p`, incluindo o próprio `p`.

Na conversão desse algoritmo seqüencial para um outro concorrente, notamos em primeiro lugar que a identificação (filtragem) de múltiplos de um dado número primo de uma série

```
import java.util.Date;

public class Producer extends Thread {
    private Buffer buffer;
    private int millisecs;
    private int iterations;

    public Producer(Buffer b, int s, int n) {
        buffer = b;
    millisecs = s; iterations = n;

    }

    public void run() {
        try {
            for (int i = 0; i<iterations; i++) {
                buffer.put(new Date().toString());
                Thread.sleep(millisecs);
            }
        } catch(InterruptedException e) { };
    }
} // Produtor
```

| Figura 17.14 Classe *Producer* (Produtor)

```
public class Consumer extends Thread {
    private Buffer buffer;
    private int millisecs;
    private int iterations;

    public Consumer(Buffer b, int s, int n) {
        buffer = b;
        millisecs = s; iterations = n;
    }

    public void run() {
        try {
           for (int i = 0; i<iterations; i++) {
                System.out.println(buffer.get());
                Thread.sleep(millisecs);
           }
        } catch(InterruptedException e) { };
    }
} // Consumidor
```

| Figura 17.15 Classe *Consumer* (Consumidor)

```
public class BoundedBuffer {
     public static void main(String[] arg) {
        Buffer buffer = new Buffer(5);
        Producer producer = new Producer(buffer, 1000, 20);
        producer.start();
        Consumer consumer = new Consumer(buffer, 3000, 20);
        consumer.start();
     }
} // BoundedBuffer
```

| Figura 17.16 Classe *Bounded Buffer*

```
class Sieve implements Runnable {
    Buffer in;

    public Sieve(Buffer b) { in = b; }

    public void run( ) {
       int p = in.get( );
       if (p < 0) return;
       System.out.println(p);
       Buffer out = new Buffer(5);
       Thread t = new Thread(new Sieve(out));
       t.start();
       while (true) {
          int num = in.get();
          if (num < 0) {
              out.put(num);
              return;
          }
          if (num % p != 0)
              out.put(num);
       }
    }
}
```

| Figura 17.17 Crivo de Eratóstenes

pode ser feita concorrentemente. Em segundo lugar, notamos que a própria série pode ser implícita ao invés de explícita. Isto é, cada filtro pode passar quaisquer números que não foram eliminados da série, por intermédio de um *buffer*, para o próximo filtro. Contanto que os números sejam mantidos em ordem numérica, o primeiro número enviado a cada *thread* concorrente deve ter passado por todos os filtros anteriores e deve ser primo. Nossa solução para o Crivo de Eratóstenes na forma de um programa concorrente aparece na Figura 17.17.[5]

5. Dershem e Jipping (1990, p. 210) apresentam uma solução para esse problema em Ada.

Nesta solução, o fim da seqüência de números é marcado com um número negativo, digamos, –1. Toda a comunicação ocorre por meio de um objeto `Buffer` que adaptamos da Figura 17.13 para acomodar inteiros em lugar de *strings*. O primeiro número recebido por intermédio do *buffer* de entrada é verificado para saber se ele é menor do que 0; se for, a seqüência está vazia e o processo de filtragem deve parar.

Caso contrário, o número é guardado na variável `p`; conforme observamos anteriormente, `p` deve ser primo, e então ele é impresso. É criado um *buffer* de saída e passado para um novo `Sieve` (crivo), iniciado como uma *thread*; daqui por diante, toda a comunicação com a *thread* iniciada é por meio do *buffer* de saída.

Então, repetidamente, obtemos outro número do *buffer* de entrada. Se o número for menor do que 0, então ele marca o fim da seqüência. Neste caso, o marcador do fim é copiado para o *buffer* de saída, e a *thread* se encerra saindo de seu método `run`. Caso contrário, o número é verificado para saber se ele é um múltiplo de `p`; se ele não for um múltiplo, então o número é passado por intermédio do crivo via *buffer* de saída.

Feito isso, só falta criar um programa para controlar a execução da aplicação (veja a Figura 17.18). Nesse programa, o número de entrada `N` é obtido da linha de comando. Então é criado um *buffer* e passado para um filtro do crivo como uma *thread*. Em seguida, todos os números de 2 até `N` são passados para o filtro do crivo por meio do *buffer*. Finalmente, o fim da seqüência de números é indicado passando-se um negativo por intermédio do *buffer*.

Como ocorreu com nossos programas concorrentes anteriores, esse assume uma memória compartilhada por meio da qual as *threads* se comunicam. Sem múltiplos processadores, esse programa rodaria mais devagar do que na versão seqüencial. Com um pouco de trabalho, esse programa poderia ser convertido em um programa distribuído. Nesse caso, o *buffer* teria de ser substituído por um *network socket*, e o código que inicia uma nova *thread* seria substituído pelo código que obtém um endereço de rede de um

```
public static void main (String[] arg) {
    if(arg.length < 1) {
      System.out.println("Usage: java Sieve number");
      System.exit(1);
    }
    try {
      int N = Integer.parseInt(arg[0]);
      Buffer out = new Buffer(5);
      Thread t = new Thread(new Sieve(out));
      t.start();
      for (int i = 2; i <= N; i++)
        out.put(i);
      out.put(-1);
    } (catch NumberFormatException e) {
      System.out.println("Illegal number: " + arg[0]);
      System.exit(1);
    }
}
```

| Figura 17.18 Programa Principal (Test Driver) para o Crivo de Eratóstenes

processo em espera. No caso de um programa distribuído, há também o problema de reorganizar-se a saída de maneira coerente. No entanto, a estratégia básica do programa permaneceria intacta.

17.4 COMUNICAÇÃO INTERPROCESSOS

Muitas linguagens suportam a implementação de vários modelos para realizar a comunicação interprocessos.

> **Definição**: *Comunicação interprocessos (IPC)* ocorre quando dois ou mais programas diferentes se comunicam simultaneamente um com o outro por meio de uma rede.

A discussão a seguir ilustra um modelo de IPC conhecido comumente como modelo *cliente-servidor*.

> **Definição**: Uma *arquitetura cliente-servidor* é uma arquitetura de rede na qual cada computador ou processo na rede é um cliente ou um servidor.

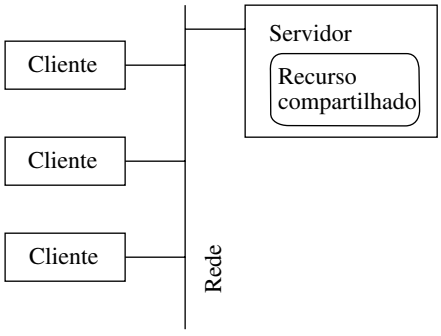
Servidores são computadores poderosos ou processos dedicados ao gerenciamento de bases de dados (servidores de arquivo), impressoras (servidores de impressão) ou de tráfego de rede (servidores de rede). Clientes são PCs ou estações de trabalho (*workstations*) nas quais os usuários rodam aplicativos. Os clientes dependem dos servidores para compartilhar recursos como arquivos, dispositivos e até mesmo poder de processamento.[6]

Assim, aplicações cliente–servidor refletem as características das arquiteturas cliente–servidor. Elas têm um programa, usualmente residindo em um único computador, que serve recursos através da rede para vários clientes simultaneamente. Cada cliente tipicamente reside em um computador diferente conectado a um servidor através de uma rede. Os clientes ficam então acessando simultaneamente os recursos fornecidos pelo servidor. Isso é visualizado na Figura 17.19.

A concorrência ocorre em uma aplicação cliente-servidor porque programas-clientes conectam, desconectam e acessam recursos compartilhados do servidor em uma ordem imprevisível. Assim, em determinado momento, qualquer número de clientes pode estar acessando o mesmo serviço simultaneamente. A tarefa do servidor é coordenar esses acessos de forma que:

1. Seja evitado o *deadlock* e a inanição;
2. Sejam evitadas as corridas críticas.

| Figura 17.19 Arquitetura Cliente-Servidor



6. A arquitetura cliente-servidor muitas vezes é contrastada com a arquitetura "peer-to-peer" (ponto a ponto), na qual cada processador pode agir tanto como um cliente quanto como um servidor. Cada uma dessas arquiteturas tem seus pontos fracos e fortes, bem como uma ampla gama de aplicações.

Deadlock e inanição são evitados quando o servidor trata todos os clientes com a mesma imparcialidade. As corridas críticas são evitadas quando o servidor desautoriza dois clientes de acessar um recurso compartilhado (por exemplo, a mesma entrada de um banco de dados) ao mesmo tempo.

Para controlar esse ambiente, o servidor inicia uma nova *thread* de controle todas as vezes em que um novo cliente pede para conectar-se e acessar os seus serviços. O servidor deve seguir convenções básicas de rede para garantir que essas conexões sejam estabelecidas com integridade. A discussão a seguir explora essas atividades com mais detalhes.

17.4.1 Endereços IP, Portas e *Sockets*

Cada computador na Internet tem um `id` (identificador de processador) único, que é um *endereço IP* (por exemplo, `139.140.1.1`) ou um *nome de domínio* (por exemplo, `bowdoin.edu`). Tem também um conjunto daquilo que chamamos de `portas`, que são pontos de acesso por meio dos quais os dados podem ser enviados ou recebidos na Internet. Cada uma dessas portas é identificada por um número único no intervalo de 0–65.535.

> **Definição**: Um *socket* é um ponto extremo de uma conexão dedicada de comunicação entre dois programas sendo executados em uma rede.

Por exemplo, a classe Java `Socket` é usada para representar um ponto final de rede.

São necessárias duas peças de informação para definir corretamente um *socket* para um programa – o `id` de seu processador e a `porta` por meio da qual o *socket* se conecta com a rede. Cada *socket* permanece dedicado a sua porta atribuída durante toda a existência da conexão.

17.4.2 Um Exemplo de Cliente-Servidor

Para ilustrar os elementos de um sistema cliente-servidor, vamos considerar uma aplicação simples que conta os votos recebidos por uma urna de votação eletrônica. Usamos Java como linguagem de ilustração.

Cada votante é um cliente e pode conectar-se à urna eletrônica por meio de qualquer computador na Internet. Uma interação típica entre a urna eletrônica e o votante pode ser a seguinte:

```
> Vote em um ou mais: 1. Allen 2. Bob 3. Alan 4. Rebecca
3
> Você quer votar novamente? (s/n)
s
> Digite outro número:
2
> Você quer votar novamente? (s/n)
n
> Bye
```

Enquanto o cliente não fornecer o valor para `n`, as opções continuam aparecendo. Naturalmente, o servidor deve garantir que nenhum cliente possa votar na mesma pessoa duas vezes, e cada cliente deve depositar pelo menos um voto válido. Uma votação está completa quando o cliente digita um valor para `n`.

No lado do servidor, o término de cada sessão é ecoado, e é fornecido o resultado da votação quando cada votante finaliza sua sessão. Veja aqui o exemplo de um resultado que pode aparecer no lado do servidor depois que quatro votantes completaram sessões como aquela acima (alguns dos detalhes foram omitidos para economizar espaço).

```
Server listening on port 9600
Voter starting
Voter finishing
Current voting tally:
     candidate 1. 0
     candidate 2. 1
     candidate 3. 1
     candidate 4. 0
Voter starting
Voter finishing
...
Voter starting
Voter starting
Voter finishing
...
Voter finishing
Current voting tally:
     candidate 1. 3
     candidate 2. 2
     candidate 3. 2
     candidate 4. 1
```

Embora quaisquer sessões dessas possam se sobrepor no tempo, após elas serem todas finalizadas, a `Current voting tally` (contagem atual dos votos) deve refletir a opinião coletiva. Particularmente, não deve ser desprezado nenhum voto de qualquer votante e também nenhum voto pode ser contado duas vezes, como poderia ocorrer se fosse permitida uma corrida crítica.

Nesta discussão, queremos focalizar os aspectos da concorrência das aplicações cliente-servidor. Portanto, omitiremos muitos dos detalhes de programação para poder discutir problemas de *deadlock*, inanição e corridas críticas. Os leitores interessados em rodar o programa completo poderão fazer o download do site do livro.[7]

O Lado do Servidor Quando o servidor é iniciado, sua tarefa é inicializar a contagem dos votos, abrir um *socket* e permanecer ativamente atendendo a novos clientes que tentam se conectar àquele *socket*. Cada solicitação é satisfeita ao ser iniciada uma nova *thread* de controle.

Uma *thread* sempre tem um método `run`, conforme já discutimos em seções anteriores. A função de cada *thread* é supervisionar uma interação com um votante em particular e compartilhar o acesso ao recurso `tally`. Na Figura 17.20 é mostrado um exemplo dessa dinâmica, quando três votantes ativos estão acessando o servidor simultaneamente.

A estratégia do servidor para iniciar as *threads* é resumida a seguir:

1. É inicializada uma variável compartilhada para totalizar os votos.
2. Abre-se um novo *socket* para se comunicar com clientes.
3. Entra em um laço que:
   (a) Atende às solicitações de conexão de clientes por meio daquele *socket*.
   (b) Sempre que chega uma solicitação, aceita-a e inicia uma nova *thread*.

7. Este aplicativo é adaptado de um outro similar em *The Java Tutorial* no site http://java.sun.com/docs/books/tutorial/networking/sockets/clientServer.html.

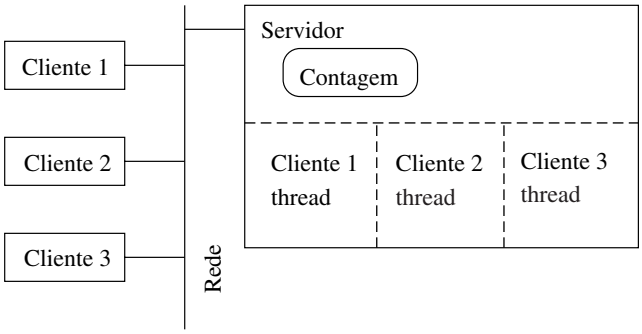


Figura 17.20 Arquitetura Cliente-Servidor para Contagem de Votos

Esse laço continua até que o servidor seja desligado. O código essencial do servidor está na Figura 17.21. Os comentários nesse código identificam as etapas acima.

A variável compartilhada `tally` se comporta como o *bounded buffer* discutido nas seções anteriores. Ela não pode ser acessada por duas *threads* ao mesmo tempo, pois cada *thread* pode potencialmente atualizá-la. Assim, cada atualização da `tally` (contagem) deve ocorrer dentro da seção crítica de uma *thread*. Por essa razão, `tally` é encapsulada dentro do objeto `TallyBuffer`, cujos métodos são `synchronized`. Essa classe está esboçada na Figura 17.22.

Deve ser observado que os métodos nessa classe são `synchronized`, o que garante que não há duas chamadas a qualquer um que possam se sobrepor no tempo. Essa restrição

```
public class Server
    ...
// 1. Inicializa a variável compartilhada para contar os votos.
    TallyBuffer tally = new TallyBuffer();
    ...
// 2. Abre um novo Socket para comunicar-se com os clientes.
        try {
           serverSocket = new ServerSocket(port);
           System.out.println(
           "Server listening on port " + port);
        } catch (IOException e) {
           System.err.println(
           "Port: " + port + " unavailable.");
           System.exit(-1);
        }
// 3. Recebe as solicitações de conexão dos clientes
//     e inicia novas threads.
         while (true) {
            ct = new ClientThread(
               serverSocket.accept(), tally);
            ct.start();
         }
...
```

| Figura 17.21 Servidor para uma Aplicação Cliente-Servidor Simples

```
public class TallyBuffer {
     int[] tally = {0,0,0,0};

     public synchronized void update(int[] aBallot) {
       for (int i=0; i<tally.length; i++)
            tally[i] = tally[i] + aBallot[i];
     }
     public synchronized void display() {
     ...
     }
}
```

| Figura 17.22 Sincronizando o Acesso à Variável Compartilhada `tally`

impede que aconteçam corridas críticas, pois a variável `tally` dentro dessa classe só pode ser atualizada por uma chamada de cada vez.

Cada *thread* cliente deve se comunicar por meio do *socket* com seu votante associado. Ela deve ser capaz de atualizar a variável compartilhada `tally` que foi declarada e inicializada pelo servidor. São necessários os seguintes passos gerais para que uma *thread* cliente possa rodar corretamente:

1. Abrir novos fluxos (*streams*) de entrada e saída com o cliente.
2. Inicializar uma nova votação e um protocolo para interagir com o votante.
3. Entrar em um laço que:
   (a) lê um voto do cliente,
   (b) processa aquele voto, atualizando a votação,
   (c) fica pronto para receber outro voto.
4. Fechar os dois fluxos e, finalmente, o *socket*.

Na Figura 17.23 são mostrados mais detalhes de uma *thread* cliente. Os comentários identificam o código necessário para implementar os três primeiros passos.

O Protocolo Cliente O Protocolo Cliente `voter` referenciado nesse programa é um objeto que controla as interações entre uma única *thread* cliente e o usuário que está votando. Isto é, ele cuida dos detalhes, como garantir que um votante deposite pelo menos um voto válido, que não vote duas vezes no mesmo candidato, e assim por diante.

A lógica é a de que o protocolo deve fazer uma única transição entre estados da interação de um votante cada vez que seu método `processInput` for chamado. Os estados da interação do votante são diretos. O estado inicial é `WAITING`, enquanto os estados intermediários são `SENTBALLOT` (o votante enviou o voto) e `WANTANOTHER` (o votante tem oportunidade de votar em um segundo, terceiro ou quarto candidato). O estado final é `DONE`, que encerra a interação do votante e dispara a atualização da variável compartilhada `tally` com o voto desse votante.

Parte do código do protocolo é mostrada na Figura 17.24, que identifica as transições potenciais entre estados para um votante. Observe também que, quando um votante finalmente completa o envio de seu voto, o voto é aceito chamando o método `update` de

```
public class ClientThread extends Thread
...
//1. Abre novos fluxos de entrada e saída com o cliente.
    PrintWriter out = new PrintWriter(
        socket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(socket.getInputStream()));

// 2. Inicializa uma nova votação e protocolo.
    String inputLine, outputLine;
    ClientProtocol voter = new ClientProtocol();
    outputLine = voter.processInput(null, tally);
    out.println(outputLine);

// 3. Executa laço para ler e processar votos individuais.
    while ((inputLine = in.readLine()) != null) {
        outputLine = voter.processInput(inputLine, tally);
        out.println(outputLine); // transmite ao cliente
        if (outputLine.equals("Bye"))
            break;
    }
...
```

| Figura 17.23 ***Thread*** **Cliente para uma Aplicação Cliente-Servidor Simples**

TallyBuffer. Esse passo simula o registro do voto de um único votante em uma urna de votação eletrônica convencional.

O Lado Cliente O programa cliente é simples porque o servidor está fazendo todo o trabalho.[8] O cliente age como um retransmissor entre a *thread* cliente (no servidor) e o usuário. Ele recebe uma mensagem da *thread*, passa-a para o usuário, recupera a resposta do usuário e a envia para a *thread*. Uma mensagem especial, `Bye`, é enviada como sinal para terminar a sessão cliente. Aqui estão as etapas básicas:

1. Abrir um *socket*.
2. Abrir fluxos de entrada e saída para o *socket* e o usuário.
3. Entrar em um laço que:
   (a) lê uma mensagem do fluxo de entrada,
   (b) mostra-a para o usuário,
   (c) recupera uma resposta do usuário,
   (d) escreve aquela resposta no fluxo de saída.
4. Fechar os fluxos de entrada e saída e, finalmente, o *socket*.

8. Um modelo como esse às vezes é chamado de sistema "thin client". O caso oposto é quando a maior parte do trabalho é feita no lado do cliente, caso em que é chamado de sistema "fat client".

```
public class ClientProtocol
...
private int state = WAITING;
int[] ballot = {0,0,0,0};

public String processInput(String response,
                 TallyBuffer tally) {
    ...
    if (state == WAITING) {
        theOutput = "Vote em mais um: " +
              "1. Allen 2. Bob 3. Alan 4. Rebecca" ;
        state = SENTBALLOT;
    }
    else if (state == SENTBALLOT)
        if ...
              state = WANTANOTHER;
    ...
    else if (state == WANTANOTHER){
        if (response.equalsIgnoreCase("y")) {
              theOutput = "Digite outro número: ";
              state = SENTBALLOT;
        }
        else {
              state = DONE;
              tally.update(ballot); // recebe o voto
              theOutput = "Bye";
        }
    ...
    return theOutput;
...
```

| Figura 17.24 Elementos do ClientProtocol para um Único Votante

O código cliente para as primeiras três etapas é dado na Figura 17.25. O programa cliente é um aplicativo Java cujo argumento de linha de comando deve fornecer o nome do servidor (host name) e a porta onde o servidor está executando. Esse programa não pode começar a executar enquanto o servidor não tiver sido ativado na máquina hospedeira (host).

Aqui, `in` e `out` representam mensagens de entrada e saída vindo do *socket* do servidor para esse cliente. O cliente simplesmente ecoa essas mensagens para o votante e passa as respostas do votante de volta para o servidor. A interação no terminal do votante é feita via `stdIn` (que é `System.in` envolvido por um `BufferedReader`) e `System.out`.

A simplicidade desse código ilustra uma vantagem de uma arquitetura *thin client*. Uma desvantagem, naturalmente, é que um grande número de *threads* simultâneas causará estresse no lado servidor. Um tratamento mais cuidadoso do comportamento de diferentes arquiteturas cliente-servidor está além do escopo desta discussão.

```
public class Client
// 1. Abre um socket.
    mySocket = new Socket(hostName, port);

// 2. Abre fluxos de entrada e saída para o socket e usuário.
    out = new PrintWriter(
            mySocket.getOutputStream(), true);
    in = new BufferedReader(new InputStreamReader(
            mySocket.getInputStream()));
    stdIn = new BufferedReader(
          new InputStreamReader(System.in));
    String fromServer;
    String fromUser;
// 3. Laço para transmitir mensagens da thread cliente
//    ao usuário, e do usuário para a thread cliente.
    while ((fromServer = in.readLine()) != null) {
       System.out.println(“>” + fromServer);
       if (fromServer.equals(“Bye”))
          break;
       fromUser = stdIn.readLine();
       if (fromUser != null)
          out.println(fromUser);
...}
```

| Figura 17.25 Classe Cliente para uma Aplicação Cliente–Servidor Simples

17.5 CONCORRÊNCIA EM OUTRAS LINGUAGENS

A concorrência é suportada em muitas outras linguagens de programação além de Java e em todos os outros paradigmas além daquele orientado a objetos. Esta seção resume rapidamente as características principais de linguagem para suporte à concorrência entre as linguagens C#, C++, High-Performance Fortran e Ada. Com isso, o leitor terá uma série de exemplos sobre como a concorrência é implementada para se adequar às necessidades de uma ampla gama de aplicações de programação e preferências além de Java.

C# C# é similar a Java em sua abordagem da concorrência; suas características refletem o conceito de monitor discutido anteriormente. C# tem duas operações, chamadas `Monitor.Enter(o)` e `Monitor.Exit(o)`, que proporcionam entrada e saída explícita do monitor de um objeto e, portanto, uma habilidade direta de bloqueio sobre o objeto. C# também suporta bloqueio de um objeto liberando-o de outras partes no código.

C++ A Biblioteca-Padrão C++ não tem suporte para a concorrência. No entanto, bibliotecas de projeto C++ independentes ampliaram a Biblioteca-Padrão de várias maneiras. Um desses projetos é chamado ACE (ADAPTIVE Communication Environment – Ambiente de Comunicação Adaptativa), e ele fornece suporte C++ para redes, comunicação e concorrência.

ACE tem um conjunto de classes envolventes C++ (C++ *wrapper classes*) que inclui primitivas de sincronização (classe Semáforo), comunicação interprocessos com *sockets* e *pipes* (Classes ACE_SOCK_Connector, ACE_SOCK_FIFO e ACE_SOCK_Addr) e concorrência (Future and Servant thread classes). Ace tem duas classes, denominadas *Reactor* e *Proactor*, que despacham mensagens de uma forma orientada a objetos juntamente a essas classes de sincronização.

Pode-se obter mais informações sobre ACE no site Distributed Object Computing da Universidade de Washington (**http://www.cs.wustl.edu/doc/**).

High-Performance Fortran High-Performance FORTRAN (HPF) tem características que permitem ao programador fornecer informações ao compilador, que o ajudam a otimizar o código em um ambiente de multiprocessamento. Veja um resumo dessas características:

Característica	Especificação HPF
1. Número de processadores	`!HPF$ PROCESSORS procs (n)`
2. Distribuição dos dados	`!HPF$ DISTRIBUTE (type)` `ONTO procs :: identifiers`
3. Alinhar dois conjuntos	`ALIGN array1 WITH array2`
4. Laço concorrente	`FORALL (identifier = begin:end) statement`

Nessa tabela, o parâmetro `type` pode ter o valor `BLOCK`, que significa "distribuir a processadores em blocos", ou `CYCLIC`, que significa "distribuir a processadores em série".

Aqui está um breve exemplo, no qual `A, B, C` e `D` são conjuntos que estão sendo manipulados em um ambiente de 10 processadores. Neste caso, gostaríamos que os conjuntos `A` e `B` fossem distribuídos em blocos entre os 10 processadores.

```
REAL A(1000), B(1000)
!HPF\$ PROCESSORS p (10)
!HPF\$ DISTRIBUTE (BLOCK) ONTO p :: A, B
...
FORALL (i = 1:1000)
    A(i) = B(i)
```

A instrução `FORALL` é usada para especificar um laço cujas iterações podem ser executadas concorrentemente. Esse aqui, em particular, especifica que todas as 1000 atribuições `A(i) = B(i)` podem ser executadas simultaneamente.

Ada Um *rendezvous* é um conceito desenvolvido durante o projeto de Ada. É um conceito simples, que sugere que duas tarefas podem se comunicar somente quando ambas estiverem prontas. Por exemplo, considere um carro aproximando-se de um cruzamento com um sinal de trânsito. O carro pode passar pelo cruzamento somente se ambas as situações a seguir ocorrerem:

1. O sinal está verde,
2. O carro está no cruzamento.

Se qualquer uma dessas condições for falsa, então não há *rendezvous* (isto é, o carro não pode passar pelo cruzamento).

Uma tarefa Ada (T) pode ter vários pontos de entrada alternativos (E) que podem ser selecionados em tempo de execução:

```
task body T is
  loop
    select
      accept E1 (params) do
      ...
      end E1;
    or
      accept E2 (params) do
      ...
      end E2;
      ...
    end select;
  end loop;
end TASK_EXAMPLE;
```

A cada ponto de entrada E está associada uma fila de mensagens esperando por ele. Inicialmente, essas filas estão todas vazias, e o laço espera no topo do comando *select*, até que chegue a primeira mensagem T. Quando chega uma mensagem, o código do ponto de entrada é executado, e o chamador (emissor da mensagem) fica suspenso durante aquele tempo. Podem ser usados parâmetros para passar informações em qualquer das direções – para o ponto de entrada ou para o emissor.

No entanto, durante a execução do código de um ponto de entrada, podem chegar outras mensagens para a tarefa T. Cada uma dessas mensagens é colocada em uma fila associada ao seu ponto de entrada. Quando o código de um ponto de entrada se completa, a execução do código do emissor é retomada, bem como a reavaliação do comando *select* a partir do seu início. Agora pode haver mais de uma fila com uma mensagem ativa esperando por serviço, caso em que uma fila é escolhida aleatoriamente para ser a próxima que recebe os serviços da tarefa T. Assim, ocorre um *rendezvous* sempre que um ponto de entrada E começa a processar uma mensagem que está no início da fila.

Além do *rendezvous*, Ada inclui duas outras características que suportam concorrência: (1) objetos protegidos e (2) comunicação assíncrona. Um objeto protegido proporciona uma maneira mais eficiente do que o *rendezvous* para implementar dados compartilhados. A comunicação assíncrona é proporcionada por meio de um refinamento da estrutura `select`. Nesse último caso, pode ser especificado um dentre dois mecanismos de ativação alternativos, uma cláusula `entry` ou uma cláusula `delay`. A cláusula entry é ativada quando é enviada uma mensagem a E, e a cláusula delay é ativada quando é atingido o tempo-limite especificado para E.

17.6 RESUMO

Conforme sugere este capítulo, a concorrência não é mais domínio exclusivo dos projetistas de sistemas operacionais ou de rede. A programação concorrente já é utilizada em larga escala e é um elemento essencial do repertório de um programador moderno.

As linguagens modernas suportam a programação concorrente de várias maneiras. A sincronização para evitar *deadlocks* e corridas é um elemento-chave da programação concorrente. Os projetistas de linguagens devem continuar a incorporar características de programação concorrente em futuras linguagens de programação.

EXERCÍCIOS

17.1 Considere o exemplo das Bolas que Saltam. Torne-o mais realístico fazendo as seguintes modificações:
(a) Mude o método `move` da bola de maneira que ela reverta a direção assim que tocar a borda da janela.
(b) Torne a *inversão de direção* mais realística fisicamente (por exemplo, acrescente a gravidade).
(c) Faça o *delta* e a cor de uma bola mais aleatórios, usando os bits menos significativos do relógio.
(d) Tente remover o vetor das bolas, transformando cada bola em um `Component`. Você pode agora simplesmente *adicionar* cada bola no quadro? Especule o que está realmente acontecendo com base em outros exemplos.
(e) Modifique o exemplo de maneira que ele possa ser executado como um aplicativo ou como um *applet*.

17.2 No exemplo das Bolas que Saltam, remova a `BallThread` e, em lugar disso, faça o aplicativo implementar a interface `Runnable`. Ajuste o restante do aplicativo conforme for necessário. Qual é melhor?

17.3 Modifique o exemplo das Bolas que Saltam para usar um *array* de tamanho 5 em vez de um `Vector`. Sem quaisquer alterações, você pode introduzir uma condição de corrida? Demonstre.

17.4 Modifique o exemplo das Bolas que Saltam de maneira que ele execute como um aplicativo ou como um *applet*.

17.5 Como um *applet*, o programa Bolas que Saltam não se comporta bem, no sentido de que a animação continua a executar, mesmo quando o *browser* é minimizado ou se move para fora da página. Corrija esse problema acrescentando métodos `start` e `stop` ao applet; no primeiro caso, chame o método `start` da *thread*. No método `stop` do *applet*, use as idéias descritas na Seção 16.3.2 para parar a *thread*. Faça os métodos `start` e `stop` do *applet* escreverem o número de bolas ativadas/paradas (isto é, o tamanho do vetor) em `System.out`.

17.6 Compile e execute o exemplo das Bolas que Saltam da Seção 17.3. Quantas mensagens são necessárias para que o produtor se retarde até a velocidade do consumidor?

17.7 Adicione um segundo produtor e um segundo consumidor ao exemplo do *bounded buffer* usando os mesmos parâmetros. Modifique as mensagens de maneira que fique claro que o produtor produziu a mensagem de data e que o consumidor a está imprimindo. Quantas mensagens são necessárias para que o produtor se retarde até a velocidade do consumidor?

17.8 No Exercício 17.7 sobre o *bounded buffer* de múltiplos produtores-consumidores, modifique o código para o método de entrada, substituindo o `while` por um `if`. Isso produz uma condição de corrida? Você é capaz de mostrá-la com experimentos?

17.9 Modifique os parâmetros de um dos consumidores do Exercício 17.7 de maneira que ele só durma por dois segundos e repita o experimento.

17.10 Modifique a condição de parada para os consumidores do Exercício 17.7 de maneira que cada um pare quando não houver mais mensagens. Pode ocorrer uma condição de corrida? Explique. Caso possa ocorrer uma condição de corrida, execute o experimento 50 vezes para ver se realmente alguma vez ocorre.

17.11 Modifique o exemplo do Crivo de Eratóstenes de maneira que a impressão seja feita por um servidor de impressão, isto é, uma *thread* que execute um laço por meio de um *buffer* aceitando números primos e imprimindo-os. Como as outras *threads*, ela deve terminar quando receber um número negativo.

17.12 (Problema do Jantar dos Filósofos (Dijkstra, 1971)) Cinco filósofos querem filosofar em conjunto. Eles passam por períodos alternativos entre filosofar e comer. Cinco tigelas e cinco garfos são dispostos ao redor de uma mesa circular, sendo que cada filósofo tem seu próprio lugar na mesa. No centro está uma tigela grande com espaguete que é abastecida continuamente. Para comer, cada filósofo precisa do garfo que está à sua esquerda e do garfo que está à sua direita. Um garfo pode ser usado somente por um filósofo de cada vez. O problema é coordenar o uso dos garfos de maneira que nenhum filósofo fique com fome. Esse problema exemplifica muito bem muitas soluções e muitos problemas encontrados na programação concorrente. Pode facilmente ocorrer o *deadlock* se cada filósofo pegar seu garfo da esquerda e se recusar a liberá-lo até ter comido. Pode ocorrer a inanição se dois filósofos conspirarem contra um terceiro. O acesso exclusivo a um garfo ilustra o problema da exclusão/sincronização mútua.

17.13 (Problema do Barbeiro Dorminhoco (Dijkstra, 1965)) Uma barbearia consiste em uma sala de espera com `n` cadeiras e uma sala que contém a cadeira do barbeiro. Se não houver clientes para serem atendidos, o barbeiro vai dormir. Se um cliente entra na barbearia e encontra o barbeiro dormindo, ele o acorda. Escreva um programa para coordenar o barbeiro e os clientes.

17.14 (Problema dos Fumantes (Patil, 1971)) Considere um sistema com três *threads* fumantes e uma *thread* agente. Cada fumante faz um cigarro e o fuma. Para fumar um cigarro, são necessários três ingredientes: fumo, papel e fósforos. Um dos fumantes tem fumo, um tem papel e o outro tem os fósforos. O agente tem um suprimento infinito dos três itens. O agente coloca dois dos ingredientes na mesa. O fumante que tem o terceiro ingrediente pode então fazer um cigarro e fumar, avisando o agente quando terminar. O agente então coloca dois dos outros três ingredientes, e o ciclo se repete. Escreva um programa isento de *deadlock* para sincronizar o agente e os fumantes.

17.15 Faça o download e execute o aplicativo cliente-servidor da Seção 17.4.2. Para fazer isso, inicie primeiro o servidor, e depois inicie quatro clientes acessando o servidor a partir de diferentes máquinas na rede.
(a) Você pode criar os resultados de votação ilustrados pelo exemplo na Seção 17.4.2?
(b) Agora recrie os mesmos resultados de votação variando a seqüência na qual os quatro votantes iniciam e terminam suas sessões.

17.16 Considere o aplicativo cliente-servidor da Seção 17.4.2. Caracterize cada uma das seguintes modificações como "simples" ou "complexa" e discuta rapidamente como ela poderia ser feita.
(a) Revise o programa de maneira que um votante possa votar em apenas um candidato, e não em mais de um.
(b) Revise o programa de maneira que qualquer número de candidatos possa receber votos, sendo esse número fornecido por um argumento ao servidor.
(c) Revise o programa de maneira que cada votante possa ver os valores atuais da variável compartilhada `tally` no início da sessão e após depositar o voto.

17.17 Se a parte c do exercício anterior não for resolvida cuidadosamente, o valor da variável `tally` mostrado ao votante após depositar o voto pode não refletir corretamente o voto daquele votante em relação ao valor mostrado no início da sessão. Explique o problema da sincronização aqui.

18 Corretude de Programa

"Para se tratar cientificamente a programação, deve ser possível especificar precisamente as propriedades necessárias dos programas. O formalismo não é necessariamente um fim por si só. A importância das especificações formais deve, no final, se justificar pela sua utilidade – sejam elas usadas ou não para melhorar a qualidade do software ou para reduzir os custos de produção e manutenção do software."

J. Horning

VISÃO GERAL DO CAPÍTULO

As linguagens de programação são poderosos veículos para projetar e implementar software. Sistemas complexos de software são difíceis de ser bem projetados, e, muitas vezes, o sistema resultante está cheio de erros. Muito se tem escrito sobre a necessidade de me-

lhores metodologias e ferramentas para projeto de software confiável, e nos últimos anos algumas dessas ferramentas começaram a se tornar promissoras.

Em nosso estudo de linguagens de programação, é oportuno examinar a questão das características da linguagem que suportam o projeto de sistemas de software confiáveis e de como essas características ampliam o poder expressivo das linguagens convencionais. Este capítulo trata do problema da corretude de programas a partir da perspectiva das características da linguagem e dos paradigmas de programação.

Intuitivamente, um programa "correto" é aquele que faz exatamente o que os projetistas e usuários desejam que ele faça em todas as situações possíveis. Um programa "formalmente correto" é aquele cuja corretude pode ser provada matematicamente, pelo menos até o ponto em que os projetistas e usuários estejam convencidos sobre sua relativa ausência de erros.

Para que um programa seja formalmente correto, deve haver uma maneira de especificar precisamente (isto é, matematicamente) o que o programa se destina a fazer, para todos os valores válidos de possíveis entradas. Essas linguagens, chamadas *linguagens de especificação*, são baseadas na lógica matemática, que é revista na Seção B.3 do Apêndice B. Uma linguagem de especificação de uma linguagem de programação é baseada em um conceito chamado *semânticas axiomáticas* (Hoare, 1969). O uso das semânticas axiomáticas para provar a corretude de pequenos programas é introduzido na Seção 18.1.

Embora tenham sido elaboradas provas formais de pequenos programas e exercícios de apoio, nosso principal interesse está em desenvolver uma noção sobre como especificar programas de forma que tais provas sejam possíveis. Acreditamos que especificar adequadamente o comportamento do programa é algo que se tornará cada vez mais importante. O teste das propriedades dos programas será conseguido por meio de ferramentas em tempo de compilação ou por intermédio de provadores de teoremas como o PVS (Owre et al., 1992). Essas ferramentas podem ser o prenúncio de uma nova era no desenvolvimento de software na qual serão produzidos programas com uma quantidade de erros dramaticamente menor a um custo mais baixo do que o das práticas atuais.

Até recentemente, as linguagens de modelagem de software foram desenvolvidas como ferramentas separadas, e não estavam totalmente integradas com as linguagens populares e os compiladores usados pelos programadores do mundo real. Ao contrário, essas linguagens, como a Universal Modeling Language (UML) (Blaha e Rumbaugh, 2005), proporcionam uma ferramenta gráfica que inclui a Object Constraint Language (OCL) (Warmer e Kleppe, 1999) para modelar propriedades de objetos e suas inter-relações em um projeto de software. Devido à sua separação do código compilado e à falta de semânticas bem definidas, essas linguagens de modelagem têm servido principalmente como documentação de software e como ferramenta para visualizar um projeto de software.

Em contraste, o surgimento das linguagens Eiffel (Meyer, 1992), ESC/Java (Flanagan et al., 2002), Spark/Ada (Barnes, 1997), JML (Leavens e Cheon, 2004) e a noção de *projeto por contrato* (Meyer, 1997) estão começando a mudar essa situação rapidamente. Esses desenvolvimentos proporcionam aos programadores o acesso a rigorosas ferramentas e técnicas de verificação que estão totalmente integradas com o próprio sistema em tempo de execução. Projeto por contrato é um formalismo pelo qual as interações entre objetos e seus clientes podem ser descritas precisamente e verificadas dinamicamente. ESC/Java é uma linguagem em nível de código para comentar e verificar estaticamente um programa quanto a uma ampla variedade de erros comuns.

A Java Modeling Language (JML) proporciona extensões no nível de código à linguagem Java, de forma que os programas podem incluir essas especificações formais e permitir sua aplicação em tempo de execução. Spark/Ada é um sistema proprietário que proporciona extensões similares à linguagem Ada. Para explorar o impacto desses avanços na corretude de programas, ilustramos o uso da JML na Seção 18.2 e o projeto por contrato na Seção 18.3.

```
int Max (int a, int b) {
    int m;
    if (a >= b)
        m = a;
    else
        m = b;
    fi
    return m;
}
```

| Figura 18.1 Uma Função Max

Os programas funcionais, devido à sua grande aproximação com as funções matemáticas, proporcionam um veículo mais direto para a prova formal da corretude de um programa. Na Seção 18.4 discutimos a aplicação de técnicas de prova nos programas funcionais usando exemplos em Haskell.

18.1 SEMÂNTICAS AXIOMÁTICAS

Embora seja importante para os projetistas de software produzir um bom projeto, é também importante poder *especificar* o que se *supõe* que o programa deva fazer. Para que o PVS ou qualquer outro provador de teorema seja capaz de provar, acima de uma dúvida razoável, que o programa e suas especificações estão formalmente de acordo umas com as outras, o programador deve ser capaz de especificar o que o programa deve fazer. Quando a prova está completa, dizemos que o programa está "correto".

Por exemplo, suponha que queremos provar que a função `Max`[1] na Figura 18.1 realmente fornece como resultado o valor máximo de dois argumentos quaisquer que correspondem aos seus parâmetros `a` e `b`. Chamando essa função uma vez, obteremos uma resposta para um determinado par de argumentos para `a` e `b`, como 8 e 13. Mas cada um dos parâmetros `a` e `b` define um amplo intervalo de valores inteiros – algo na ordem de 4 milhões. Assim, chamar essa função 16 trilhões de vezes, cada vez com um par de valores diferente para `a` e `b`, para provar sua corretude, seria uma tarefa inviável.

As semânticas axiomáticas proporcionam um veículo para raciocínio sobre programas e suas computações. Isso permite aos programadores demonstrar o comportamento de um programa de uma maneira mais convincente do que executar o programa várias vezes usando valores de entrada aleatórios como casos de teste.

18.1.1 Conceitos Fundamentais

A semântica axiomática é baseada na noção de uma *asserção*.

> **Definição**: Uma *asserção* é um predicado boleano que descreve o *estado* de um programa em um ponto de sua execução.

1. Por razões que se tornarão óbvias, neste capítulo nós alteramos ligeiramente a sintaxe de Clite para eliminar o uso de colchetes. Adotamos uma convenção similar a Algol 68 (veja a Seção 2.1.5) na qual as instruções *if* terminam com um `fi` e as instruções *while* com um `od`.

Uma asserção pode definir o significado de uma computação como, por exemplo, *"m é igual ao máximo entre a e b"* sem se preocupar sobre como essa computação é feita.

O código da Figura 18.1 é apenas uma maneira de expressar algoritmicamente o cálculo de um valor máximo; mesmo para uma função tão simples como essa, há outras variações. Não importa qual variação seja usada, pode ser usada a seguinte asserção Q para descrever a função *Max* declarativamente:

$$Q \equiv m = max(a, b)$$

Isto é, esse predicado especifica o significado matemático da função `Max(a, b)` para quaisquer valores inteiros `a` e `b`. Ele então descreve *qual* deverá ser o resultado, e não *como* ele deverá ser computado. Para provar que o programa na Figura 18.1 realmente calcula *max(a, b)*, devemos provar que a expressão lógica Q é válida para todos os valores de `a` e `b`. Nesse exercício de verificação formal, Q é chamado de *pós-condição* para o programa `Max`.

> **Definição**: A *pós-condição* de um programa é uma asserção que define o resultado do programa.

A semântica axiomática nos permite desenvolver uma prova direta, raciocinando sobre o comportamento de cada instrução individual no programa, começando com a pós-condição Q e a última instrução e trabalhando em ordem reversa. O predicado final, digamos P, que é derivado nesse processo, é chamado *pré-condição* do programa.

> **Definição**: Uma *pré-condição* de um programa é uma asserção que expressa o que deve ser *verdade* antes de começar a execução do programa para que a pós-condição seja válida.

No caso de `Max`, a pós-condição Q pode ser satisfeita para qualquer par de valores inteiros de `a` e `b`. Isso sugere a seguinte pré-condição:

$$P \equiv \textit{verdade}$$

Isto é, para que se prove que o programa está correto, não são necessárias restrições ou pré-condições sobre os valores de `a` e `b`. Uma pré-condição assim tão frágil nem sempre é apropriada. Por exemplo, se estivéssemos tentando provar a corretude de uma função `Sqrt(x)`que calcula a raiz quadrada do tipo `float` de um valor `float` de `x`, uma pré-condição apropriada seria $P \equiv x \geq 0$. Voltaremos a esse exemplo particular mais adiante neste capítulo.

Outra consideração deve ser mencionada antes de examinarmos os detalhes das provas de corretude. Isto é, para *alguns* valores iniciais das variáveis que satisfazem à pré-condição P do programa, a execução do programa pode *nunca* chegar à sua última instrução. Essa situação pode ocorrer quando temos um dos seguintes eventos anormais:

1. O programa tenta calcular um valor que não pode ser representado na máquina (virtual) em que ele está executando.
2. O programa entra em um laço infinito.

Para ilustrar esse evento, suponha que escrevemos um programa para calcular o fatorial de n para um valor muito grande de n. Por exemplo, $n = 21$ dá como resultado $n! = 51090942171709440000$, que não pode ser calculado se usarmos inteiros de 32 ou

64 bits. A tentativa de executar um cálculo desses faria a execução normal ser interrompida por uma exceção de estouro de inteiro.[2]

Nesta seção, focalizamos a prova da corretude do programa somente para aqueles valores iniciais de variáveis nas quais não ocorre nenhum desses eventos anormais e o programa é executado até o fim. Essa noção restrita de corretude é chamada *corretude parcial*. Em uma seção mais adiante, abordamos novamente a questão de corretude de programas para casos em que as exceções são geradas em tempo de execução.

Pesquisas recentes desenvolveram ferramentas e técnicas pelas quais a manipulação de exceção pode ser incorporada nas especificações formais de um programa, permitindo assim que a corretude seja estabelecida mesmo quando ocorre uma terminação anormal do programa. No entanto, o segundo evento anormal citado, quando um programa entra em um laço infinito, não pode ser incluído automaticamente para o caso geral. Isso é assegurado pela impossibilidade de solucionar-se o problema da parada, um resultado bem conhecido da teoria da computação que confirma a não-existência de um algoritmo que pode determinar se um programa arbitrário encerra sua execução para todas as entradas possíveis.

Provas de terminação para um determinado programa e laço geralmente podem ser criadas trivialmente pelo programador. Por exemplo, um laço `for` C++/Java que tenha ligações explícitas e incremento diferente de zero define uma seqüência finita de valores para a variável de controle. Assim, qualquer laço desse tipo sempre termina. Por outro lado, uma prova de terminação para um laço `while`, embora muitas vezes relativamente simples, não é possível no caso geral, já que a terminação de um laço reverte no fim à questão de se resolver o problema da paralisação.

Apesar dessas considerações, podemos provar a corretude (parcial) de um programa colocando sua pré-condição antes de sua primeira instrução e sua pós-condição após sua última instrução, e então derivando sistematicamente uma série de predicados válidos juntamente com o significado do código do programa, uma instrução por vez. Para qualquer instrução ou série *s* de instruções, o predicado

$$\{P\}\ s\ \{Q\}$$

representa formalmente a idéia de que *s* está parcialmente correta com relação à pré-condição *P* e à pós-condição *Q*. Essa expressão é chamada *tripla de Hoare* porque ela apareceu primeiro na proposta de C. A. R. Hoare para axiomatizar a semântica das linguagens de programação (Hoare, 1969).

> **Definição**: Um programa (ou instrução) *s* é *parcialmente correto* com relação às asserções *P* e *Q* se, sempre que *s* inicia a execução em um estado que satisfaz *P*, o estado resultante satisfaz *Q* (supondo que a instrução *s* termina).
>
> **Definição**: Um programa (ou instrução) *s* é *totalmente correto* com relação às asserções *P* e *Q* se *s* for parcialmente correta com relação às asserções *P* e *Q* e se, sempre que *s* começa a execução em um estado que satisfaz *P*, garante-se que *s* termina.

Para provar a corretude parcial de nosso exemplo de programa, precisamos mostrar a validade da tripla de *Hoare*:

2. A máquina virtual Java, curiosamente, não inclui estouro de inteiro entre suas exceções, embora ela inclua a divisão por zero. Assim, o cálculo de 21! por um programa Java equivalente àquele na Figura 18.3 dá o resultado incorreto de -1195114496, e não apresenta nenhuma exceção em tempo de execução! No entanto, Haskell faz esse cálculo corretamente para todo valor de *n*, já que ela suporta aritmética para inteiros arbitrariamente grandes.

Tabela 18.1 Regras de Inferência para Diferentes Tipos de Instruções

Tipos de Instrução	Regra de Inferência
1. Atribuição	$\vdash \{Q[t \leftarrow (\text{e})]\}\ \text{t} = e\ \{Q\}$
2. Seqüência	$\{P\}\ s_1\ \{R\},\ \{R\}\ s_2\ \{Q\} \vdash \{P\}\ s_1\ s_2\ \{Q\}$
3. Salto	$\{Q\};\ \{Q\}$
4. Condicional	$\{\text{teste} \wedge P\}\ s_1\ \{Q\},\ \{\neg\ (\text{teste}) \wedge P\}\ s_2\ \{Q\} \vdash$ $\{P\}$ if (teste) s_1 else s_2 fi $\{Q\}$
5. Laço	$\{\text{teste} \wedge P\}\ s_1\ \{P\{,\ P \wedge \neg(\text{teste}) \Rightarrow Q \vdash$ $\{P\}$ while (teste) s_1 od $\{Q\}$
6. Regra de Conseqüência da Pré-condição	$P \Rightarrow P',\ \{P'\}\ s\ \{Q\} \vdash \{P\}\ s\ \{Q\}$
7. Regra de Conseqüência da Pós-condição	$\{P\}\ s\ \{Q'\},\ Q' \Rightarrow Q \vdash \{P\}\ s\ \{Q\}$

$$\{true\}\ \max\ \{m = max(a, b)\}$$

em que `max` representa o corpo da função na Figura 18.1. Fazemos isso derivando as triplas de *Hoare* intermediárias $\{P\}\ s\ \{Q\}$ válidas para instruções s individuais no programa, começando com a última instrução e com a pós-condição Q do programa. Esse processo continua até que tenhamos derivado uma tripla de *Hoare* como aquela acima, que completa a prova de corretude.

Como são derivadas essas triplas de *Hoare* intermediárias? Isso é feito usando-se regras de inferência que caracterizam o que sabemos sobre o comportamento dos diferentes tipos de instruções na linguagem. Programas em linguagens imperativas simples têm os seguintes tipos diferentes de instruções: *Atribuição*, *Seqüência*, *Salto*, *Condicional* e *Laço*. Cada tipo de instrução tem uma *regra de inferência* que define o sentido daquele tipo de instrução em termos das pré e pós-condições que ela satisfaz. As regras para esses tipos de instrução são mostradas na Tabela 18.1.

Quanto à notação na Tabela 18.1, observamos que essas regras são da forma $p \vdash q$, que é similar àquela regra usada na discussão do cálculo de predicado na Seção B.3. Para interpretar essas regras, se a(s) expressão(ões) à esquerda de $\vdash$ aparecem em uma prova, elas podem ser sintaticamente substituídas pela expressão à direita. Depois, observamos que a vírgula (,) em regras da forma $p_1, p_2 \vdash q$ representa conjunção. Essa forma então deverá ser lida da seguinte maneira: "se p_1 e p_2 forem válidas, então q é válida".

Finalmente, observamos que normalmente as provas vão dos axiomas e das regras de inferência até aquilo que deve ser provado. Aqui, ao contrário, tudo se processa em ordem reversa, daquilo que deve ser provado para aquilo que é conhecido, e usualmente isso se chama *derivação da prova*. Uma vez obtida, a derivação da prova poderia ser transformada em uma prova invertendo-se os passos. Na dedução da prova original, se temos r, e para provar r precisamos aplicar uma regra de inferência da forma $p, q \vdash r$, isso é escrito como:

$$\frac{p,q}{r}$$

O processo é então repetido separadamente sobre p e q, construindo assim uma árvore de prova, até que todas as asserções a serem provadas sejam equivalentes a *true*.

18.1.2 A Regra de Atribuição

A regra de inferência *Atribuição* não tem nenhuma premissa e garante que sempre podemos derivar a conclusão. A notação Q[$t \leftarrow (e)$] significa "o predicado que resulta ao substituir todas as ocorrências de t em Q por (e)". A substituição é puramente sintática. São usados os parênteses para que o sentido não seja alterado acidentalmente pela associatividade ou precedência dos operadores adjacentes; nos casos em que os parênteses são desnecessários, eles podem ser eliminados.

Considere os exemplos a seguir. A coluna da esquerda mostra uma instrução de atribuição e sua pós-condição. A coluna da direita mostra a pré-condição derivada aplicando-se a Regra de Atribuição. No último exemplo, $i!$ é a função fatorial.

Exemplo	Q[t ← (e)]
{?} $x = 1$ { $x = 1 \wedge y = 4$}	{$1 = 1 \wedge y = 4$}
{?} m = a {$m = max(a, b)$}	{$a = max(a, b)$}
{?} i = i + 1 {$0 \leq i \wedge i < n$}	{$0 \leq i + 1 \wedge i + 1 < n$}
{?} x = a {$x \geq 0$}	{$a \geq 0$}
{?} i = i + 1 {$f * i = i!$}	{$f * (i + 1) = (i + 1)!$}

Deve-se observar vários itens aqui. Conforme mostram o terceiro e o quinto exemplos, todas as ocorrências da variável de destino são simultaneamente substituídas pela expressão de origem. O terceiro exemplo demonstra que a expressão de origem pode conter várias instâncias da variável de destino, nesse caso, `i`. No último exemplo são mantidos os parênteses para preservar o sentido correto.

18.1.3 Regras de Conseqüência

A *Regra de Conseqüência da Pré-Condição* nos permite executar simplificações aritméticas e lógicas em um predicado durante o processo de prova. Particularmente, podemos reforçar uma pré-condição de instrução (isto é, substituir P por P' quando $P \Rightarrow P'$) para fazê-la corresponder melhor à pós-condição da instrução anterior durante o processo de prova.

Em nosso exemplo envolvendo a Figura 18.1, em algum ponto precisaremos mostrar:

$$\{a \geq b\}\ m = a;\ \{m = max(a, b)\}$$

Podemos usar o reforço da pré-condição aplicando a regra 6:

$$\frac{\begin{array}{c} a \geq b \Rightarrow \\ P' \end{array} \quad , \quad \begin{array}{c} \{P'\} \\ m = a; \\ \{m = max(a, b)\} \end{array}}{\{a \geq b\} \quad m = a; \quad \{m = max(a, b)\}}$$

para a asserção desconhecida P'. Esta última pode ser encontrada aplicando-se a Regra de Atribuição, resultando em:

$$\frac{\begin{array}{c} a \geq b \Rightarrow \\ a = max(a, b) \end{array} \quad , \quad \begin{array}{c} \{a = max(a, b)\} \\ m = a; \\ \{m = max(a, b)\} \end{array}}{\{a \geq b\} \quad \text{m = a;} \quad \{m = max(a, b)\}}$$

Como P'no lado direito da árvore de prova foi obtido usando a Regra de Atribuição, somente a implicação na árvore de prova da esquerda precisa ser provada.

A regra de conseqüência da pré-condição também sugere que qualquer uma dentre várias alternativas pode ser derivada de uma determinada tripla de Hoare, usando várias propriedades conhecidas dos domínios da matemática e da lógica para as variáveis envolvidas. Essa pré-condição, que é a menos restritiva sobre as variáveis envolvidas, é chamada *pré-condição mais fraca*. Por exemplo, a pré-condição $\{a \geq b\}$ é a pré-condição mais fraca para a atribuição `m = a`; e sua pós-condição $\{m = max(a, b)\}$. Determinar as pré-condições mais fracas é importante porque isso permite simplificação da prova em vários estágios.

Da mesma forma, usando a regra de inferência 7, a Regra de Conseqüência da Pós-condição, podemos fortalecer uma pós-condição (isto é, substituir Q por Q' quando $Q' \Rightarrow Q$) para melhor combinar com a pré-condição da instrução seguinte.

18.1.4 Corretude da Função Max

Uma estratégia para provar a corretude parcial do programa na Figura 18.1 funciona sistematicamente a partir da pós-condição reversa por meio do if, e depois por meio das duas instruções de atribuição nas partes then- e else- até uma derivação da pré-condição para o programa. Se a estratégia for bem-sucedida, teremos derivado uma árvore de prova, cujo inverso demonstra que o programa é *parcialmente correto* em relação às suas pré e pós-condições dadas.

Começamos com a tripla de Hoare

$$\{\ true\ \}\ \text{max}\ \{\ m = max(a, b)\ \}$$

em que max representa o corpo da função na Figura 18.1. No seu nível mais externo, a instrução do programa é uma instrução *if*. Portanto, aplicamos a regra de inferência para uma instrução *Condicional*, resultando na árvore de provas parcial:

$$\frac{\begin{array}{cc}\{\ true \wedge a > b\} & \{true \wedge \neg(a > b)\ \} \\ m = a; & \text{m = b;} \\ \{\ m = max(a, b)\ \} \quad , & \{\ m = max(a, b)\ \}\end{array}}{\{\ true\ \}\ \text{if (a > b)m = a; else m = b; fi}\ \{\ m = max(a, b)\ \}}$$

Em seguida, continuamos a prova da subárvore esquerda observando que a instrução é uma instrução de atribuição. Portanto, como fizemos na seção anterior, usamos a regra de inferência 6 para substituir a pré-condição $\{\ true \wedge a > b\ \}$ por uma pré-condição desconhecida, depois usamos o axioma Atribuição para gerar a pré-condição desconhecida, ficando apenas uma implicação para ser provada.

$$\frac{\begin{array}{cc}true \wedge a > b \Rightarrow & \{\ a = max(a, b)\ \} \\ a = max(a, b) \quad , & \text{m = a;} \\ & \{\ m = max(a, b)\ \}\end{array}}{\begin{array}{cc}\{\ true \wedge a > b\} & \{true \wedge \neg(a > b)\ \} \\ \text{m = a;} & \text{m = b;} \\ \{\ m = max(a, b)\ \} \quad , & \{\ m = max(a, b)\ \}\end{array}}$$
$$\{\ true\ \}\ \text{if (a > b)m = a; else m = b; fi}\ \{\ m = max(a, b)\ \}$$

Dadas as definições de $>$ e $max(a, b)$, pode-se mostrar que $a > b \Rightarrow a = max(a, b)$. Uma árvore de prova completa é mostrada na Figura 18.2. Assim, provamos a corretude da tripla de Hoare

$$\{\ true\ \}\ \text{if (a > b) m=a; else m=b; fi}\ \{\ m = max(a, b)\ \}$$

usando as regras de inferência da Tabela 18.1.

$$\dfrac{\dfrac{\{\ a = max(a,b)\}}{true \wedge a > b \Rightarrow} \quad \dfrac{\text{m = a;}}{a = max(a,b)} \quad , \quad \{\ m = max(a,b)\ \}}{\{\ true \wedge a > b\}} \quad \dfrac{\dfrac{\{b = max(a,b)\ \}}{true \wedge \neg(a > b) \Rightarrow} \quad \dfrac{\text{m = b;}}{b = max(a,b)} \quad , \quad \{\ m = max(a,b)\ \}}{\{true \wedge \neg(a > b)\ \}}$$

$$\dfrac{\dfrac{\text{m = a}}{\{\ m = max(a,b)\ \}} \quad , \quad \dfrac{\text{m= b}}{\{\ m = max(a,b)\ \}}}{\{\ true\ \}\ \text{if (a > b) m = a; else m = b; fi}\ \{\ m = max(a,b)\ \}}$$

| Figura 18.2 Corretude Parcial da Função Max

18.1.5 Corretude de Programas com Laços

A corretude (parcial) de um laço depende não somente de se conectar logicamente as pré e pós-condições de sua tripla de Hoare com o restante do programa, mas também da corretude de cada iteração no próprio laço. Para essa finalidade, introduzimos a idéia de *invariante do laço*.

Para ilustrar essas idéias, suponha que queremos provar que a função `Factorial` na Figura 18.3 calcula $n!$, para qualquer inteiro n onde $n \geq 1$, supondo que a função termina normalmente. Com $n!$ queremos dizer o produto $1 \times 2 \times ... \times n$. Ignoramos aqui a possibilidade de estouro de inteiro.

A pré-condição P para `Factorial` é $1 \leq n$, enquanto a pós-condição é $f = n!$. Em geral, um programa que envolve um laço usa a Regra de Seqüência da Tabela 18.1 para dividir o código em duas partes. Como mostra a Figura 18.4, lá, P é a pré-condição do programa, Q é sua pós-condição e R é a *invariante do laço*.

> **Definição**: Uma *invariante de laço* é uma asserção que permanece *verdadeira* antes e depois de qualquer iteração do laço.

O lado direito da Figura 18.4 aplica o esquema geral à esquerda ao problema de provar a corretude do programa fatorial aos dois problemas menores, onde R é a invariante do laço, e, nesse caso, a parte da finalização está vazia. Esses subproblemas podem ser provados em qualquer ordem conveniente.

```
int Factorial (int n) {
    int f = 1;
    int i = 1;
    while (i < n)
        i = i + 1;
        f = f * i;
    od
    return f;
}
```

| Figura 18.3 Uma Função Fatorial

$$
\begin{array}{l|c}
\{\, P \,\} & \\
\text{inicialização} & \\
\{\, R \,\} & \{\, 1 \le n \,\} \quad\quad \{R\,\} \\
\text{while(test)} & \text{f = 1; i = 1;} \quad\quad \text{while . . . od} \\
\quad \text{corpo do laço} & \{\, R \,\} \quad , \quad \{\, f = n! \,\} \\
\text{od} & \{\, 1 \le n \,\}\ \text{Factorial}\ \{\, f = n! \,\} \\
\text{finalização} & \\
\{\, Q \,\} &
\end{array}
$$

| Figura 18.4 Triplas de Hoare para um Programa com um Laço

Em geral, não há uma maneira algorítmica para derivar uma invariante do laço a partir das pré e pós-condições do programa.[3] No entanto, é preciso que o programador forneça uma invariante para cada laço de um programa, se tiver que ser provada a corretude do programa inteiro.

Uma das razões para que se aprenda a provar a corretude de um programa é que, muitas vezes, uma invariante do laço não contém todas as informações necessárias para completar a prova. Em um caso desses, uma prova feita por um provador automático de teoremas falhará. O programador deve então fortalecer a invariante do laço, adicionando novas informações. Para a função *Factorial* dada na Figura 18.3, a invariante do laço é $R = \{\, 1 \le i \wedge i \le n \wedge f = i! \,\}$. Se deixarmos de fora qualquer um dos três conjuntos, a prova falhará.

Começando na base da árvore de prova, o primeiro passo (como mostra a Figura 18.4) é desmembrar o problema em duas partes. A prova do código de inicialização é dada na Figura 18.5. O primeiro passo aplica a Regra da Seqüência e depois a Regra da Atribuição para descobrir a asserção intermediária de ligação, que efetivamente prova a

$$
\dfrac{\begin{array}{c}
1 \le n \\ \Rightarrow \\ 1 \le 1 \wedge 1 \le n \\ \wedge\, (1 = 1!)
\end{array}
\quad
\begin{array}{c}
\{\, 1 \le 1 \wedge 1 \le n \\ \wedge\, (1 = 1!) \,\} \\ \text{f = 1;} \\ \{\, 1 \le 1 \wedge 1 \le n \\ \wedge\, (f = 1!) \,\}
\end{array}
\quad
\begin{array}{c}
\{\, 1 \le 1 \wedge 1 \le n \\ \wedge\, (f = 1!) \,\} \\ \text{i = 1;} \\ \{\, 1 \le i \wedge i \le n \\ \wedge\, (f = i!) \,\}
\end{array}}
{\begin{array}{c}
\{\, 1 \le n \,\} \\ \text{f = 1;} \\ \{\, 1 \le 1 \wedge 1 \le n \\ \wedge\, (f = 1!) \,\}
\end{array}
\quad , \quad
\begin{array}{c}
\{\, 1 \le i \wedge i \le n \\ \wedge\, (f = i!) \,\}
\end{array}}
$$

$$
\{1 \le n\,\}\ \text{f = 1; i = 1;}\ \{\, 1 \le i \wedge i \le n \wedge (f = i!) \,\}
$$

| Figura 18.5 Prova de Corretude da Função Factorial – Inicialização

3. Encontrar uma invariante do laço, muitas vezes, é complicado. Aconselhamos os leitores interessados a procurar informações adicionais (como em Gries, 1981) que desenvolvem esse tópico com mais detalhes.

corretude da subárvore da direita. O processo é repetido usando-se a Regra de Conseqüência da Pré-Condição para desmembrar a prova da subárvore esquerda em duas partes, em que a asserção de ligação é derivada na subárvore direita usando-se a Regra de Atribuição. Com isso, resta apenas a prova da implicação a ser feita:

$$\begin{array}{lcl} 1 = 1! & \equiv & \mathit{true} \\ 1 \leq 1 & \equiv & \mathit{true} \\ 1 \leq n & \Rightarrow & 1 \leq n \end{array}$$

A segunda instrução na metade direita da Figura 18.4 é um laço, então aplicamos nossa Regra do Laço, obtendo:

$$\frac{\begin{array}{cc} \{\ (\mathrm{i} < n) \wedge & \neg(i < n) \\ 1 \leq i \wedge i \leq n \wedge f = i!\ \} & 1 \leq i \wedge \\ \mathrm{i} = \mathrm{i} + 1; & i \leq n \wedge \\ \mathrm{f} = \mathrm{f} * \mathrm{i}; & f = i! \Rightarrow \\ \{1 \leq \mathrm{i} \wedge \mathrm{i} \leq \mathrm{n} \wedge \mathrm{f} = \mathrm{i}!\ \} \ , & f = n! \end{array}}{\{1 \leq i \wedge i \leq n \wedge f = i!\} \text{ while } \ldots \text{ od } \{\, f = n!\}}$$

O lado direito é mais fácil de provar, porque envolve somente uma implicação. Em termos da Figura 18.4, devemos mostrar que $\neg \mathit{test} \wedge R \Rightarrow Q$. Isso pode ser mostrado por intermédio de nossas habilidades algébricas:

$$\begin{array}{r} \neg(i < n) \wedge 1 \leq i \wedge i \leq n \wedge f = i! \Rightarrow \\ (i = n) \wedge f = i! \Rightarrow \\ f = n! \end{array}$$

Isto é, como $\neg(i < n)$ e $i \leq n$, segue que $i = n$.

Dessa forma, provamos que o predicado de laço é, sem dúvida, uma invariante do laço. Essa prova é dada na Figura 18.6. Como o corpo do laço é uma seqüência, aplicamos a Regra da Seqüência e depois a Regra da Atribuição para derivar a asserção de ligação. O processo é então repetido com o uso da Regra de Conseqüência da Pré-Condição e novamente a Regra da Atribuição na subárvore direita para derivar a asserção de ligação. Assim, resta

$$\frac{\dfrac{\begin{array}{cc} i < n \wedge & \{\ i \leq n \wedge \\ 1 \leq i \wedge i \leq n & 1 \leq i \wedge \\ f * i = i! & f * (i + 1) = (i + 1)!\} \\ \Rightarrow & \mathrm{i} = \mathrm{i} + 1; \\ i + 1 \leq n \wedge 1 \leq i + 1 & \{\ i \leq n \wedge 1 \leq i \wedge \\ f * (i + 1) = (i + 1)! \ , & \wedge f * i = i!\ \} \end{array}}{\begin{array}{c} \{i < n \wedge 1 \leq i \wedge i \leq n \wedge f = i!\} \\ \mathrm{i} = \mathrm{i} + 1; \\ \{i \leq n \wedge 1 \leq i \wedge f * i = i!\} \end{array}} \quad \begin{array}{c} \{\ i \leq n \wedge 1 \leq i \\ \wedge f * i = i!\} \\ \mathrm{f} = \mathrm{f} * \mathrm{i}; \\ \{i \leq n \wedge 1 \leq i \\ \wedge f = i!\ \} \end{array}}{\{\ i < n \wedge 1 \leq i \wedge i \leq n \wedge f = i!\ \}\ \mathrm{i} = \mathrm{i} + 1; \mathrm{f} = \mathrm{f} * \mathrm{i}; \{\ 1 \leq i \wedge i \leq n \wedge f = i!\}}$$

| Figura 18.6 Prova de Corretude da Função Factorial – Corpo do Laço

apenas uma implicação na subárvore da extrema esquerda para ser provada:

$$
\begin{array}{lcl}
i < n & \Rightarrow & i + 1 \leq n \\
1 \leq i & \Rightarrow & i \leq i + 1 \\
f = i! & \Rightarrow & f * (i + 1) = (i + 1)! \\
 & & \text{pois } 1 \leq i + 1
\end{array}
$$

Isso conclui nossa prova da corretude (parcial) da função *Factorial* na Figura 18.3. Para provar que o algoritmo termina, devemos apenas identificar um valor inteiro que decresce em cada iteração do laço e possui limite inferior igual a zero. Esse valor é $n - i$, pois a invariante do laço inclui o conjunto $i \leq n$ e i aumenta 1 em cada iteração do laço. Assim, mostramos a corretude total na ausência de exceções.

Observe que nossa prova não trata da corretude quando o cálculo de $n!$ não pode ser completado, porque foi passado um valor muito grande de n. Voltaremos a esse assunto na Seção 18.2.

18.1.6 Perspectivas em Métodos Formais

As semânticas axiomáticas e as técnicas correspondentes para provar a corretude dos programas imperativos foram desenvolvidas na década de 1960 e no início da década de 1970. Naquela época, as pessoas esperavam que a maioria dos programas fosse rotineiramente provada como correta e que os produtos de software se tornassem mais confiáveis em geral. Dado o estado da indústria de software nos dias atuais, está claro que essas expectativas estão longe de ser atendidas.

Uma área chamada *métodos formais* no projeto de software surgiu nos últimos 20 anos. Essa área tenta desenvolver e aplicar ferramentas e técnicas de corretude a duas fases diferentes do processo de desenvolvimento de software – especificações de software e validação de software (teste). Ferramentas como a Universal Modeling Language (UML) e a Java Modeling Language (JML), por exemplo, surgiram para ajudar os projetistas a especificar mais formalmente o comportamento dos componentes em grandes sistemas. Técnicas como *projeto por contrato* (Meyer, 1997) foram propostas para prover uma base sobre a qual os componentes de software podem ser validados com um maior grau de confiabilidade do que as várias técnicas de teste do passado.

Alguns cientistas da computação argumentam que provas formais de corretude de programa são uma tarefa inútil. A utilidade e a importância das provas de corretude em projeto de software continuam sendo assunto de um caloroso debate, especialmente na última década. Muitos engenheiros de software rejeitam o uso dos métodos formais para validação de software (Millo et al., 1979), argumentando que se trata de um processo muito complexo e demorado para ser dominado pela maioria dos programadores. Em vez disso, eles sugerem o uso de métodos de teste mais elaborados para convencer os projetistas e os usuários de que o software funciona corretamente na maioria das vezes.

Um contra-argumento a essa visão foi apresentado muitos anos atrás por Dijkstra (1972), que simplesmente reconheceu que o teste podia apenas provar a presença de erros, mas nunca provar sua ausência. Por exemplo, um simples programa que recebe dois inteiros de 32 bits, computa uma função e apresenta como resultado um inteiro de 32 bits tem 2^{64} entradas possíveis (aproximadamente 10^{20}), de maneira que mesmo que alguém pudesse testar e verificar (!) 100 milhões de casos por segundo, um teste completo desse simples programa levaria aproximadamente 10^5 anos. Imagine quanto tempo levaria para executar um teste completo de um programa maior, usando-se métodos de teste tão exaustivos.

Naturalmente, as provas de corretude também são difíceis e demoradas. Muitos programadores e engenheiros de software não têm o treinamento matemático para incorporar esses métodos formais de verificação no seu processo de projeto. Já tivemos muitos pro-

dutos de software projetados com um uso cuidadoso dos métodos formais, tanto na fase de verificação quanto (mais importante ainda, talvez) na fase de projeto.

Os resultados obtidos nesses projetos têm sido impressionantes – o software resultante apresenta um padrão de confiabilidade e uma robustez muito mais altos do que os do software projetado com o uso de técnicas tradicionais. Porém, muitos argumentam que esse resultado tem um preço – maior tempo de projeto e de custo de desenvolvimento em geral, se comparado com os métodos tradicionais de teste.

Dadas essas permutas, seria razoável perguntar se há um meio-termo entre verificação formal de software complexa e demorada, como aquela descrita acima, e os métodos tradicionais de teste que produzem software bem pouco confiável? Nós achamos que sim.

Primeiro, propriedades de programas que não sejam as da corretude podem ser testadas rotineiramente. Essas propriedades incluem a segurança dos programas quando esta é um problema crítico. A ausência de *deadlock* em programas concorrentes, muitas vezes, também pode ser testada formalmente.

Segundo, os métodos que definem os comportamentos de uma classe em um programa orientado a objetos freqüentemente são bem pequenos. Provas informais de corretude para esses métodos rotineiramente são possíveis, embora freqüentemente não sejam praticadas. Uma razão para isso é que muitos programadores, que na sua maioria não são treinados no uso de lógica formal, não podem nem sequer definir as pré e pós-condições para os métodos que eles escrevem.

No entanto, programadores treinados em corretude de programas podem e fazem asserções de entrada-saída para os métodos que eles escrevem usando inglês formal (ou outra linguagem natural); isso leva a uma documentação muito melhor.

Um exemplo de como esse formalismo poderia ser mais bem utilizado: considere a documentação `javadoc` da Sun para os vários métodos `String` na JDK 1.1.[4] Lá, o comentário de definição para o método `substring`:

```
public String substring(int beginIndex, int endIndex);
```

diz o seguinte:

> Retorna uma nova *string* que é uma *substring* dessa *string*. A *substring* começa no `beginIndex` especificado e se estende até o caractere no índice `endIndex-1`. Gera `IndexOutOfBoundsException` se o `beginIndex` ou o `endIndex` estiver fora do intervalo.

Como poderia um implementador executar uma prova de corretude para `substring`, dada uma especificação tão vaga? Essa especificação precisa ser traduzida em uma representação lógica, de forma que as questões a seguir possam ser respondidas clara e precisamente:

> Quais são os valores válidos para `beginIndex` e `endIndex`?
>
> Para uma dada *string* $s = s_0 s_1 \ldots s_{n-1}$, que resultado é retornado normalmente?
>
> O que acontece no caso anormal, quando qualquer um dos índices não é válido?

Um programador interessado em produzir mesmo uma prova informal de uma implementação da `substring` precisaria pelo menos de uma descrição mais formal das pré e pós-condições desse método. A Figura 18.7 mostra uma dessas descrições (que, para simplificar, omite o caso anormal). Diferentemente da descrição informal, essa descrição especifica formalmente os valores aceitáveis de *beginIndex*, *endIndex* e o tamanho *n* da *string s* para a qual a `substring` é bem definida, bem como a natureza exata do próprio resultado.

4. www.cs.wm.edu/~noonan/1.1.8/api/java.lang.String.html#substring(int,%20int)

$\{0 \leq beginIndex < endIndex \leq n \wedge s = s_0 s_1 \ldots s_{n-1}\}$

s.substring(beginIndex, endIndex);

$\{s = s_0 s_1 \ldots s_{n-1} \wedge result = s_{beginIndex} s_{beginIndex+1} \ldots s_{endIndex-1}\}$

| Figura 18.7 Especificação Formal do Método Java `substring`

Na Seção 18.2, discutimos os aperfeiçoamentos recentes em projeto de linguagem e a metodologia de software que estão ajudando os desenvolvedores a lidar com esses tipos de problema com mais eficiência.

18.2 FERRAMENTAS DE MÉTODOS FORMAIS: JML

Durante os últimos anos, foram desenvolvidas novas ferramentas e técnicas de modelagem para ajudar os desenvolvedores de software a criar especificações mais rigorosas e projetos mais confiáveis. Uma dessas ferramentas é a *Java Modeling Language* (abreviado como JML), que está totalmente implementada e é adaptável a uma variedade de atividades de projeto e verificação de software. Uma técnica promissora de modelagem é chamada *projeto por contrato* (Meyer, 1997), que proporciona uma infra-estrutura operacional dentro da qual podem ser projetados com confiabilidade programas orientados a objetos.

Essas duas técnicas funcionam em conjunto. Isto é, a JML proporciona uma linguagem para incorporar e verificar especificações formais em programas Java, enquanto a técnica de projeto por contrato proporciona as diretrizes operacionais dentro das quais as especificações podem ser usadas para garantir a integridade do sistema quando as classes interagem umas com as outras.

Nesta seção, introduzimos as características da JML da forma como elas se aplicam à especificação e verificação formal de uma função individual, como a função *Factorial*, por exemplo, que especificamos e verificamos manualmente na seção anterior. Mostramos também como a JML nos permite especificar exceções em tempo de execução, fornecendo um veículo mais robusto do que as triplas de Hoare puras em um ambiente de computação real onde realmente ocorrem as exceções.

Considere a versão comentada de JML da função *Factorial* mostrada na Figura 18.8. Essa versão difere do programa da Figura 18.3 em apenas uma forma significativa. Isto é, a função `Factorial` é comentada com dois comentários estilizados (escritos como `/*@...@*/`), um contendo `requires` e `ensures` e o outro começando com `loop_invariant`. O primeiro comentário é a codificação JML para pré e pós-condições P e Q que são usadas para formar triplas de Hoare a partir da função `Factorial` em preparação para sua prova de corretude. O segundo comentário é a codificação JML da asserção R que representa a invariante do laço naquela prova.

Cada uma das cláusulas `requires`, `ensures` e `loop_invariant` tem uma expressão boleana de estilo Java como elemento principal. Variáveis mencionadas nessas cláusulas, como `n`, são as variáveis e os parâmetros comuns que são visíveis ao próprio código da função `Factorial`. Os nomes adicionais mencionados nessas cláusulas são de dois tipos, variáveis locais (como `i` e `j` neste exemplo) e palavras reservadas JML (como `\result` e `\product` neste exemplo). *Uma importante advertência para cláusulas JML como essas é que sua execução não pode ter efeitos colaterais sobre o estado da computação.*

```
public class myFactorial {

/*@ requires 1 <= n;
    ensures \result == (\product int i; 1<=i && i<=n; i);
  @*/
static int Factorial (int n) {
  int f = 1;
  int i = 1;
/*@ loop_invariant i <= n &&
          f == (\product int j; 1 <= j && j <= i; j);
  @*/
  while (i < n) {
    i = i + 1;
    f = f * i;
  }
  return f;
}
public static void main(String[] args) {
   int n = Integer.parseInt(args[0]);
   System.out.println("Factorial of " + n +
                      " = " + Factorial(n));
}
}
```

| Figura 18.8 Uma Versão Java JML–Comentada da Função Factorial

Em JML, a palavra reservada `\result` identifica de forma única o resultado retornado por uma função não-vazia. A palavra reservada `\product` é um quantificador matemático (na Tabela 18.2 há um resumo dos principais quantificadores e operadores JML) e tem o mesmo significado do $\prod$ em expressões matemáticas. O restante da cláusula `ensures` define os limites sobre a variável de controle `i` e a expressão que recebe o cálculo do produto. Assim, a expressão JML `(\product int i; 1<=i && i<=n; i)` é equivalente à expressão matemática $\prod_{i=1}^{n} i$.

Dois pontos significativos precisam ser destacados sobre as cláusulas JML `requires`, `ensures` e `loop_invariant` em relação ao nosso exercício de verificação usando pré e pós-condições P e Q e invariante do laço R na seção anterior. Primeiro, essas três cláusulas estão integradas com o compilador Java e seu sistema de execução, permitindo que suas especificações sejam verificadas quanto à sintaxe, ao tipo e aos erros em tempo de execução. Em segundo lugar, no entanto, essas três cláusulas não proporcionam por si só uma plataforma sobre a qual o código será automaticamente provado como correto por um agente anônimo.[5]

5. Entretanto, foram desenvolvidas ferramentas que funcionam a partir de especificações JML para executar vários níveis de verificação formal. Exemplos dessas ferramentas são ESC/JAVA2 (Flanagan et al., 2002) e LOOP (van den Berg e Jacobs, 2001).

Tabela 18.2 Resumo de Expressões JML

Expressão JML	Significado
`requires p ;`	`p` é uma pré-condição para a chamada
`ensures p ;`	`p` é uma pós-condição para a chamada
`signals (E e) p;`	Quando a exceção tipo `E` é gerada pela
`loop_invariant p;`	`p` é uma invariante do laço
`invariant p ;`	`p` é uma invariante de classe (veja a próxima seção)
`\result == e`	`e` é o resultado retornado pela chamada
`\old(v)`	o valor de `v` na entrada para a chamada
`(\product int x ; p(x); e(x))`	$\prod_{x \in p(x)} e(x)$; isto é, o produto de `e(x)`
`(\sum int x ; p(x); e(x))`	$\Sigma_{x \in p(x)} e(x)$; isto é, a soma de `e(x)`
`(\min int x ; p(x); e(x))`	$\min_{x \in p(x)} e(x)$; isto é, o mínimo de `e(x)`
`(\max int x ; p(x); e(x))`	$\max_{x \in p(x)} e(x)$; isto é, o máximo de `e(x)`
`(\forall type x ; p(x) ; q(x))`	$\forall x \in p(x)$: `q(x)`
`(\exists type x ; p(x) ; q(x))`	$\exists x \in p(x)$: `q(x)`
`p ==> q`	$p \Rightarrow q$
`p <== q`	$q \Rightarrow p$
`p <==> q`	$p \Leftrightarrow q$
`p <=!=> q`	$\neg (p \Leftrightarrow q)$

Assim, podemos concluir que a JML fornece uma linguagem formal para definir as pré-condições, as pós-condições e as invariantes do laço de uma função executável e para verificar automaticamente em tempo de execução que uma chamada à função irá:

1. Satisfazer à pré-condição *P*.
2. Satisfazer a cada uma das invariantes do laço *R* durante a execução da função.
3. Retornar um resultado que satisfaz à pós-condição *Q*.

Embora isso não seja uma verificação formal *per se*, a JML proporciona uma base robusta para integrar as especificações de um programa com seu código. Isso ocorre porque as especificações JML podem ser ativamente compiladas e interpretadas com cada compilação e execução do programa.[6]

Aqui estão os resultados de quatro execuções diferentes para o programa da Figura 18.8, que ilustra vários resultados possíveis. Na primeira execução, o programa executa normalmente, fornecendo apenas o resultado e sem apontar erros.

```
% jmlrac myFactorial 3
Factorial of 3 = 6
```

6. Para compilar um programa Java com especificações JML embutidas, é usada a linha de comando `%jmlc -Q myProgram.java`. Para executar um programa desses, é usada a linha de comando `%jmlrac myProgram`. Os leitores interessados poderão visitar o site http://www.jmlspecs.org, que disponibiliza gratuitamente o software JML e outras documentações.

A segunda execução, uma tentativa de calcular o fatorial de −5, é conseguida com um `JMLEntryPreConditionError`. Isso diz que a chamada ao método `myFactorial.Factorial` viola a pré-condição daquele método `1 <= n`; ela reporta que o argumento atual `'n' is -5`. Como esse evento é uma instância da manipulação de exceção Java, há também um rastreamento (trace) das chamadas de método que estão ativas para esse erro JML.

```
% jmlrac myFactorial -5
Exception in thread "main"
   org.jmlspecs.jmlrac.runtime.JMLEntryPreconditionError:
   by method myFactorial.Factorial regarding specifications at
File "myFactorial.java", line 3, character 15 when
        'n' is -5
        at myFactorial.checkPre$Factorial$myFactorial
          (myFactorial.java:240)
        at myFactorial.Factorial(myFactorial.java:382)
        at myFactorial.main(myFactorial.java:24)
```

A terceira e a quarta execuções mostram um pouco da vulnerabilidade das especificações em relação a algumas características particulares de Java. Como Java não possui a exceção `ArithmeticOverflow`, o cálculo de qualquer valor `int` que ultrapasse $2^{31} - 1 = 2147483647$ dará um resultado incorreto.[7] O maior valor `int` de n para o qual $n! \leq 2^{31} - 1$ é 12.

```
% jmlrac myFactorial 21
Factorial of 21 = -1195114496
% jmlrac myFactorial 32
Factorial of 32 = -2147483648
```

Examinando os resultados da terceira e da quarta execuções, podemos agora entender por que não foi apontado nenhum erro. Isto é, o laço `while` dá o mesmo resultado espúrio calculado pela verificação JML em tempo de execução que foi especificado pela pós-condição. Assim, duas respostas igualmente incorretas criam a ilusão de que tudo está bem com essa função para os argumentos 21 e 32. Na Seção 18.2.1, examinamos novamente a manipulação de exceções em tempo de execução usando especificações JML.

A exceção `JMLEntryPreConditionError` é apenas um dos vários tipos de exceções que podem ocorrer ao executar programas JML anotados. Uma breve descrição desta e de outras exceções JML importantes é dada na Tabela 18.3.

7. Para o tipo Java `long`, o valor máximo é $2^{63} - 1 = 9223372036854775807$, e para o tipo `BigInteger` o valor máximo é ilimitado. Assim, aplicações práticas que calculam fatoriais provavelmente usam valores `BigInteger` para eliminar as possibilidades de estouro. Temos evitado o uso da classe `BigInteger` aqui. Se fizéssemos isso, iríamos introduzir uma enorme bagagem extra no código Java, tornando nossa discussão de especificações formais quase ininteligível.

| **Tabela 18.3** Algumas das Exceções JML Predefinidas

Exceção JML	Significado
`JMLEntryPreconditionError`	Os parâmetros da chamada de um método não satisfazem à cláusula `requires` do método
`JMLNormalPostconditionError`	Uma chamada de método encerra normalmente, mas seu resultado não satisfaz à cláusula `ensures` do método
`JMLExceptionalPostconditionError`	Uma chamada de método encerra anormalmente, gerando uma exceção definida pela cláusula `signals` do método
`JMLLoopInvariantError`	Alguma execução de um corpo de laço não satisfaz a sua cláusula `loop_invariant`
`JMLInvariantError`	Alguma chamada a um método ou construtor não deixa o objeto em um estado que satisfaça à cláusula `invariant`

No entanto, suponha que o laço `while` na Figura 18.8 fosse alterado ligeiramente para introduzir um erro no cálculo do fatorial, substituindo a linha de código `while (i < n)` pela linha `while (i <= n)`. Isso causaria o aparecimento do seguinte erro quando fosse verificada a invariante do laço:

```
% jmlrac myFactorial 3
Exception in thread "main"
   org.jmlspecs.jmlrac.runtime.JMLLoopInvariantError:
       LOOP INVARIANT:
   by method myFactorial.Factorial regarding specifications at
File "myFactorial.java", line 9, character 24 when
         'n' is 3
         at myFactorial.internal$Factorial(myFactorial.java:102)
         at myFactorial.Factorial(myFactorial.java:575)
         at myFactorial.main(myFactorial.java:211)
```

Aqui, a mensagem `'n' is 3` indica que o valor resultante de `n` não satisfaz à invariante do laço especificada na Figura 18.8. A informação dada nessa mensagem não conta realmente a história toda. Isto é, teria sido útil ver o valor de `i` e a invariante do laço, bem como `n`, já que é esse valor que faz a invariante se tornar *falsa*.[8]

8. Devemos enfatizar que a JML era um trabalho em andamento à época desta publicação. Talvez, em uma versão futura, ela relate os valores de todas as variáveis em tais expressões, quando ocorrer um erro.

Se não tivéssemos incluído a invariante do laço no programa, seria apontada a exceção a seguir pela linha incorreta `while (i <= n)`:

```
% jmlrac myFactorial 3
Exception in thread "main"
   org.jmlspecs.jmlrac.runtime.JMLNormalPostconditionError:
   by method myFactorial.Factorial regarding specifications at
File "myFactorial.java", line 4, character 23 when
   'n' is 3
   '\result' is 24
   at myFactorial.checkPost$Factorial$myFactorial
       (myFactorial.java:321)
   at myFactorial.Factorial(myFactorial.java:392)
   at myFactorial.main(myFactorial.java:24)
```

Essa mensagem mostra um desacordo entre as cláusulas `ensures` e o resultado retornado pela função, conforme apontado pela linha `'\result' is 24`. Aqui, novamente, o programador fica sabendo somente que há alguma coisa errada com a especificação ou alguma coisa errada com o código do programa (ou ambos).

Um desacordo entre a invariante do laço e seu código poderia muito bem apontar um erro na invariante e não no código. Por exemplo, se inserirmos uma invariante incorreta na Figura 18.4, como, por exemplo, escrever erradamente `j <= i` como `j < i`, será gerada novamente uma exceção invariante do laço:

```
% jmlrac myFactorial 3
Exception in thread "main"
   org.jmlspecs.jmlrac.runtime.JMLLoopInvariantError:
       LOOP INVARIANT:
   by method myFactorial.Factorial regarding specifications at
File "myFactorial.java", line 9, character 24 when
        'n' is 3
        at myFactorial.internal$Factorial(myFactorial.java:101)
        at myFactorial.Factorial(myFactorial.java:573)
        at myFactorial.main(myFactorial.java:209)
```

Mas, dessa vez, é a invariante que precisará ser corrigida, e não o código.

Um outro benefício da verificação em tempo de execução da pré e pós-condição é que o programador pode inserir uma implementação diferente de uma função no programa, e depois testá-la usando as mesmas pré e pós-condições. Por exemplo, suponha que decidimos implementar a função `Factorial` recursivamente em vez de iterativamente, com o seguinte código:

```
static int Factorial (int n) {
  if (n < 2) return n;
  else return n*Factorial(n - 1);
}
```

```
/*@ requires 1 <= n;
    ensures \result == (\product int i; 1<=i && i<=n; i);
    signals (ArithmeticException) n > 12;
@*/
static int Factorial (int n) {
    if (n > 12) throw new ArithmeticException();
    else {
        int f = 1;
        int i = 1;
        /*@ loop_invariant i <= n &&
                  f == (\product int j; 1 <= j && j <= i; j);
        @*/
        while (i < n) {
            i = i + 1;
            f = f * i;
        }
        return f;
    }
}
```

| Figura 18.9 Acrescentando Manipulação de Exceção a uma Especificação JML

As cláusulas `requires` e `ensures` de JML permanecem ambas intactas enquanto compilamos e executamos essa versão; assim podemos imediatamente testar se ela satisfaz às pré-condições e pós-condições.

18.2.1 Manipulação de Exceção JML

Métodos formais para corretude de programas deverão suportar a especificação das condições sob as quais ocorrem as exceções. Para essa finalidade, a JML fornece uma cláusula `signals`:

```
signals (exception) expression ;
```

que pode aparecer juntamente com as cláusulas `requires` e `ensures` da função. Quando aquela `exception` ocorre, é verificada a `expression`; se aquela `expression` não for verdadeira (*true*), é mostrada a `exception` e o programa é interrompido. A Figura 18.9 mostra uma variante da função `Factorial` que incorpora essas idéias.

Agora, quando executamos esse programa para computar o fatorial de um número que causará estouro aritmético, é mostrada uma exceção:

```
% jmlrac myFactorial 13
Exception in thread "main" java.lang.ArithmeticException
        at myFactorial.internal$Factorial(myFactorial.java:9)
        at myFactorial.Factorial(myFactorial.java:610)
        at myFactorial.main(myFactorial.java:213)
```

Leitores atentos observarão que cláusulas `signals` podem ser evitadas em muitos casos, simplesmente se forem escritas pré-condições mais poderosas, de modo que restrinjam entradas que resultem em exceções. Por exemplo, na função `Factorial`, podemos facilmente substituir a cláusula `signals` pela seguinte alteração para as cláusulas `requires`:

```
requires 1 <= n && n < 13;
```

Agora a chamada `Factorial(13)` causará um erro `JMLEntryPrecondition` em lugar de um erro Java `ArithmeticException`.

Em situações mais complexas de projeto de software, a necessidade de especificações JML para apontar exceções explicitamente é mais obrigatória do que nesse exemplo simples. Por exemplo, considere a tarefa de definir especificações JML completas para todas as cláusulas e os métodos na biblioteca de classes Java. Entre esses métodos está o método `substring`, cujas especificações informal e formal foram discutidas anteriormente neste capítulo (veja a Figura 18.7).

Abaixo está uma reprodução estilo JML daquelas especificações formais, com cláusulas `signals` acrescentadas para descrever ações quando as pré-condições não são satisfeitas pela chamada.

```
/*@ requires 0 <= beginIndex && beginIndex < endIndex &&
          endIndex <= s.length();
     ensures \result ==
          "s[beginIndex] s[beginIndex+1] ... s[endIndex-1]";
     signals (StringIndexOutOfBoundsException)
          0 > beginIndex || beginIndex >= endIndex ||
          endIndex > s.length();
  @*/
     s.substring(intbeginIndex, intendIndex)
          throws StringIndexOutOfBoundsException
```

Essa especificação é uma pequena abreviação das especificações completas JML para `substring` usadas para definir toda a biblioteca de classe Java. Particularmente, a linha "`s[beginIndex] s[beginIndex+1]...s[endIndex−1]`" é nossa especificação informal algébrica para o valor do objeto *string* que é retornado pela `substring`. De fato, aquela linha aparece na especificação JML como:

```
this.stringSeq.subsequence(beginIndex,endIndex);
```

Aqui, `stringSeq` é uma classe JML que define *strings* como seqüências de caracteres, e `subsequence` é um método naquela classe. Os leitores interessados podem consultar o site **http://www.jmlspecs.org** para mais detalhes sobre essas convenções.

18.3 CORRETUDE DOS PROGRAMAS ORIENTADOS A OBJETO

Programas orientados a objeto são coleções de classes. Cada classe define um tipo de objeto e um conjunto de características (métodos) que podem transformar aquele tipo de objeto. Ao implementar um programa orientado a objeto, o programador precisa

ter padrões ou ferramentas por meio das quais ele pode formalmente especificar o que cada classe se destina a fazer. Além da formalização das funções individuais, conforme descrevemos na Seção 18.2, os programas orientados a objeto proporcionam duas ferramentas adicionais por meio das quais os programadores podem garantir a corretude.

Primeiro, cada interação na qual um objeto em uma classe é acessado ou modificado por uma chamada de método de uma classe chamada *cliente* deve ser guiada por certas "regras de compromisso". Essas regras garantem que o cliente forneça ao método chamado os valores apropriados para seus parâmetros e que o método chamado retorne um resultado para o cliente que seja consistente com a finalidade do método. Essas regras de compromisso são chamadas de *contrato* entre a classe e o cliente. Quando todas as interações entre classes em um projeto de software seguem essas regras, dizemos que o software foi *projetado por contrato*.

Segundo, sempre que um objeto em uma classe é transformado, ele deve manter o mesmo conjunto de propriedades que o identificam como um membro daquela classe em particular e que ele tinha quando foi criado. Esse conjunto de propriedades é conhecido como *invariante de classe*.

Definimos e ilustramos o uso do projeto por contrato e a invariante de classe nas Seções 18.3.1 e 18.3.2.

18.3.1 Projeto por Contrato

A metodologia do projeto por contrato foi desenvolvida por Bertrand Meyer (Meyer, 1997).

> **Definição**: *Projeto por contrato* é um framework formal que estabelece o que chamamos de obrigações e benefícios (responsabilidades e direitos) entre cada par fornecedor/cliente que pode ocorrer durante a vida do produto de software.

Essa estrutura ajuda os projetistas de software a documentar restrições que aparecem na interface entre classes em interação e efetivamente atribuir a culpa quando as coisas saírem erradas.

A estrutura do projeto por contrato é uma tabela 2×2 que identifica obrigações e benefícios em uma dimensão e classes fornecedor/cliente em outra, como mostra a Tabela 18.4. Tipicamente, o cliente quer chamar um método público que é implementado dentro da classe do fornecedor. Nesse instante, o cliente é obrigado a passar argumentos que satisfaçam à pré-condição do método. Se isso for feito, o cliente poderá ter certeza de que o resultado desejado será computado. Por outro lado, se os parâmetros do cliente não satisfizerem à pré-condição do método, não será possível ter nenhuma confiança na veracidade do resultado. Se esse evento causar um erro, a culpa pelo erro será do cliente.

Para a mesma chamada, o fornecedor tem a obrigação de computar um resultado que satisfaça à pós-condição do método e de deixar o objeto em um estado no qual a chamada *invariante de classe* seja satisfeita (veja a Seção 18.3.2 para uma discussão sobre a invariante de classe). O fornecedor se beneficia do contrato por não ter de incluir código que

Tabela 18.4 A Estrutura de Projeto por Contrato

	Obrigações	Benefícios
Cliente	Satisfaz à pré-condição	O resultado é computado
Fornecedor	Satisfaz à pós-condição	Codificação mais simples

verifica explicitamente que o argumento satisfaz à pré-condição. Se ocorrer um erro e/ou a pós-condição não for atingida, a culpa pelo erro será do fornecedor.

Vamos examinar um exemplo simples de como o projeto por contrato funciona para o cálculo do fatorial discutido na Seção 18.2.

	Obrigações	**Benefícios**
Cliente	Passa $n > 0$	Recebe $n!$ calculado
Fornecedor	Calcula $n!$	Pode supor que $n > 0$

Se o cliente ou o fornecedor não satisfizer suas obrigações, será responsável por qualquer erro que ocorrer durante a chamada. Se ocorrer um erro de pré-condição, o cliente será responsável por aquele erro. Se ocorrer um erro de pós-condição, o fornecedor será responsável.

Se nenhum dos erros ocorrer durante a chamada, então tanto o cliente quanto o fornecedor receberão benefícios apropriados. Especificamente, o cliente pode ter certeza de que o valor $n!$ é retornado corretamente, e o fornecedor não precisa verificar explicitamente que $n > 0$ dentro de seu próprio código.

18.3.2 A Invariante de Classe

Invariante de classe é uma ferramenta que tem a função de garantir que todos os objetos na classe retenham sua integridade durante todo o seu tempo de vida, não importa que métodos sejam aplicados a eles. Na discussão das invariantes de classe, seguimos a abordagem de Meyer (1997). Ilustramos essa abordagem com a formalização da classe `MyStack` que foi apresentada originalmente no Capítulo 13 (Figura 13.6).

Uma *invariante de classe* é uma expressão de valor boleano que especifica as condições sob as quais um objeto naquela classe permanece bem definido. Essa expressão descreve o *estado interno* do objeto usando as variáveis de instância pública e privada da classe. Uma expressão *INV* é uma invariante de classe correta para a classe *C*, se ela satisfaz às duas condições seguintes:

- Toda chamada a um construtor *C* com argumentos que satisfazem à pré-condição de *C* cria um novo objeto com um estado que satisfaz *INV*.
- Toda chamada a um método público *M* com argumentos que satisfazem à pré-condição de *M* deixa o objeto em um estado que satisfaz *INV*.

Assim, a invariante de classe deve se tornar *true* (verdadeira) depois que o objeto for criado por um construtor, e ela deve permanecer *true* depois que qualquer método público na classe for chamado. Durante a execução do código dentro de um método, a invariante de classe pode ser temporariamente rompida; no entanto, tal condição precisa ser reparada no instante em que o método é completado.

Por exemplo, reconsidere a classe `MyStack` na Figura 13.6, que criamos e expandimos na Figura 18.11. Essa nova versão tem uma invariante de classe, uma variável de instância privada adicional `n`, três novos métodos públicos e pré e pós-condições apropriadas acrescentadas a todos os métodos. A nova variável de instância privada `n` é uma contagem do número de elementos na pilha, e a variável de instância `theStack` é uma referência ao elemento mais próximo ao topo da pilha.

Por ora, vamos nos concentrar na especificação da invariante de classe, que tem a seguinte forma geral em JML:

```
public invariant expression ;
```

A invariante é verificada automaticamente pela JML cada vez que um construtor ou uma chamada de método é iniciado ou concluído, e a exceção `JMLInvariantError` é gerada sempre que a `expression` da variante não for *true* (verdadeira).

Considere a especificação JML para a invariante para a classe `myStack` na Figura 18.11:

```
  /*@ public model Node S;
    private represents S <- theStack;
    public invariant S == null || n == this.size();
  @*/
private /*@ spec_public @*/ Node theStack = null;
private /*@ spec_public @*/ int n = 0;
```

Aqui, usamos algo chamado *variável modelo* S, que só é conhecida nas especificações JML e não tem funcionalidade em tempo de execução dentro do programa Java. A finalidade de uma variável modelo é facilitar a escrita de especificações JML que sejam isentas de detalhes da implementação da classe.

Nosso uso da variável modelo S dispensa as especificações JML de mencionar explicitamente a variável `theStack` que é um nome que depende da implementação. A cláusula que inicia com `private represents` identifica a relação entre a variável modelo e a variável real que ela representa.

Outra cláusula JML que aparece aqui é a cláusula `spec_public`. Essa cláusula permite que uma variável `private` seja tratada como variável pública pelas próprias especificações JML. Assim, o valor `private` de uma variável pode ser acessado sempre que ele for mencionado por uma especificação JML, mas permanece inacessível a qualquer cliente da classe em tempo de execução.

18.3.3 Exemplo: Corretude de uma Aplicação Stack (Pilha)

Conforme sugerido acima, qualquer método em uma classe pode ser aumentado com pré e pós-condições que restringem o intervalo de valores que seus argumentos e resultados podem ter. Considere, por exemplo, o método `pop` que aparece na classe `MyStack` na Figura 13.6:

```
public int pop( ) {
        int result = theStack.val;
        theStack = theStack.next;
        return result;
}
```

Qual seria uma pré-condição apropriada para esse método? Isto é, o que seria necessário para que o método `pop` pudesse completar sua tarefa de forma bem-sucedida? No mínimo, não iríamos querer que um método `pop` fosse alguma vez chamado quando a pilha estivesse vazia. Ou seja, as referências no código acima a `theStack.val` e `theStack.next` somente têm significado se o valor de `theStack` não for `null` no instante da chamada. Assim, uma boa pré-condição para `pop` iria requerer que $n > 0$.

O que gostaríamos de ver para uma pós-condição? Informalmente, sabemos que o elemento superior da pilha é removido e retornado por essa operação, de forma que a pilha resultante pode se tornar vazia. Além disso, podemos esperar que a variável `n` seja decrementada de 1 para significar remoção do elemento de cima.

```
/*@ requires n > 0;
  ensures \result==\old(S).val &&
          S==\old(S).next && n==\old(n)-1;
@*/
public /*@ pure @*/ int pop( ) {
     int result = theStack.val;
     theStack = theStack.next;
     n = n-1;
     return result;
}
```

| Figura 18.10 Método `pop` de Pilha com Especificações Acrescentadas

Essas pré e pós-condições são formalizadas ao se acrescentar especificações JML a `pop`, como mostra a Figura 18.10. Lá, vemos um uso da função JML especial `\old` para especificar o valor *a priori* de `theStack` e `n` após entrada no método `pop`. Assim, o valor final de uma variável pode ser especificado como uma função de seu valor na entrada para o método.

A especificação JML `/*@ pure @*/` é usada para qualquer método que não tenha efeitos colaterais não locais e provavelmente não esteja em laço. Isto é, métodos puros anulam, de certa forma, a noção de corretude parcial discutida em uma seção anterior. Para que um método seja usado na especificação JML da invariante de classe, ele deve ser um método puro. Esse é o caso, por exemplo, dos métodos `push`, `pop` e `top` na classe `MyStack`.

Usando essas convenções, mostramos na Figura 18.11 um conjunto completo de especificações para os métodos da classe `MyStack`. Isso inclui pré e pós-condições para os métodos `push`, `pop`, `top`, `isEmpty` e `size`, a invariante de classe e o uso da variável modelo `S` em toda parte.

Testando o Contrato Anotando a classe `MyStack` com pré e pós-condições e uma invariante de classe temos um ambiente executável no qual o contrato entre a classe e seus clientes pode ser testado continuamente. Além disso, essas anotações proporcionam um mecanismo para atribuir culpa quando o contrato é rompido pela classe ou pelo seu cliente.

Para ilustrar essa atividade de teste, o programa mostrado na Figura 18.12 pode exercitar os métodos da classe `MyStack`. O comando a seguir pode ser usado para executar o programa.

```
% jmlrac myStackTest 4 5 6
Stack size = 3
Stack contents = 6 5 4
Is Stack empty? true
```

Os parâmetros fornecem os valores a serem colocados na pilha. A saída normal produzida por esse programa vem após o comando.

```
public class MyStack {
    private class Node {
        /*@ spec_public @*/ int val;
        /*@ spec_public @*/ Node next;
        Node(int v, Node n) {
            val = v; next = n;
        }
    }
    /*@ public model Node S;
        private represents S <- theStack;
        public invariant n == this.size();
    @*/
    private /*@ spec_public @*/ Node theStack = null;
    private /*@ spec_public @*/ int n = 0;
    /*@ requires n > 0;
        ensures \result==\old(S).val && S==\old(S).next;
    @*/
    public /*@ pure @*/ int pop( ) {
        int result = theStack.val;
        theStack = theStack.next;
        n = n-1;
        return result;
    }
    //@ ensures S.next==\old(S) && S.val==v;
    public /*@ pure @*/ void push(int v) {
        theStack = new Node(v, theStack);
        n = n+1;
   }
   /*@ requires n > 0;
       ensures \result==S.val && S == \old(S);
   @*/
   public /*@ pure @*/ int top() {
       return theStack.val;
   }
   //@ ensures \result == (S == null);
   public /*@ pure @*/ boolean isEmpty() {
       return theStack == null;
   }
   //@ ensures \result == n;
   public /*@ pure @*/ int size() {
       int count;
       Node p = theStack;
       for (count=0; p!=null; count++)
           p = p.next;
       return count;
   }
}
```

| Figura 18.11 Uma Classe Stack (Pilha) Totalmente Especificada Usando JML

```
public class myStackTest {
public static void main(String[] args) {
   MyStack s = new MyStack();
   int val;
   for (int i=0; i<args.length; i++)
      s.push(Integer.parseInt(args[i]));
   System.out.println("Stack size = " + s.size());
   System.out.print("Stack contents =");
   for (int i=1; i<=n; i++) {
      System.out.print(" " + s.top( ));
      s.pop( );
   }
   System.out.println( );
   System.out.println("Is Stack empty?" + s.isEmpty( ));
}
}
```

| Figura 18.12 Um Programa para Testar a Classe MyStack Especificada em JML

Para explorar os vários aspectos do contrato entre a classe e seu cliente, podemos fazer diversos testes. O primeiro teste, cujos resultados são mostrados abaixo, ilustra o que acontece quando o método `top` remove erroneamente o elemento superior bem como quando o retorna – isto é, age incorretamente como um método `pop`.

```
Exception in thread "main"
   org.jmlspecs.jmlrac.runtime.JMLNormalPostconditionError:
   by method MyStack.top regarding specifications at
File "MyStack.java", line 31, character 26 when
        '\old(S)' is MyStack$Node@5ff48b
        '\result' is 5
        'this' is MyStack@affc70
        at MyStack.checkPost$top$MyStack(MyStack.java:999)
        at MyStack.top(MyStack.java:1078)
        at myStackTest.main(MyStackTest.java:15)
```

Para disparar esse erro, acrescentamos duas linhas extras ao método `top` na Figura 18.11 de maneira que seu corpo ficasse parecido com aquele do método `pop`. Como o resultado do método `top` agora não satisfaz à sua pós-condição `S = \old(S)`, é gerado um erro `JMLNormalPostconditionError` e os valores de `\old(S)` e `\result` são relatados pela JML. Com essas informações, a culpa pelo erro pode ser atribuída ao método `top` e não ao seu chamador.

Para o segundo teste, excluímos a linha `n=n-1;` do método `pop` mostrado na Figura 18.11, criando assim uma situação em que a pós-condição do método é satisfeita, mas a invariante de classe é violada. Veja o resultado a seguir.

```
Stack size = 3
```

```
Stack contents = 6
Exception in thread "main"
   org.jmlspecs.jmlrac.runtime.JMLInvariantError:
   by method MyStack.pop@post<File "MyStack.java", line 16,
            character 17>
   regarding specifications at
File "MyStack.java", line 11, character 30 when
         'this' is MyStack@9664a1
         at MyStack.checkInv$instance$MyStack(MyStack.java:102)
         at MyStack.pop(MyStack.java:525)
         at myStackTest.main(MyStackTest.java:21)
```

Aqui, vemos que o programa começa a executar normalmente, mas, quando é completada a primeira chamada ao método `pop`, o teste JML da invariante de classe `n==this.size();` falha. Isso faz aparecer a mensagem `JMLInvariantError`, juntamente com algumas informações úteis para o rastreamento do programa. Para verificar se essa invariante foi satisfeita, a JML chama o método `this.size()`, que recalcula o tamanho da pilha independentemente percorrendo sua representação de lista vinculada.

O terceiro teste foi criado adicionando-se uma nova linha final `s.pop();` ao programa que controla o teste, para que ele tentasse fazer um pop em um elemento de uma pilha vazia. Aqui está o resultado.

```
Stack size = 3
Stack contents = 6 5 4
Is Stack empty? true
Exception in thread "main"
   org.jmlspecs.jmlrac.runtime.JMLEntryPreconditionError:
   by method MyStack.pop regarding specifications at
File "MyStack.java", line 16, character 21 when
         'this' is MyStack@9664a1
         at MyStack.checkPre$pop$MyStack(MyStack.java:330)
         at MyStack.pop(MyStack.java:479)
         at myStackTest.main(MyStacktest.java:24)
```

Essa execução relata um erro `JMLEntryPreconditionError`, indicando que a pré-condição `n > 0` para `pop` foi violada. Assim, a culpa por esse erro vai para o chamador e não para o método `pop`. A presença de uma pré-condição verificada ativamente protege o próprio método, evitando que ele entre em uma "programação defensiva", que seria necessária se a pré-condição não estivesse lá.

Corretude da Classe MyStack E quanto à corretude da classe `MyStack`? Comentá-la com pré e pós-condições e uma invariante de classe, e depois testar o contrato com um programa, certamente não garante a corretude em um sentido formal.

Informalmente, uma classe é *correta* se, para todo objeto na classe e todo construtor ou a chamada de método que satisfaz sua pré-condição, o término da chamada satisfizer suas pós-condições e deixar as variáveis de instância do objeto em um estado que satisfaça à invariante de classe. Essa idéia assume que nenhum construtor ou nenhuma chamada de método resultará em um laço infinito, e assim ela é uma afirmação sobre a corretude parcial.

Vamos experimentar uma definição mais formal de corretude de classe[9] usando a noção de triplas de Hoare que introduzimos no início do capítulo. Suponha que R representa uma invariante de uma classe e P_i e Q_i representam a pré-condição e pós-condição para seu i-ésimo construtor C_i ou método M_i. Então podemos dizer que uma classe é *correta* com relação às suas asserções se:

1 Para todo conjunto de argumentos válidos x para todo construtor C_i,
$\{ P_i(x) \}\ C_i\ .body\ \{ Q_i(x) \wedge INV \}$

2 Para todo conjunto de argumentos válidos x para todo método M_i,
$\{ P_i(x) \wedge INV \}\ M_i\ .body\ \{ Q_i(x) \wedge INV \}$.

A regra 1 diz, com efeito, que a execução de qualquer construtor para um objeto na classe deverá estabelecer a validade da invariante de classe. A regra 2 diz que a execução de qualquer chamada de método na qual a invariante de classe é válida no início deverá preservar a validade da invariante após o término da chamada. Assim, essa definição requer que provemos a corretude de todo construtor e método público individualmente. Cada uma dessas provas é conduzida de uma maneira similar àquela desenvolvida para o método `Factorial` na Seção 18.1.5.

Vamos ilustrar essas idéias desenvolvendo algumas das provas de que a implementação da nossa lista vinculada da classe `MyStack` está formalmente correta. Primeiro, observamos que a classe construtor padrão `MyStack()`estabelece os valores `theStack==null` e `n==0`. A própria invariante de classe *INV* é expressa em termos do método público `size`. A verificação do corpo do método `size` com essa invariante estabelecida deverá satisfazer à sua pós-condição e preservar a invariante. Formalmente, queremos provar:

$\{theStack = null \wedge n = 0\}$ size.body $\{theStack = null \wedge n = 0 \wedge \backslash result = n\}$

Informalmente, notamos que as variáveis locais `p` e `count` para `size` são inicializadas em `null` e 0, respectivamente. Assim, o laço não é executado de forma nenhuma e é retornado o resultado 0, estabelecendo a validade da tripla de Hoare acima. (Uma prova formal teria aplicado mais rigorosamente todos os passos do método de prova descrito na Seção 18.1.5; aqui, nós tomamos um atalho no processo para manter nosso foco principal na verificação da classe.)

Agora precisamos verificar todos os métodos na classe. Vamos ilustrar isso observando o método `pop`. Isto é, precisamos estabelecer a validade da seguinte tripla de Hoare:

$\{n > 0 \wedge n = this.size()\}$

pop.body

$\{\backslash result = \backslash old(S).val \wedge S = \backslash old(S).next \wedge n = this.size()\}$

A pré-condição `n > 0` estabelece que a pilha não pode estar vazia; isso é equivalente a requerer que `S` não seja `null`.

O conjunto de atribuições em `pop.body` leva diretamente à validade da pós-condição. Ou seja, o primeiro elemento é removido da lista vinculada e é retornada a lista resultante. Mais precisamente, observe que a pós-condição para `pop` especifica esse

9. Essa formalização é adaptada de Meyer, 1997.

efeito em termos da variável modelo `S`, que é uma substituta para a variável de instância `theStack`. Como `\old(S)` identifica o valor de `S` no início da chamada a `pop`, a expressão `S` $\equiv$ `\old(S).next` declara que a pilha resultante `S` é idêntica à pilha de entrada com o primeiro elemento removido. Paralelamente, o valor da variável de instância `n` é decrementado, de forma a preservar a validade da invariante $n = this.size$.

Finalmente, a corretude da classe `MyStack` depende implicitamente da suposição da corretude da classe `Node`. Em um ambiente de verificação formal, a classe `Node` precisaria ser especificada formalmente e também provada como correta.

18.3.4 Observações Finais

Discutimos o suficiente sobre os procedimentos de corretude formal para uma classe razoavelmente complexa, para concluir que a prova de corretude de qualquer programa razoavelmente grande é (na melhor das hipóteses) muito tediosa. Quais são as perspectivas para o uso eficaz dos métodos formais em projeto de software?

Primeiro, muitos acreditam que outras técnicas de projeto de software têm mais potencial para resolver a crise atual de software do que os métodos formais. Por exemplo, aquilo que chamam de capability maturity model (CMM) (Ahern et al., 2004) focaliza o refinamento dos processos de gerenciamento de software como chave para melhorar a qualidade do software.

Segundo, é verdade também que os métodos formais têm sido usados efetivamente para verificar componentes de produtos de software muito críticos no aspecto da segurança. Por exemplo, uma autoridade para certificação segura para *smart cards* foi desenvolvida pela Praxis Critical Systems (Hall e Chapman, 2002), usando-se métodos formais para estabelecer a corretude das propriedades críticas de segurança do sistema. Há muitos outros exemplos de uso efetivo dos métodos formais na prática de projeto de software.

Terceiro, muitos acreditam que o custo de desenvolvimento de um sistema que possa ser provado como correto é alto, em relação ao custo de se usar métodos tradicionais de teste e depuração. No entanto, a Praxis Critical Systems agora está desenvolvendo sistemas com uma quantidade de erros cem vezes menor, a um custo mais baixo do que o de software convencional, usando métodos formais. O maior problema é que a maioria dos programadores não está bem treinada no uso da lógica matemática para raciocinar sobre seus programas. Eles precisam ser treinados em lógica e no uso de ferramentas avançadas de software que os ajudem na verificação formal.

Resumindo, concluímos que a comunidade interessada em desenvolver melhores métodos formais para projeto de software ganhou um reforço fundamental nos últimos anos. Certamente, o uso de métodos formais por si só não é uma panacéia para a crise de software, porém ele possibilita uma dose de rigor para o processo de projeto de software que é cada vez mais necessária. Por essa simples razão, esperamos que mais ferramentas de linguagem de programação como JML, ESC/JAVA e LOOP continuem a evoluir e causem um impacto no processo de projeto de software no futuro.

18.4 CORRETUDE DE PROGRAMAS FUNCIONAIS

Esta seção trata da questão da corretude de programa da programação funcional. Vamos abordar a questão sobre como testar se um programa está correto para o caso especial em que ele é escrito em um programa funcional puro – um programa que é isento de estado e se baseia na composição funcional e na recursão como base para suas semânticas.

A Seção 18.4.1 ilustra esse processo, fazendo uma forte conexão entre uma função recursiva e uma prova indutiva de sua corretude. A Seção 18.4.2 contém três exemplos adicionais, o que dispensa atenção particular ao uso da indução estrutural – isso é, indução sobre estruturas de dados como listas e *strings*, em vez de inteiros.

18.4.1 Recursão e Indução

Ao considerar a questão da corretude para programas escritos em uma linguagem funcional pura, como Haskell, nos encontramos em uma situação bastante diferente. Primeiro, na ausência de noção de estado do programa e da atribuição, não precisamos escrever triplas de Hoare para manter o controle das transformações de estado como teríamos de fazer com programas escritos em linguagens imperativas orientadas a objeto.

Em vez disso, os programas funcionais são escritos como coleções de funções que são bem fundamentadas na matemática das funções e relações de recorrência. Isso nos permite basear as provas de corretude para funções Haskell na técnica bem conhecida de indução matemática, em vez de provas diretas que se baseiam no raciocínio sobre transformações de estado em todos os passos. Geralmente, a verificação de programas funcionais é um processo mais direto do que a verificação dos programas imperativos orientados a objeto.

Para um exemplo simples, considere a função Haskell que calcula o fatorial de um inteiro não-negativo `n`:

```
>   fact n
>         | n == 1 = 1                -- fact.1 (basis step)
>         | n > 1  = n*fact(n-1)     -- fact.2 (induction step)
```

Suponha que queremos provar que essa função calcula o produto dos primeiros n inteiros não-negativos, dado n. Isto é, queremos provar que:

$fact(1) = 1$

$fact(n) = 1 \times 2 \times \ldots \times (n-1) \times n$ quando $n > 1$

Para uma prova indutiva, lembre-se de que precisamos mostrar:

1. (Etapa Básica) Que a função calcula o resultado correto para $n = 1$.
2. (Etapa de Indução) Supondo a hipótese de que a função calcula o resultado correto para algum inteiro $n = k - 1$, podemos concluir que a função calcula o resultado correto para o próximo inteiro $n = k$.

Como a função `fact` é definida recursivamente, seus comandos condicionais (*guarded*) naturalmente delineiam a etapa básica a partir da etapa de indução, conforme está indicado pelos comentários à direita. Assim, a etapa básica é manipulada pela primeira linha da definição da função e a etapa de indução é manipulada pela segunda.

A definição da função satisfaz a etapa básica por observação. Isto é, quando $n = 1$ temos `fact(1) = 1`, usando a linha comentada `fact.1`.

Para a etapa de indução, suponha que $n > 1$ e $fact(n-1) = 1 \times 2 \times \ldots \times (n-1)$. Então a corretude é estabelecida para `fact(n)` usando a linha comentada `fact.2` e a hipótese, seguida de uma simplificação algébrica:

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

A partir desse exemplo particular, os leitores deverão notar a facilidade relativa com a qual a corretude de um programa em uma linguagem funcional pode ser provada em contraste com sua equivalente em uma linguagem imperativa. As triplas de Hoare e suas técnicas de prova são substituídas por um processo direto de indução no qual a definição da função espelha diretamente a prova.

18.4.2 Exemplos de Indução Estrutural

Uma estratégia de indução pode ser usada também para provar propriedades de funções Haskell que operam sobre listas e *strings*. A indução em funções de processamento de lista e processamento de *string*, muitas vezes, é chamada "indução estrutural" porque ela simplifica a estrutura (tamanho) de uma lista ou *string* à medida que define as hipóteses e mostra a validade da etapa de indução.

Esta seção fornece exemplos de provas de indução para várias funções Haskell que envolvem encadeamento de lista, reversão e comprimento. Como uma *string* Haskell é uma lista de caracteres, essas provas se aplicam a *strings* e também a listas.

Reversão de Lista e Encadeamento Considere as seguintes funções definidas para encadeamento de lista e reversão (elas espelham as funções Haskell padrão ++ e `reverse`, respectivamente):

```
> cat []     ys = ys                        -- cat.1
> cat (x:xs) ys = x : (cat xs ys)           -- cat.2
>
> rev []        = []                        -- rev.1
> rev (x:xs)    = cat (rev (xs)) [x]        -- rev.2
```

Suponha que tenhamos de provar a seguinte propriedade sobre a relação entre essas duas funções:

```
rev (cat xs ys) = cat (rev ys) (rev xs)
```

Por exemplo, se duas listas (strings) forem "hello" e "world", então a seguinte afirmação é verdadeira:

```
rev (cat "hello " "world")
    = cat (rev "world") (rev "hello ")
    = "dlrow olleh"
```

Para provar essa propriedade por indução, começamos com a etapa básica e usamos as definições dessas duas funções. Assim, precisamos primeiro mostrar que:

```
rev ([] ++ ys) = rev (ys) ++ rev ([])
```

Usando várias linhas nas definições dessas funções, provamos isso por substituição, da seguinte forma (as justificativas de cada etapa são mostradas à direita):

```
rev (cat [] ys) = rev (ys)                          (de cat.1)
                = cat (rev (ys) [])                 (de rev.2)
                = cat (rev (ys) rev [])             (de rev.1)
```

A hipótese de indução para essa prova é escrita formulando-se a conclusão para quaisquer duas listas xs e ys.

```
rev (cat xs ys) = cat reverse(ys) reverse(xs)
```

Agora a etapa de indução pode ser completada mostrando como uma lista ligeiramente mais longa (um elemento a mais) que a lista x:xs obedece à mesma regra, a seguir:

```
rev (cat (x:xs) ys) = cat (rev ys) (rev (x:xs))
```

Aqui, transformamos o lado esquerdo dessa expressão usando nossa hipótese e várias linhas nas definições das funções `rev` e `cat`, para obter o seguinte:

```
rev (cat (x:xs) ys) = rev (x : (cat xs ys))    (de cat.2)
     = rev (cat (cat xs ys) [x])               (de rev.2)
     = cat (cat (rev ys) (rev xs)) [x]         (de nossa hipótese)
     = cat (rev ys) (cat (rev xs) [x])         (associatividade de cat)
     = cat (rev ys) (rev (x:xs))               (de rev.2)
```

Finalmente, observe que a quarta linha nessa derivação assume associatividade para o operador `cat`, que pode ser provada separadamente por indução. Deixamos isso como exercício.

Comprimento da Lista e Concatenação Considere a seguinte função Haskell, que calcula explicitamente o comprimento de uma lista. Como isso é predefinido em Haskell como `length`, nós o redefinimos aqui com um nome ligeiramente diferente. Uma vez mais, os comentários à direita serão usados em provas sobre as propriedades dessa função.

```
> len []     = 0                          -- len.1
> len (x:xs) = 1 + (length xs)            -- len.2
```

Para essa função, a primeira linha define o tamanho, 0, de uma lista vazia, e a segunda mostra como calcular o tamanho de uma lista com base no tamanho de uma lista de tamanho conhecido ligeiramente menor do que ela. Por exemplo:

```
len [1,3,4,7]
   = 1 + len [3,4,7]
   = 1 + (1 + len [4,7])
   = 1 + (1 + (1 + len [7]))
   = 1 + (1 + (1 + (1 + len [])))
   = 1 + (1 + (1 + (1 + 0)))
   = 4
```

As quatro primeiras chamadas usam a segunda linha da função `len`, enquanto a quinta chamada usa a primeira linha.

Aqui está uma prova indutiva de que o tamanho das duas *strings* concatenadas é idêntico à soma de seus tamanhos individuais.

```
len (cat xs ys) = len xs + len ys
```

Observe, nessa prova, que é usado um padrão familiar: a etapa básica usa a primeira linha na definição recursiva, e a etapa de indução na prova usa a segunda linha. Essa prova fornece outro exemplo de indução estrutural.

Para a etapa básica, precisamos mostrar que:

```
len (cat [] ys) = len [] + len ys
```

Isso é feito pelas duas linhas a seguir:

```
len (cat [] ys) = len ys                    por cat.1
        = 0 + len ys                        por aritmética
        = len [] + len ys                   por len.1
```

Para a etapa indutiva, admitimos que a hipótese é verdadeira para *strings* arbitrárias xs e ys:

```
len (cat xs ys) = len xs + len ys for lists xs e ys.
```

Agora vamos ver o que acontece quando acrescentamos um caractere adicional à primeira *string*:

```
len (cat x:xs ys) = len x: (cat xs ys)      por cat.2
        = 1 + len (cat xs ys)               por len.2
        = 1 + len xs + len ys               por hipótese
        = len x:xs + len ys                 por len.2
```

Isso completa a prova.

Conforme ilustram esses exemplos, Haskell proporciona um suporte especialmente forte para provas de corretude. Embora, infelizmente, não haja um grande número de sistemas de software implementados em Haskell, aqueles que o são desfrutam um alto nível geral de confiabilidade.

No entanto, linguagens funcionais como Haskell estão sendo consideradas cada vez mais seriamente pelos projetistas de software como veículos para definir especificações precisas para protótipos de software. Linguagens convencionais como C++ e Ada têm sido inadequadas para essa finalidade (Hudak, 2000).

18.5 RESUMO

Este capítulo considera a corretude de programas e seu impacto sobre o projeto de linguagens de programação. A corretude de programas imperativos e orientados a objetos é baseada em semânticas axiomáticas.

O uso de ferramentas para especificar e verificar a corretude de programas é ilustrado por intermédio da Java Modeling Language (JML). É explicada também a conexão entre JML e o conceito de projeto por contrato. Outras linguagens também suportam métodos formais em programação, como Spark/Ada e Haskell.

Um tratamento completo sobre corretude de programas ocorreria mais naturalmente em um curso de projeto de software. Este capítulo somente introduz a idéia, focalizando os fundamentos teóricos, o projeto básico de linguagens e os problemas de programação.

EXERCÍCIOS

18.1 Sugira uma maneira diferente de escrever a função `Max(a, b)` na Figura 18.1 sem mudar o sentido da função.

18.2 Abaixo está uma tripla de Hoare que inclui um fragmento de programa similar a C, para calcular o produto `z` de dois inteiros `x` e `y`.

$\{y \geq 0\}$
$z = 0;$
n = $y;$
while (n $>$ 0)
$z = z + x;$
n = n– –1;
od
$\{z = x \times y\}$

(a) Quais regras de inferência da Tabela 18.1 e que conhecimento adicional sobre álgebra podem ser usados para inferir que a pré-condição nesta tripla de Hoare é equivalente à asserção $\{y \geq 0 \wedge 0 = x(y-y)\}$?

(b) Usando a regra de inferência de atribuição, complete a seguinte tripla de Hoare para as primeiras duas instruções neste trecho de programa:

$\{y \geq 0 \wedge 0 = x(y - y)\}$
$z = 0;$
n = $y;$
$\{y \geq 0 \wedge$ $\}$

(c) Explique como a linha a seguir pode ser uma invariante para o laço while neste trecho de programa.

$\{y \geq 0 \wedge n \geq 0 \wedge z = x(y - n)\}$

Isto é, por que essa asserção é *true* (verdadeira) antes da execução da primeira instrução no laço, e por que ela deve ser *true* antes da execução de qualquer repetição sucessiva do laço?

(d) Mostre que essa invariante vale para uma única passagem por meio das instruções do laço.

(e) Usando a regra de inferência para laços, mostre como a invariante é resolvida para uma asserção que implica a validade da pós-condição para o programa inteiro.

18.3 A função C++ a seguir calcula a raiz quadrada aproximada de `a` com um pequeno erro `epsilon`, usando o método de Newton.

```
float mySqrt (float a, float epsilon) {
    float x;
    x = 1.0;
    while (x*x > a+epsilon || x*x < a-epsilon)
        x = (x + a/x)/2.0;
    return x;
}
```

(a) Descreva a pré-condição e a pós-condição, P e Q, que serviriam como especificações formais apropriadas para essa função.
(b) Descreva a invariante do laço que serviria para descrever o laço nessa função.
(c) Há algumas circunstâncias especiais sob as quais uma chamada a essa função não terminará ou não satisfará à sua pós-condição? Explique.
(d) Prove a corretude (parcial) dessa função.

18.4 Suponha que a função no Exercício 18.3 fosse parte de uma classe Java que suporta funções matemáticas. Descreva um contrato que seja apropriado para qualquer cliente daquela classe com relação ao seu uso da função `mySqrt`. Forneça cláusulas JML `requires` e `ensures` que especifiquem esse contrato.

18.5 Escreva uma função Java que calcule a soma de uma série de inteiros armazenados como uma lista encadeada (similar às classes `MyStack` e `Node` discutidas neste capítulo). Escreva as pré e pós-condições para essa função e depois desenvolva uma prova de sua corretude usando as regras de inferência da Tabela 18.1.

18.6 No sentido de projetar software a partir de especificações formais, encontre uma definição precisa na língua inglesa no site Java para o método `indexOf(String)`na classe `java.lang.String`.
(a) Traduza aquela definição para uma pré e pós-condição formal.
(b) Agora traduza a sua especificação para cláusulas JML `requires` e `ensures`.

18.7 Forneça uma implementação recursiva C/C++ da função `Factorial` na Figura 18.3. Prove a corretude parcial da sua implementação recursiva para todos os valores de $n > 0$. *Nota*: para provar a corretude de uma função recursiva, deve ser usada a indução. Isto é, o caso-base e a chamada recursiva na definição de função correspondem à etapa básica e à etapa de indução na prova.

18.8 Um programa tem *corretude total* se ele (completa sua execução e) satisfaz à sua pós-condição para *todos* os valores de entrada especificados em sua pré-condição. Suponha que alteremos a função `Factorial` na Figura 18.3 de forma que o tipo de seu argumento e o resultado sejam `long` em vez de `int`.
(a) Experimentalmente, determine o maior valor de `n` para o qual a sua versão alterada da função `Factorial` fornecerá um resultado. O que acontece quando ela não fornece um resultado?
(b) Refine a pré-condição para essa versão da função `Factorial` de forma que sua prova de corretude se torne a prova de corretude total.
(c) Como a própria prova de corretude é alterada por essas mudanças, se houver? Explique.

18.9 Altere a versão JML da definição da função `Factorial` na Figura 18.8 de maneira que os tipos de seu argumento e o resultado sejam `long` em vez de `int`. Acrescente recursos de geração de exceção a essa função de forma que ela gere uma exceção `ArithmeticError` sempre que o fatorial não puder ser calculado corretamente. Finalmente, acrescente uma cláusula JML `signals` à especificação que abranja esse evento.

18.10 Reimplemente a função `Factorial` de forma que ela retorne um valor do tipo `BigInteger`. De que maneiras essa implementação é melhor do que a versão apresentada na Figura 18.8?

18.11 Reimplemente a função `Factorial` em Haskell. De que maneiras essa implementação é melhor do que a versão apresentada na Figura 18.8 e Exercício 18.10? De que maneiras ela é inferior?

18.12 Forneça uma prova de indução para a corretude da sua implementação Haskell da função `Factorial` no Exercício 18.11. Para isso, você deverá se basear na definição matemática de fatorial.

18.13 Discuta as possibilidades que existem entre as escolhas de se refinar a pré-condição e adicionar uma cláusula `signals` ao especificar a resposta de uma função a um valor de entrada para o qual ela não pode calcular um resultado significativo. Por exemplo, essas escolhas estão ilustradas nos Exercícios 18.8 e 18.9.

18.14 Considere a corretude da classe `MyStack` na Seção 18.3.

(a) O método `size` foi verificado em conjunto com a inicialização de um objeto usando o construtor de classe. Escreva uma tripla de Hoare apropriada e então verifique o método `size` para todos os outros estados que o objeto pode ter.

(b) Escreva uma tripla de Hoare apropriada e depois verifique o método `push`.

18.15 Prove por indução que o operador ++ Haskell é associativo, usando sua definição recursiva denominada `cat` que é dada neste capítulo. Mostre que para todas as listas xs, ys e zs:

```
cat (cat xs ys) zs = cat xs (cat ys zs)
```

Ou seja, isso é equivalente a `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`.

18.16 Dada a definição da função Haskell `len` neste capítulo, prove por indução o seguinte:

```
len (reverse xs) = len xs
```

18.17 Considere a seguinte implementação Haskell (correta, mas ineficiente) da função familiar Fibonacci:

```
> fibSlow n
>    | n == 0  = 1                  -- fib.1
>    | n == 1  = 1                  -- fib.2
>    | otherwise  = fibSlow(n-1) + fibSlow(n-2)      -- fib.3
```

A corretude dessa função pode ser provada rapidamente, já que ela é uma transcrição direta da definição matemática familiar abaixo e porque o tipo Haskell Integer é uma série infinita:

$$fib(0) = 1$$
$$fib(1) = 1$$
$$fib(n) = fib(n-1) + fib(n-2) \quad \text{if } n \geq 2$$

Dê uma prova por indução da corretude de `fibSlow`.

18.18 Conforme sugere seu nome, a eficiência da função `fibSlow` do Exercício 18.17 é suspeita.

(a) Experimente rodar `fibSlow(25)` e depois `fibSlow(50)` no seu sistema e veja quanto tempo esses cálculos demoram. O que causa essa ineficiência?

(b) Uma definição alternativa do cálculo Fibonacci pode ser enunciada da seguinte maneira: defina uma função `fibPair` que gera um conjunto de dois elementos contendo o n-ésimo número de Fibonacci e seu sucessor.

Defina outra função `fibNext` que gere a próxima tupla a partir da atual. Então, a própria função Fibonacci, chamada de forma otimista como `fibFast`, pode ser definida selecionando-se o primeiro membro do n-ésimo par `fibPair`. Em Haskell, isso é escrito como:

```
> fibPair n
>    | n == 0    = (1,1)
>    | n > 0     = fibNext(fibPair(n-1))
> fibNext (m,n)  = (n,m+n)
> fibFast n = fst(fibPair(n))
```

Experimente executar a função `fibFast` para calcular o 25º e o 50º números de Fibonacci. Ela deverá ser muito mais eficiente do que a função `fibSlow`. Explique.

(c) Prove por indução que $\forall n \geq 0 : \mathit{fibFast}(n) = \mathit{fibSlow}(n)$.

Apêndice A

VISÃO GERAL DO CAPÍTULO

Este apêndice fornece uma definição completa da sintaxe, do sistema de tipos e das semânticas de Clite. As quatro primeiras seções definem, respectivamente, a sintaxe léxica e concreta, a sintaxe abstrata, o sistema de tipos e as semânticas do núcleo da linguagem Clite. Essas são as definições apresentadas inicialmente nos Capítulos 2, 6, 8, 11. A quinta seção suplementa aquela definição de núcleo de linguagem, acrescentando variáveis globais e funções a Clite, conforme foram introduzidas no Capítulo 10.

Este apêndice serve como uma referência cruzada rápida para a definição precisa das características Clite à medida que os leitores trabalharem nos vários exercícios ao longo do texto. Para discussões mais detalhadas de qualquer característica particular, aconselhamos os leitores a consultar novamente o capítulo em que o assunto foi introduzido inicialmente.

A.1 SINTAXE LÉXICA E CONCRETA DE CLITE

Programa → `int main ( )` { *Instruções de Declaração* }
Declarações → { *Declaração* }
Declaração → *Identificador de Tipo* [`[` *Inteiro* `]`] { , *Identificador* [`[` *Inteiro* `]`] };
Tipo → `int` | `bool` | `float` | `char`
Instruções → { *Instrução* }
Instrução → ; | *Bloco* | *Atribuição* | *Instrução If* | *Instrução While*
Bloco → { *Instruções* }
Atribuição → *Identificador* [`[` *Expressão* `]`] = *Expressão*;
Instrução If → `if` (*Expressão*) *Instrução* [`else` *Instrução*]
Instrução While → `while` (*Expressão*) *Instrução*

Expressão → *Conjunção* { || *Conjunção* }
Conjunção → *Igualdade* { && *Igualdade* }
Igualdade → *Relação* [*Relação EquOp*]
EquOp → == | !=
Relação → *Adição* [*Adição RelOp*]
RelOp → < | <= | > | >=
Adição → *Termo* { *Termo AddOp* }
AddOp → + | −
Termo → *Fator* { *Fator MulOp* }
MulOp → * | / | %
Fator → [*UnárioOp*] *Primário*
UnárioOp → − | !
Primário → *Identificador* [[*Expressão*]] | *Literal* | (*Expressão*)
| *Tipo* (*Expressão*)

Identificador → *Letra* { *Letra* | *Dígito* }
Letra → `a` | `b` | . . . | `z` | `A` | `B` | . . . | `Z`
Dígito → `0` | `1` | . . . | `9`
Literal → *Inteiro* | *Boleano* | *Float* | *Char*
Inteiro → *Dígito* { *Dígito* }
Boleano → `true` | `false`
Float → *Inteiro* . *Inteiro*
Char → ‘ *ASCIIChar* ’

A.2 SINTAXE ABSTRATA DE CLITE

Programa = *Declarações* `decpart`; *Instruções* `body`;
Declarações = *Declaração**
Declaração = *VariávelDecl* | *ArrayDecl*
VariávelDecl = *Variável* `v`; *Tipo* `t`
ArrayDecl = *Variável* `v`; *Tipo* `t`; *Inteiro* `size`
Tipo = `int` | `bool` | `float` | `char`
Instruções = *Instrução**
Instrução = *Skip* | *Bloco* | *Atribuição* | *Condicional* | *Laço*
Skip =
Bloco = *Instruções*
Condicional = *Expressão* `test`; *Instrução* `thenbranch`, `elsebranch`
Laço = *Expressão* `test`; *Instrução* `body`
Atribuição = *VariávelRef* `target`; *Expressão* `source`
Expressão = *VariávelRef* | *Valor* | *Binário* | *Unário*
VariávelRef = *Variável* | *ArrayRef*
Binário = *BinárioOp* `op`; *Expressão* `term1`, `term2`
Unário = *UnárioOp* `op`; *Expressão* `term`
Operador = *BoleanoOp* | *RelacionalOp* | *AritméticoOp* | *UnárioOp*
BoleanoOp = `&&` | ||
RelacionalOp = = | != | < | <= | > | >=
AritméticoOp = + | − | * | /
UnárioOp = ! | −
Variável = *String* `id`
ArrayRef = *String* `id`; *Expressão* `index`
Valor = *IntValor* | *BoolValor* | *FloatValor* | *CharValor*
IntValor = *Inteiro* `intValue`
FloatValor = *Float* `floatValue`
BoolValor = *Boleano* `boolValue`
CharValor = *Caractere* `charValue`

A.3 SISTEMA DE TIPOS DE CLITE

Regra de Tipo A.1 *Todas as variáveis referenciadas devem ser declaradas.*

Regra de Tipo A.2 *Todas as variáveis declaradas devem ter nomes únicos.*

Regra de Tipo A.3 *Um Programa é válido em tipos se suas Declarações* `decpart` *forem válidas e seu Bloco* `body` *for válido em relação ao mapa de tipos para aquelas Declarações particulares.*

Regra de Tipo A.4 *Uma Instrução (Statement) é válida em relação ao mapa de tipos do programa se ela satisfaz às seguintes restrições:*

1. *Um Skip é sempre válido.*
2. *Uma Atribuição (Assignment) é válida se todos os itens a seguir forem verdadeiros (true):*
   (a) *Sua Variável* `target` *é declarada.*
   (b) *Sua Expressão* `source` *é válida.*
   (c) *Se o tipo de sua Variável* `target` *for* `float`*, então o tipo de sua Expressão* `source` *deve ser* `float` *ou* `int`.
   (d) *Ou então, se o tipo de sua Variável* `target` *for* `int`*, o tipo de sua Expressão* `source` *deve ser* `int` *ou* `char`.
   (e) *Ou então, o tipo de sua Variável* `target` *deve ser o mesmo que o tipo de sua Expressão* `source`.
3. *Uma Condicional é válida se a sua Expressão* `test` *for válida e tiver o tipo* `bool` *e suas Instruções (Statements)* `thenbranch` *e* `elsebranch` *forem ambas válidas.*
4. *Um Laço (Loop) é válido se a sua Expressão* `test` *for válida e tiver o tipo* `bool` *e sua Instrução (Statement)* `body` *for válida.*
5. *Um Bloco (Block) é válido se todas as suas Instruções (Statements) forem válidas.*

Regra de Tipo A.5 *A validade de uma Expressão (Expression) é definida usando o mapa de tipos do programa e cada uma das subclasses da Expressão:*

1. *Um Valor (Value) é válido.*
2. *Uma Variável é válida se o seu* `id` *aparecer no mapa de tipos.*
3. *Um Binário (Binary) é válido se todos os itens a seguir forem verdadeiros (true):*
   (a) *Suas Expressões* `term1` *e* `term2` *são válidas.*
   (b) *Se o seu Operador Binário (BinaryOp)* `op` *for aritmético (*`+`, `-`, `*`, `/`*), então ambas as suas Expressões (Expressions) devem ser* `int` *ou* `float`.
   (c) *Se* `op` *for relacional (*`==`, `!=`, `<`, `<=`, `>`, `>=`*), então ambas as suas Expressões devem ter o mesmo tipo.*
   (d) *Se* `op` *for boleano (*&&, ||*), então ambas as suas Expressões devem ser* `bool`.
4. *Um Unário (Unary) é válido se todos os itens a seguir forem verdadeiros (true):*
   (a) *Seu* `term` *da Expressão for válido.*
   (b) *Se o seu Operador Unário (UnaryOp)* `op` *for* `!`*, então* `term` *deve ser* `bool`.
   (c) *Se* `op` *for* −*, então* `term` *deve ser* `int` *ou* `float`.
   (d) *Se* `op` *é a conversão de tipo* `float()` *ou* `char()`*, então* `term` *deve ser* `int`.
   (e) *Se* `op` *é a conversão de tipo* `int()`*, então* `term` *deve ser* `float` *ou* `char`.

Regra de Tipo A.6 *O tipo do resultado de toda Expressão é determinado da seguinte forma:*

1. *Se a Expressão for um Valor, então o tipo de seu resultado é o mesmo tipo do Valor.*
2. *Se a Expressão for uma Variável, então o tipo de seu resultado é o mesmo tipo da Variável.*
3. *Se a Expressão for um Binário, então:*
   (a) *Se o Operador for aritmético* (+,−, *, ou */*), então seus termos devem ser do mesmo tipo e o tipo do resultado é o tipo de seus operandos. Por exemplo, a Expressão* `x+1` *requer que* `x` *seja* `int` *(já que* `1` *é* `int`*), assim o tipo de seu resultado é* `int`.

(b) *Se o Operador for relacional* (<,<=,>,>=,==, !=) *ou boleano* (&&, ||), *então o tipo de seu resultado é* `bool`.

4 *Se a Expressão for um Unário, então:*
(a) *Se o Operador for* `!`, *então o tipo de seu resultado é* `bool`.
(b) *Se o Operador for* `-`, *então o tipo de seu resultado é o tipo de seu operando.*
(c) *Se o Operador for uma conversão de tipo, então o tipo de seu resultado é dado pela conversão.*

A.4 SEMÂNTICAS DE CLITE

Regra de Significado A.1 *O significado de um Programa é definido como sendo o significado de seu* `body` *quando é dado um estado inicial, consistindo de variáveis do* `decpart`, *cada uma das quais inicializada com o valor undef correspondendo ao seu tipo declarado.*

Regra de Significado A.2 *O significado de uma instrução Skip é uma função identidade no estado; isto é, o estado é inalterado.*

Regra de Significado A.3 *O significado de uma instrução de Atribuição é o resultado obtido quando se substitui o valor da Variável* `target` *pelo valor da Expressão* `source` *no estado atual.*

Regra de Significado A.4 *O significado de uma Condicional depende da verdade ou falsidade de seu* `test` *boleano no estado atual. Se* `test` *for verdadeiro (true), então o significado da Condicional é o significado da Instrução (Statement)* `thenbranch`*; ou, então, ele é o significado da Instrução (Statement)* `elsebranch`.

Regra de Significado A.5 *Se o* `test` *for falso, o significado (estado de saída) de um Laço (Loop) é o mesmo do estado de entrada. Ou, então, o significado é o resultado obtido quando se aplica essa regra novamente ao significado de seu* `body` *no estado atual.*

Regra de Significado A.6 *O significado de um Bloco (Block) é o significado agregado de suas instruções (statements) quando aplicado ao estado atual. Se um Bloco não tem instruções, o estado não é alterado. Caso contrário, o estado resultante do significado da primeira Instrução (Statement) no Bloco torna-se a base para definir o significado do resto do bloco.*

Regra de Significado A.7 *O significado de uma Expressão (Expression) em um estado é um Valor (Value) definido da seguinte maneira:*

1 *Se a Expressão for um Valor, então seu significado é o significado do próprio Valor.*
2 *Se a Expressão for uma Variável, então seu significado é o Valor da Variável no estado atual.*
3 *Se a Expressão for um Binário, então o significado de cada um de seus operandos* `term1` *e* `term2` *é determinado primeiro. Então a Regra A.8 determina o significado da expressão aplicando o operador* `op` *aos Valores daqueles dois operandos.*
4 *Se a Expressão for um Unário, então o significado de seu operando* `term` *é determinado. Então a Regra A.9 determina o significado da expressão aplicando o operador* `op` *ao Valor do operando.*

Regra de Significado A.8 *O significado de uma expressão Binária é um Valor definido da seguinte maneira:*

1. *Se o operando* `term1` *ou* `term2` *for indefinido, o programa é semanticamente sem significado.*
2. *Se o operador for um operador aritmético inteiro, então um* `int+`, `int−`, `int*` *executa uma adição, subtração ou multiplicação inteira sobre seus operandos inteiros, dando um resultado inteiro. Se o operador for* `int/`*, então o resultado é o mesmo que o de uma divisão matemática com os resultados truncados (elimina-se a parte fracionária).*
3. *Se o operador for um operador de ponto flutuante, então é executada a aritmética de ponto flutuante usando o padrão IEEE sobre os operandos* `float`*, dando um resultado* `float`.
4. *Se o operador for um operador relacional, então os operandos são comparados, dando como resultado* `true` *ou* `false`*. O resultado é o mesmo que o resultado matemático correspondente, exceto que* `false` $<$ `true`.
5. *Se o operador for um operador boleano, então:*
   (a) *O operador* && *é interpretado como:*

   a && $b \equiv$ if a then b else *false*

   (b) *O operador* || *é interpretado como:*

   a || $b \equiv$ if a then *true* else b

Regra de Significado A.9 *O significado de uma expressão Unária é um Valor definido da seguinte forma:*

1. *Se o operando* `term` *for indefinido, o significado da expressão é indefinido.*
2. *Se o operador for* `!` *(not), então o operando boleano é invertido.*
3. *Se o operador for* `int−`/`float−`*, então o sinal do operando inteiro ou do ponto flutuante é invertido.*
4. *Se o operador for* `i2f`*, então o operando inteiro é convertido em ponto flutuante.*
5. *Se o operador for* `c2i`*, então o operando* `char` *é convertido para inteiro usando o código ASCII para o caractere. Efetivamente, exceto quanto à adição de bits de zeros anteriores, o valor fica inalterado, exceto quanto ao seu tipo.*
6. *Se o operador for* `i2c`*, então o operando inteiro é convertido em caractere. Se o operando inteiro for maior do que 255 ou menor do que zero, o significado da expressão é indefinido. Se a conversão for bem-sucedida, então, exceto quanto à perda dos bits dos zeros anteriores, o padrão de bits do valor é o mesmo; somente o tipo é alterado.*
7. *Se o operador for* `f2i`*, então o valor em ponto flutuante é convertido para inteiro, descartando os dígitos após o ponto decimal, isto é, eles são truncados. Se a parte inteira do valor em ponto flutuante for muito grande para armazenar em um inteiro, o significado da expressão é indefinido.*

As próximas três regras definem os elementos das Semânticas de Clite que incluem conjuntos dinâmicos. Elas utilizam as funções *new* e *delete* introduzidas no Capítulo 11.

Regra de Significado A.10 *O significado de uma ArrayDecl* `ad` *é:*

1. *Compute addr (*`ad[0]`*) = new (*`ad.size`*), em que o valor da Expressão* `ad.size` *é computado conforme descrito no Capítulo 7.*
2. *Push addr (*`ad[0]`*) para a pilha.*
3. *Push* `ad.size` *para a pilha.*
4. *Push* `ad.type` *para a pilha.*

Regra de Significado A.11 *O significado de uma ArrayRef* `ar` *é:*

1. *Compute addr* (`ad[ar.index]`) = *addr* (`ad[0]`) + `ad.index` − 1.
2. *If addr* (`ad[0]`) ≤ *addr* (`ad[ar.index]`) < *addr*(`ad[0]`) + `ad.size`, *retorna o valor em addr (*`ad[ar.index]`*).*
3. *Ou, então, mostra um erro índice-fora-do-range (ou limite).*

O significado de uma *Atribuição* `as` cujo `target` é uma *ArrayReference* `ar` é:

Regra de Significado A.12 *O significado de uma Atribuição* `as` *é:*

1. *Compute addr* (`ad[ar.index]`) = *addr* (`ad[0]`) + `ad.index` − 1.
2. *If addr* (`ad[0]`) ≤ *addr*(`ad[ar.index]`) < *addr* (`ad[0]`) + `ad.size`, *então muda o valor em addr* (`ad[ar.index]`) *para o valor de* `as.source`.
3. *Ou, então, mostra um erro índice-fora-do-range.*

A.5 ACRESCENTANDO FUNÇÕES À CLITE

A.5.1 Sintaxe Léxica e Concreta

Nesta definição, o sublinhado indica adições às regras originais para Clite dadas na Seção A.1.

Programa → { *Identificador de Tipo FunctionOrGlobal* } *MainFunction*
Tipo → `int` | `boolean` | `float` | `char` | `void`
FunçãoOuGlobal → (*Parâmetros*) { *Declarações Instruções (Statements)* } | *Global*
Parâmetros → [*Parâmetro* { , *Parâmetro* }]
Parâmetro → *Identificador de Tipo*
Global → { , *Identificador* } ;
MainFunction → `int main ( )` { *Declarações Instruções (Statements)* }
Instrução (Statement) → ; | *Bloco* | *Atribuição* | *InstruçãoIf*
InstruçãoWhile | *InstruçãoCall* | *InstruçãoReturn*
InstruçãoCall → *Call*;
InstruçãoReturn → *Expressãoreturn*;
Factor → *Identificador* | *Literal* | (*Expressão*) | *Call*
Call → *Identificador* (*Argumentos*)
Argumentos → [*Expressão* { , *Expressão* }]

Assim, a *MainFunction* é idêntica à noção original de *Programa*, exceto pelo fato de agora a *MainFunction* poder ser precedida por qualquer número de funções e/ou declarações globais de variável para formar um *Programa* completo.

A.5.2 Sintaxe Abstrata

Nesta definição, o sublinhado indica adições às regras originais para Clite dadas na Seção A.2.

Programa = *Declarações* globals; funções *Functions*
Funções = *Função**
Função = *Tipo* t; *String* id; *Declarações* params, locals; *Bloco* body
Tipo = int | boolean | float | char | void
Instrução = *Skip* | *Bloco* | *Atribuição* | *Condicional* | *Laço* | *Chamada* | *Retorno*
Chamada = Nome *String*; args *Expressões*
Expressões = *Expressão**
Retorno = *Variável* target; *Expressão* result
Expressão = *Variável* | *Valor* | *Binário* | *Unário* | *Chamada*

A.5.3 Sistema de Tipos

Regra de Tipo A.7 *Toda função e* `id` *global devem ser únicos.*

Regra de Tipo A.8 *Todos os* `params` *e* `locals` *de Função devem ser mutuamente únicos* `id`*'s.*

Regra de Tipo A.9 *Toda Instrução (Statement) no* `body` *de cada função deve ser válida com relação às variáveis locais da função, dos parâmetros e dos globais visíveis.*

Regra de Tipo A.10 *Uma instrução Return deve aparecer no* `body` *de toda função não-vazia exceto* `main`*, e sua Expressão deve ter o mesmo Tipo daquela função.*

Regra de Tipo A.11 *Nenhuma instrução Return pode aparecer em uma função* `void`.

Regra de Tipo A.12 *Todo* `name` *de uma Instrução Call deve identificar o* `id` *de uma Função vazia no programa, e toda Expressão Call deve identificar uma Função não-vazia.*

Regra de Tipo A.13 *Toda Chamada deve ter um número de* `args` *igual ao número de* `params` *na Função identificada, e cada um desses* `args` *deve ter o mesmo tipo de seu* `param` *correspondente, se for lido da esquerda para a direita.*

Regra de Tipo A.14 *O tipo de uma Chamada é o tipo da função identificada, e a expressão na qual ela aparece deve ser válida de acordo com as Regras de Tipos A.5 e A.6.*

A.5.4 Semântica

Regra de Significado A.13 *O significado de uma chamada (Call) c a uma função (Function) f tem os seguintes passos:*

1. *Cria um novo registro de ativação, acrescenta a ele os* `params` *de f e* `locals`.
2. *Avalia cada um dos* `args` *de c e atribui seu Valor ao* `param` *correspondente de f no registro de ativação.*
3. *Se a função for não-vazia, acrescenta ao registro de ativação uma variável de resultado idêntica ao nome e ao tipo da função.*
4. *Coloca o registro de ativação na pilha de tempo de execução.*
5. *Interpreta as instruções no* `body` *de f.*
6. *Tira o registro de ativação da pilha de tempo de execução.*
7. *Se a função for não-vazia, retorna o valor da variável do resultado para a chamada.*

Regra de Significado A.14 *O significado de um Return é computado substituindo-se o valor da Variável* `result` *(o nome da função chamada) no registro ativado pelo valor da Expressão* `result`.

Regra de Significado A.15 *O significado de um Bloco é o significado agregado de suas instruções quando aplicado ao estado corrente, e até incluindo o ponto onde é encontrado o primeiro Return. Se houver Blocos aninhados, o primeiro Return encontrado assinala o fim do Bloco mais externo. Isto é, o primeiro Return encontrado termina o corpo da função.*

Apêndice B

Revisão Matemática Discreta

VISÃO GERAL DO CAPÍTULO

A finalidade deste apêndice é fazer uma rápida revisão de certos tópicos importantes em matemática, necessários em um estudo formal das linguagens de programação.

B.1 SÉRIES E RELAÇÕES

As séries constituem a estrutura de dados mais fundamental da matemática discreta. Elas estão em toda parte na teoria da ciência da computação.

> **Definição:** Uma *série* é qualquer coleção de objetos; usualmente, todos os objetos em uma série têm o mesmo tipo.

Uma série pode conter números, *strings*, livros, teoremas, outras séries etc.

Uma maneira de descrever uma série é enumerar seus elementos. Veja um exemplo da série de vogais:

$\{a, e, i, o, u\}$

O uso de chaves para conter os números de uma série é uma convenção comum. As reticências (...) são usadas muitas vezes para suprimir elementos de uma série quando a seqüência é bem conhecida; um exemplo é a série de letras minúsculas do alfabeto:

$\{a, \ldots, z\}$

A eliminação de elementos de uma série também é comum quando a série é de tamanho infinito; um exemplo disso é a série de inteiros **I**:

$$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$$

Uma das operações fundamentais em uma série é o teste de participação para saber se o elemento pertence à série. Se A é uma série, podemos perguntar se x é um membro de A escrevendo $x \in A$. Por exemplo, a declaração:

$$a \in \{a, e, i, o, u\}$$

é verdadeira (true). O teste oposto, dizendo que x não é um membro de A, é escrito como $x \notin A$; por exemplo,

$$z \notin \{a, e, i, o, u\}$$

também é verdadeira.

O teste de participação, muitas vezes, é usado em conjunto com uma condição de restrição para definir uma série. Por exemplo, uma maneira de definir a série **N** de inteiros não-negativos é:

$$\{n \in \mathbf{I} \mid n \geq 0\}$$

em que **I** é a série de inteiros definida acima. Essa definição pode ser lida como a série de todos os inteiros n para os quais n é maior ou igual a zero. A barra vertical é lida como *tal que*, e ela introduz uma ou mais condições de restrição. Naturalmente, a série **N** de inteiros não-negativos poderia da mesma forma ser definida como:

$$\{0, 1, 2, \ldots\}$$

Considere a definição da série de números pares não-negativos como:

$$\{n \in \mathbf{N} \mid n \bmod 2 = 0\}$$

que diz que a série de números pares é a série dos inteiros não-negativos que são divisíveis exatamente por 2.

Todas as séries têm duas propriedades fundamentais. Primeira, nenhuma série contém elementos duplicados:

$$\{a, b, c\} = \{a, a, a, b, b, c\}$$

Isto é, as notações acima descrevem a mesma série. A segunda propriedade é que as séries são coleções não-ordenadas:

$$\{a, b, c\} = \{c, b, a\}$$

Portanto, duas séries são *iguais* se elas tiverem exatamente os mesmos elementos, independentemente da ordem em que eles aparecem.

As séries a seguir recebem nomes especiais devido à sua importância fundamental na matemática e na ciência da computação.

Notação	Significado	Explicação
R	Números reais	
I	Inteiros	Abreviados como $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
N	Números naturais	Abreviados como $\{0, 1, 2, \ldots\}$
B	Constantes boleanas	Abreviados como {*true*, *false*}

Muitas vezes, nos interessam as relações entre duas séries. Dizemos que a série A é um *subconjunto* da série B, que escrevemos como $A \subseteq B$, se cada elemento de A for também um elemento de B. Por exemplo:

$$\{a, e, i, o, u\} \subseteq \{a, \ldots, z\}$$

isto é, a série de vogais é um subconjunto da série de letras do alfabeto. Dizemos que a série A é um *subconjunto próprio* de B, escrito como $A \subseteq B$, mas algum elemento de B não é um elemento de A; veja novamente um exemplo:

$$\{a, e, i, o, u\} \subset \{a, \ldots, z\}$$

Ou seja, a série das vogais é um subconjunto próprio da série de letras do alfabeto porque algumas letras não são vogais.[1] Está claro que, para qualquer série A, A é um subconjunto de si própria, $A \subseteq A$. Além disso, essas relações podem ser invertidas: se $A \subseteq B$, então $.B \supseteq A$, ou seja, se A é um subconjunto de B, então B é um *superconjunto* de A.

Então, podemos definir formalmente a *igualdade de séries* como A igual a B ($A = B$) se, e somente se, $A \subseteq B$ e $B \subseteq A$. Em outras palavras, A é igual a B se, e somente se, cada elemento de A for também um elemento de B e cada elemento de B for também um elemento de A. Informalmente, as séries A e B têm exatamente os mesmos elementos.

Uma série especial é a *série vazia* ou *série nula* $\emptyset$ (ou $\{\}$), que não tem elementos. A série vazia é um subconjunto de todas as séries, incluindo ela própria.

As séries têm muitas operações úteis. Abaixo está um resumo das principais séries para nosso estudo.

O *tamanho de uma série* (representado por $| A |$) é uma contagem do número de seus elementos. Por exemplo, se V é a série de vogais, $| V |$ é 5.

A *união* de duas séries A e B ($A \cup B$) é definida como a série cujos elementos consistem de todos os elementos de A e todos os elementos de B. Por exemplo:

$$\{a, b\} \cup \{b, c, d\} = \{a, b, c, d\}$$

Observe que, embora o tamanho da primeira série seja 2 e o tamanho da segunda série seja 3, o tamanho da união das duas é 4, e não 5.

A *intersecção* de duas séries A e B ($A \cap B$) é definida como sendo a série cujos elementos consistem de todos os elementos que estão tanto em A quanto em B. Por exemplo:

$$\{a, b\} \cap \{b, c, d\} = \{b\}$$

O *produto cruzado* de duas séries A e B ($A \times B$) é a série de todos os pares ordenados $\langle x, y \rangle$ em que x é um membro de A e y é um membro de B. Por exemplo:

$$\{a, b\} \times \{1, 2\} = \{ \langle a, 1\rangle, \langle a, 2\rangle, \langle b, 1\rangle, \langle b, 2\rangle\}$$

A *diferença* entre duas séries A e B ($A - B$) é definida como a série cujos elementos consistem de todos os elementos de A que não sejam também elementos de B. Por exemplo:

$$\{a, b\} - \{b, c, d\} = \{a\}$$

1. Uma maneira alternativa de expressar a segunda parte é dizer que B não é um subconjunto de A, escrito como $B \not\subseteq A$.

A *junção natural* de duas séries A e B ($A \otimes B$), em que A e B são séries de pares ordenados, é definida como a série de pares cujo primeiro elemento está tanto em A quanto em B.
Por exemplo:

$$\{\langle x, 1\rangle, \langle y, 2\rangle\} \otimes \{\langle y, 9\rangle, \langle w, 4\rangle\} = \{\langle y, 2\rangle, \langle y, 9\rangle\}$$

A *união de substituição* ou *negada* de A e B ($A \overline{\cup} B$) é definida como:

$$A \overline{\cup} B = (A - (A \otimes \mathrm{B})) \cup \mathrm{B}$$

Por exemplo:

$$\{\langle \mathrm{x}, 1\rangle, \langle \mathrm{y}, 2\rangle\} \overline{\cup} \{\langle \mathrm{y}, 9\rangle\} = \{\langle \mathrm{x}, 1\rangle, \langle \mathrm{y}, 9\rangle\}$$

Uma série de pares ordenados é uma ferramenta útil para modelar memória, na qual o primeiro componente do par é um nome de variável e o segundo componente do par é o valor da variável. Com isso em mente, a união negada fornece um modelo matemático da operação de atribuição. Usamos a união negada em nosso tratamento dos assuntos de semântica.

Definição: Uma *relação* é um subconjunto de um produto cruzado de duas séries.

Se a série é um produto cruzado binário $A \times B$, então a relação é uma *relação binária*. Se a série é um produto cruzado n-ário $A_1 \times \ldots \times A_n$, então dizemos que a relação é uma *relação n-ária*.

Relações binárias comuns entre membros de uma série são a igualdade e a desigualdade. Como exemplo, podemos citar a igualdade numérica. Por exemplo, $5 = 5$, mas $5 \neq 1$. Na ciência da computação existem muitas outras relações comuns: o valor de uma variável, a visibilidade (pública *versus* privada) de um identificador, o tipo de uma variável etc.

Assim como as séries não têm nenhuma propriedade particular além da participação, as relações também têm apenas a propriedade de que uma tupla, isto é, um elemento particular de um produto cruzado, é ou não um membro de uma relação. Por exemplo, se a memória do computador for modelada como um produto cruzado de nomes e valores de variável, então o elemento $\langle f, 5\rangle$ está ou não na relação da memória do computador.

Uma *função* é um tipo especial de relação. Exemplos de funções são as funções trigonométricas (seno, cos etc.), log, raiz quadrada etc. da matemática.

Definição: Uma relação n-ária R é uma *função*, se sempre que $\langle a_1,..., a_{n-1}, x\rangle \in R$ e $\langle a_1,..., a_{n-1}, y\rangle \in R$, então $x = y$.

Quando vistos como uma função, então os primeiros $n-1$ elementos de R são os argumentos da função e o último elemento é o resultado da função. Assim, R é uma função se, sempre que for dada a ela a mesma série de argumentos, ela retornar sempre o mesmo resultado. Por exemplo, o quadrado de um número x, *quadrado* (x), é uma função; por exemplo, *quadrado*(*2*) é sempre 4.

Se a relação f no produto cruzado das séries A e B for uma função, então representamos isso escrevendo $f : A \rightarrow B$. No estudo das linguagens de programação, isso muitas vezes é chamado de *assinatura* de um método ou de uma função. Por exemplo, a assinatura da função *quadrado*:

$$quadrado : \mathbf{R} \rightarrow \mathbf{R}$$

significa que *quadrado* toma um número real como argumento e retorna um resultado real.

B.2 DIAGRAMAS

Outra idéia matemática que é essencial para o estudo formal das linguagens de programação e sua implementação é o *diagrama*. Em particular, a teoria e a construção de compiladores usam um conjunto de algoritmos diagrama.

> **Definição:** Um *diagrama* tem uma série de nós (ou vértices) N, uma série de arestas E e uma função $g : E \rightarrow N \times N$ que mapeia uma aresta para um par de nós. Se E contém $< v, u >$ sempre que ele contém $< u, v >$, então o diagrama é *indirecionado*. Caso contrário, o diagrama é *direcionado*.

Outro nome para um diagrama direcionado é *dígrafo*.

Por exemplo, um sistema de uma rodovia que é puramente bidirecional pode ser tratado como um diagrama indirecionado. Ao contrário, um diagrama direcionado é como um sistema de uma rodovia com estradas de mão única, onde as estradas de mão dupla são tratadas como duas estradas de mão única, uma em cada direção. Em qualquer dos casos, as intersecções do sistema de rodovias correspondem aos nós.

Podemos traçar diagramas direcionados usando círculos para os nós e os arcos com setas para as arestas. A Figura B.1 é uma representação gráfica do diagrama com nós $N = \{1, 2, 3, 4, 5\}$ e arestas:

Aresta	g(aresta)
a	$\langle 1, 2 \rangle$
b	$\langle 2, 2 \rangle$
c	$\langle 2, 3 \rangle$
d	$\langle 2, 4 \rangle$
e	$\langle 4, 5 \rangle$

O uso de diagramas direcionados ocorre nas Seções 3.2.2 e 3.3.1.

Um *caminho* do nó i para o nó j é uma seqüência de arestas $e_1, \ldots, e_k \in E$ na qual $g(e_1)$, $\ldots$, $g(e_k)$ é a seqüência de pares ordenados $\langle n_1, n_2 \rangle, \langle n_2, n_3 \rangle, \ldots, \langle n_{k-1}, n_k \rangle$ em que $i = n_1$ e $j = n_k$. Isto é, um caminho corresponde a percorrer uma seqüência de arcos na direção da seta.

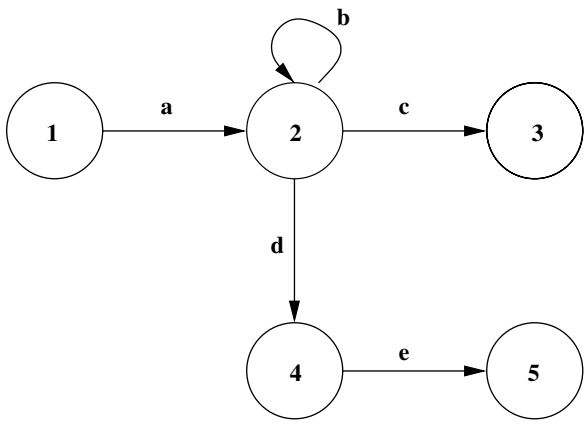


Figura B.1
Exemplo de um Diagrama Direcionado

A Figura B.1 contém um caminho do nó 1 até o nó 3, percorrendo as arestas *a*, *b*, *c*, isso é, indo do nó 1 ao nó 2, do nó 2 para ele mesmo e, finalmente, do nó 2 para o nó 3.

O *comprimento de um caminho* é o número de arcos na seqüência. Ou seja, o exemplo de caminho que vimos tem um comprimento 3. No entanto, um caminho mais curto, do nó 1 ao nó 3, tem um comprimento 2. Um *caminho mais curto* do nó *i* ao nó *j* é aquele com o comprimento mínimo, mas esse caminho pode não ser o único.

Um diagrama direcionado contém um *ciclo*, se ele tiver algum caminho em que um nó é repetido. No exemplo do dígrafo, a aresta *b* por si só constitui um ciclo. Da mesma forma, qualquer caminho que contenha a aresta *b* tem um ciclo.

Naturalmente, pode não haver nenhum caminho do nó *i* ao nó *j*. Dizemos que o nó *j* é acessível a partir do nó *i,* se houver um caminho de *i* a *j*. No exemplo da figura, todo nó é acessível a partir do nó 1, mas o nó 5 não é acessível a partir do nó 3 (e vice-versa).

Um diagrama indirecionado está *conectado* se, para quaisquer dois nós *i* e *j*, houver um caminho de *i* a *j*. A situação é mais complicada para um diagrama direcionado. Dizemos que um diagrama direcionado é *fracamente conectado* se o diagrama considerado um diagrama indirecionado estiver conectado. O diagrama na Figura B.1 é fracamente conectado. Um dígrafo é *unilateralmente conectado* se, para quaisquer dois nós *i* e *j*, houver um caminho de *i* a *j* ou de *j* a *i*. O dígrafo mostrado como exemplo não está unilateralmente conectado porque não há um caminho de 3 para 5 nem um caminho de 5 para 3. Um diagrama direcionado é *fortemente conectado* se, para quaisquer dois nós *i* e *j*, houver um caminho de *i* a *j*.

B.3 LÓGICA

A lógica proposicional está relacionada às *proposições*, que são afirmações que podem ser Verdadeiro (V) ou Falso (F). Um exemplo é a proposição "São duas horas da manhã e tudo está em paz".

Podemos agora formar novas proposições usando operadores lógicos $\wedge$ (que significa "e"), $\vee$ (que significa "ou") e $\neg$ (que significa "não") e outras proposições. Por exemplo, $p \wedge q$, onde p e q são proposições, representa a nova proposição "p e q são ambos verdadeiros". Esses operadores são definidos por aquilo que chamamos de *tabelas verdade*.

Por exemplo, considere a proposição negação $\neg p$. Isso significa que, se p é verdadeiro, então $\neg p$ é falso, e vice-versa. Isso é definido em forma tabular na Tabela B.1. Observe que são dados todos os valores possíveis no domínio de p, ou seja, V e F.

Para os operadores binários *e* e *o*u, cada um dos operandos p e q pode ser V ou F, levando a quatro pares distintos de valores no domínio. A proposição $p \wedge q$ é F se qualquer dos operandos for F; caso contrário, ela é V. Em contraste, a proposição $p \vee q$ é V se qualquer dos operandos for V; caso contrário, ela é F. A tabela verdade para os operadores *e* e *ou* é apresentada na Tabela B.2.

A proposição *if p then q* (*se p então q*) (ou de forma equivalente, *p implica q*) é escrita como $p \Rightarrow q$. A proposição é verdadeira exceto quando p é V e q é F. A tabela dá também

Tabela B.1

Tabela Verdade para $\neg p$

p	$\neg p$
V	F
F	V

Tabela B.2
Tabela Verdade para Operadores E e Ou

p	q	$p \wedge q$	$p \vee q$
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	F

Tabela B.3
Tabela Verdade para Implicação e Equivalência

p	q	$p \Rightarrow q$	$p \Leftrightarrow q$
V	V	V	V
V	F	F	F
F	V	V	F
F	F	V	V

a proposição *p é equivalente a q*, escrita como $p \Leftrightarrow q$. Essa proposição é equivalente a:

$$p \Rightarrow q \wedge q \Rightarrow p$$

Essas tabelas verdade estão na Tabela B.3.

Podemos estender a idéia de uma proposição para incluir aquilo que chamamos de quantificadores, constantes e variáveis que podem assumir valores em diferentes domínios além daquele de {V, F}. Quando estendemos proposições dessa maneira, temos o conceito de um atributo.

Os dois principais quantificadores usados em atributos são $\forall$ ("para todo") e $\exists$ ("existe"). Um exemplo de um atributo seria:

> Para todo número primo x, existe um outro número primo y, tal que y é maior do que x.

Essa é uma maneira mais formal de se dizer que não há um maior número primo. Como x e y são variáveis que compreendem os números naturais **N**, esse exemplo pode ser escrito como:

$$\forall x \ (x \text{ é primo} \Rightarrow \exists y(y \text{ é primo} \wedge y > x))$$

Atributos podem ser definidos da maneira a seguir. Primeiro, um *termo* pode ser:

1. Uma constante
2. Uma variável
3. $f(t_1, \ldots, t_n)$, em que f é uma função e $t_1, \ldots, t_n$ são termos.

Segundo, uma *fórmula atômica* pode ser:

1. Uma variável atributo
2. $P(t_1, \ldots, t_n)$, em que P é um atributo e $t_1, \ldots, t_n$ são termos.

Terceiro, um *atributo* pode ser:

1. Uma fórmula atômica.
2. $\forall x(P)$ e $\exists x(P)$, em que P é um atributo e x é uma variável individual.
3. $\neg P$, $P \wedge Q$, $P \vee Q$, $P \Rightarrow Q$, e $P \Leftrightarrow Q$, em que P e Q são atributos.

O atributo $\forall x\ (P(x))$ é lido como: "para todo x (no domínio implícito), x tem a propriedade p". Reescrevendo novamente nosso exemplo de números primos, obtemos:

$$\forall x(x \in \mathbf{N} \wedge \mathit{éPrimo}(x) \Rightarrow \exists y(y \in \mathbf{N} \wedge \mathit{éPrimo}(y) \wedge y > x))$$

em que **N** é o conjunto de números naturais.

Finalmente, reconsideramos os conceitos de variáveis livres e ligadas. O *escopo* de uma variável é a expressão à qual o quantificador é aplicado. Por exemplo, o escopo do quantificador universal $\forall(x)P$ é P. A variável x é *ligada* se ela ocorrer dentro do escopo de seu quantificador. Caso contrário, dizemos que ela é *livre*. Por exemplo, a variável x é livre e a variável y é ligada no atributo $\exists y(y \in \mathbf{N}\ \ \mathit{éPrimo}(y) \wedge y > x)$.

As proposições possibilitam representações simbólicas para *expressões lógicas*; isso é, instruções que podem ser interpretadas como *verdadeiro* (true) ou *falso* (false). Por exemplo, se p representa a proposição "Maria fala russo" e q representa a proposição "Bob fala russo", então $p \wedge q$ representa a proposição "Maria e Bob falam (ambos) russo", e $p \vee q$ representa a proposição "Maria ou Bob (ou ambos) fala russo". Se, além disso, r representa a proposição "Maria e Bob podem se comunicar", então a expressão $p \wedge q \Rightarrow r$ representa "Se Maria e Bob ambos falam russo, então eles podem se comunicar".

Atributos incluem todas as proposições como aquelas acima e também incluem variáveis em diversos domínios (inteiros, reais, *strings*, listas etc.), funções com valores boleanos com essas variáveis e quantificadores. Um *atributo* é uma proposição na qual algumas das variáveis boleanas são substituídas por funções com valores boleanos e expressões quantificadas.

Uma *função com valor boleano* é uma função com um ou mais argumentos que fornece como resultado *verdadeiro* (true) ou *falso* (false). Aqui estão alguns exemplos:

$\mathit{primo}(n)$ − *verdadeiro* (*true*), se o valor inteiro n for um número primo; *falso* (*false*) em caso contrário.

$0 \leq x + y$ − *verdadeiro* se a soma real de x e y for não-negativa.

$\mathit{fala}(x, y)$ − *verdadeiro* se a pessoa x fala a língua y.

Um atributo combina esses tipos de funções usando os operadores do cálculo proposicional e os quantificadores $\forall$ (que significa "para todo") e $\exists$ (que significa "existe"). Veja alguns exemplos:

$0 \leq x \wedge x \leq 1$ − *verdadeiro* se x estiver entre 0 e 1, inclusive; caso contrário, *falso*.

$\mathit{fala}(x, \mathit{Russo}) \wedge \mathit{fala}(y, \mathit{Russo}) \Rightarrow \mathit{comunicacom}(x, y)$ − *verdadeiro* se o fato de que tanto x quanto y falam russo implica que x se comunica com y; caso contrário, *falso*.

$\forall x(\mathit{fala}(x, \mathit{Russo}))$ − *verdadeiro* se todos no planeta falam russo; *falso*, caso contrário.

$\exists x(\mathit{fala}(x, \mathit{Russo}))$ − *verdadeiro* se pelo menos uma pessoa no planeta fala russo; *falso* em caso contrário.

$\forall x \exists y(\mathit{fala}(x,y))$ − *verdadeiro* se toda pessoa no planeta fala uma linguagem; *falso* em caso contrário.

$\forall x(\neg \mathit{alfabetizado}(x) \Rightarrow (\neg \text{escreve}(x) \wedge \neg\ \exists y(\mathit{livro}(y) \wedge \mathit{láleu}(x,y))))$ − *verdadeiro* se toda pessoa alfabetizada x não escreve e não lê um livro.

A Tabela B.4 resume os significados dos diferentes tipos de expressões que podem ser usados em lógica proposicional e atributo.

| Tabela B.4 Resumo da Notação Lógica de Atributo

Notação	Significado
true, *false*	Constantes boleanas (truth)
p, q, . . .	Variáveis boleanas
$p(x, y \ldots), q(x, y \ldots), \ldots$	Funções boleanas
$\neg p$	Negação de p
$p \wedge q$	Conjunção de p e q
$p(x) \vee q(x)$	Disjunção de p e q
$p(x) \Rightarrow q(x)$	Implicação: p implica q
$p(x) \Leftrightarrow q(x)$	Equivalente lógico de p e q
$\forall x\ p(x)$	Expressão quantificada universalmente
$\exists x\ p(x)$	Expressão quantificada existencialmente
$p(x)$ é válida	Atributo $p(x)$ é *verdadeiro* para todo valor de x
$p(x)$ é possível de ser satisfeita	Atributo $p(x)$ é *verdadeiro* para pelo menos um valor de x
$p(x)$ é uma contradição	Atributo $p(x)$ é *falso* para todo valor de x

Atributos que são *verdadeiros* (*true)* para todos os valores possíveis de suas variáveis são chamados *válidos*. Por exemplo, $par(x) \vee ímpar(x)$ é válido, pois todos os inteiros x são par ou ímpar. Atributos que são *falsos* (*false*) para todos os valores possíveis de suas variáveis são chamados *contradições*. Por exemplo, $par(x) \wedge ímpar(x)$ é uma contradição, pois nenhum inteiro pode ser par e ímpar ao mesmo tempo.

Atributos que são *verdadeiros* (*true*) para alguma atribuição particular de valores de suas variáveis são chamados possíveis de serem satisfeitos. Por exemplo, o atributo *fala*(x, *Russo*) é satisfeito (mas não válido) porque presumivelmente pelo menos uma pessoa no planeta fala russo (mas há outras pessoas que não falam). Semelhantemente, o atributo $y \geq 0 \wedge n \geq 0 \wedge z = x(y - n)$ é satisfeito, mas não válido, pois podem ser encontradas diferentes seleções de x, y, z e n que tornam esse atributo *verdadeiro* ou *falso*.

Os atributos têm várias propriedades algébricas, que muitas vezes são úteis quando estamos analisando e transformando expressões lógicas. Na Tabela B.5 está um resumo dessas propriedades.

As propriedades comutativa, associativa, distributiva e idempotente têm interpretações diretas. A propriedade identidade simplesmente diz que uma proposição ou sua negação devem sempre ser *verdadeiras*, mas que a proposição e sua negação não podem ambas simultaneamente ser *verdadeiras*.

A propriedade de DeMorgan fornece um dispositivo conveniente para remover a disjunção (ou conjunção) de uma expressão sem mudar seu significado. Por exemplo, dizer "não está chovendo ou nevando" é equivalente a dizer "não está chovendo e não está nevando". Além disso, essa propriedade declara a equivalência de "João e Maria não estão ambos na escola" e "ou João ou Maria não está na escola".

Semelhantemente, as propriedades de implicação e quantificação fornecem veículos para remover implicações, quantificadores universais ou existenciais de uma expressão, sem mudar seu sentido. Por exemplo, "nem toda criança consegue ler" é equivalente a dizer

Tabela B.5 Propriedades dos Atributos

Propriedade	Significado	
Comutatividade	$p \vee q \Leftrightarrow q \vee p$	$p \wedge q \Leftrightarrow q \wedge p$
Associatividade	$(p \vee q) \vee r \Leftrightarrow p \vee (q \vee r)$	$(p \wedge q) \wedge r \Leftrightarrow p \wedge (q \wedge r)$
Distributividade	$p \vee q \wedge r \Leftrightarrow (p \vee q) \wedge (p \vee r)$	$p \wedge (q \vee r) \Leftrightarrow p \wedge q \vee p \wedge r$
Idempotente	$p \vee p \Leftrightarrow p$	$p \wedge p \Leftrightarrow p$
Identidade	$p \vee \neg p \Leftrightarrow true$	$p \wedge \neg p \Leftrightarrow false$
deMorgan	$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$	$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$
Implicação	$p \Rightarrow q \Leftrightarrow \neg p \vee q$	
Quantificação	$\neg \forall x\, p(x) \Leftrightarrow \exists x \neg p(x)$	$\neg \exists x\, p(x) \Leftrightarrow \forall x \neg p(x)$

"há pelo menos uma criança que não consegue ler". Semelhantemente, dizer "não há moscas na minha sopa" é equivalente a dizer "todas as moscas não estão na minha sopa".

B.4 REGRAS DE INFERÊNCIA E PROVA DIRETA

Um argumento a ser provado, muitas vezes, assume a forma $p_1 \wedge p_2 \wedge \cdots \wedge p_n \Rightarrow q$, na qual os *p*s são as hipóteses e q é a conclusão.

Uma *prova direta* de um argumento desses é uma seqüência de atributos válidos, cada um dos quais é idêntico a uma hipótese ou uma derivável de atributos anteriores na seqüência usando uma *propriedade* (Tabela B.5) ou uma *regra de inferência*. O último atributo na prova deve ser a conclusão q do argumento. Cada atributo na seqüência é acompanhado de uma "justificação", que é uma breve notação de qual regra de influência e quais etapas anteriores foram usadas para chegar a esse atributo.

Algumas das regras de inferência principais para atributos estão resumidas na Tabela B.6. Para interpretar essas regras, se a expressão (ou as expressões) à esquerda do

Tabela B.6 Regras de Inferência para Atributos

Regra de Inferência	Significado
Modus ponens (modo de afirmar)	$p, p \Rightarrow q \vdash q$
Modus tollens (modo que nega)	$p \Rightarrow q, \neg q \vdash \neg p$
Conjunção	$p, q \vdash p \wedge q$
Simplificação	$p \wedge q \vdash p$
Adição	$p \vdash p \vee q$
Instanciação universal	$\forall x\, p(x) \vdash p(a)$
Instanciação existencial	$\exists x\, p(x) \vdash p(a)$
Generalização universal	$p(x) \vdash \forall x\, p(x)$
Generalização existencial	$p(a) \vdash \exists x\, p(x)$

sinal $\vdash$ aparece em uma prova, ela pode ser substituída mais adiante na seqüência pela expressão que está à direita (mas *não* vice-versa). Abaixo está uma prova direta do seguinte argumento:

> Todo estudante gosta de palavras cruzadas. Alguns estudantes gostam de sorvete. Portanto, alguns estudantes gostam de sorvete e de palavras cruzadas.

Suponha que atribuímos os seguintes nomes aos atributos nesse problema:
$S(x)$ = "x é um estudante"
$C(x)$ = "x gosta de palavras cruzadas"
$I(x)$ = "x gosta de sorvete"
Então o argumento pode ser escrito como:

$$\forall x(S(x) \rightarrow C(x)) \wedge \exists x(S(x) \wedge I(x)) \rightarrow \exists x(S(x) \wedge C(x) \wedge I(x))$$

Aqui está uma prova direta desse argumento:

1	$\forall x(S(x) \rightarrow C(x))$	Hipótese
2	$\exists x(S(x) \wedge I(x))$	Hipótese
3	$S(a) \wedge I(a)$	2, Instanciação existencial
4	$S(a) \rightarrow C(a)$	1, Instanciação universal
5	$S(a)$	3, Simplificação
6	$C(a)$	4, 5, *Modus ponens*
7	$S(a) \wedge C(a) \wedge I(a)$	3, 6, Conjunção
8	$S(a) \wedge I(a) \wedge C(a)$	7, Comutatividade
9	$\exists x(S(x) \wedge I(x) \wedge C(x))$	8, Generalização existencial

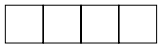
As observações na coluna da direita são justificações para os passos individuais na prova. Cada justificação inclui números de linha de etapas anteriores a partir das quais se pressupõe uma propriedade ou uma regra de inferência da Tabela B.5 ou B.6.

B.5 PROVA POR INDUÇÃO

Este método de prova é muito importante na precisão do programa como também em muitas outras áreas da ciência da computação. Uma *prova por indução* pode ser aplicada a qualquer argumento que tenha a forma $\forall n\ p(n)$. Aqui, o domínio de n deve ser calculável, como é o caso dos inteiros ou das *strings* de caracteres ASCII, por exemplo. A estratégia para uma prova por indução tem duas etapas:[2]

1 (Etapa básica) Prova $p(1)$.
2 (Etapa de indução) Assumindo a hipótese de que $p(k)$ é válida para um valor arbitrário de $k > 1$ no domínio de n, prova-se $p(k + 1)$.

Considere o exemplo a seguir. Suponha que queremos provar por indução que o número de lados distintos em uma fileira de n quadrados adjacentes é $3n + 1$. Aqui, por exemplo, está uma fileira de quatro quadrados adjacentes, tendo 13 lados adjacentes:



2. Esta estratégia, muitas vezes, é chamada de "indução fraca". A estratégia de "indução forte" difere somente na suposição que se faz durante a etapa de indução. Isto é, com a indução forte, você pode criar a hipótese de que $p(1)$, $p(2)$, ..., $p(k)$ são todas válidas para um valor arbitrário de $k > 1$, para provar $p(k + 1)$.

Aqui está a prova indutiva:

1 A etapa básica é simples, já que um quadrado tem $3 \times 1 + 1 = 4$ lados.

2 Para a etapa de indução, suponha como nossa *hipótese de indução* que k quadrados têm $3k + 1$ lados. Agora precisamos provar que isso leva à conclusão de que $k + 1$ quadrados têm $3(k + 1) + 1$ lado. Mas, para construir um quadrado $k + 1$ agora, nós precisamos apenas acrescentar três lados à fileira de quatro quadrados. Isso leva à conclusão de que o número de lados em uma fileira de $k + 1$ quadrados é $3k + 1 + 3 = 3(k + 1) + 1$, que completa a etapa de indução.

Glossário

Abstração procedural O processo de *abstração procedural* permite ao programador se preocupar principalmente com a interface entre a função e o que ela calcula, ignorando os detalhes de como o cálculo é executado. 280

Acesso aleatório Arquivos de *acesso aleatório* (ou *direto*) permitem o processamento não seqüencial de informações em um arquivo. 175

Agregação A classe C_1 é uma *agregação* de uma classe C_2, se C_1 contiver objetos do tipo C_2. 320

Aliasing *Aliasing* ocorre quando, dentro de uma função ou um procedimento, a mesma localização de memória pode ser acessada com o uso de nomes diferentes. 233

Ambiente de referência Para escopo estático, o *ambiente de referência* para um nome é seu escopo de definição e todos os subescopos aninhados. 93

Ambiente O *ambiente y,* de uma função ativa *f* é um conjunto de pares que unem variáveis cujo escopo inclui *f,* com *endereços de memória* específicos. 255

Analisador O propósito do *analisador sintático*, ou *analisador*, é construir uma árvore de análise usando como entrada o fluxo de *tokens* fornecido pelo *lexer*. 70

Analisador descendente recursivo Um analisador *descendente recursivo* é um analisador no qual cada não-terminal na gramática é convertido para uma função que reconhece entradas deriváveis desse não-terminal. 71

Analisador léxico A função do *analisador léxico* (ou simplesmente *lexer*) é varrer o programa e transformá-lo em um fluxo de *tokens*. Como parte dessa transformação, todos os espaços em branco, incluindo comentários, são descartados. 42

Analisador semântico A fase do *analisador semântico* de um compilador assegura que as regras de semântica de tempo de compilação da linguagem são impostas, que todos os identificadores referenciados estão declarados e todos os operandos, para cada operador, possuem um tipo apropriado. 45

Analisador sintático O *analisador sintático* (ou *analisador*) lê um fluxo de *tokens* e constrói uma árvore de análise, de acordo com as regras da gramática. 45

Applet Uma *applet* é um programa em Java projetado para ser executado dentro de um navegador *web*. 467

Argumento Uma expressão que aparece em uma chamada de função ou procedimento é conhecida como *argumento*. 227

Arquitetura cliente-servidor Uma *arquitetura cliente-servidor* é uma arquitetura de rede na qual cada computador ou processo na rede é um cliente ou um servidor. 506

Arquivos As origens e os destinos de operações de entrada e saída são chamados de *arquivos*. 169

Árvore de análise *Árvore de análise* é uma derivação na qual cada passo é desenhado como uma subárvore. 28

Asserção Uma *asserção* é um predicado boleano que descreve o *estado* de um programa em um determinado ponto de sua execução. 521

Assinatura da função Uma *assinatura da função* é uma declaração do nome de uma função junto ao seu tipo de retorno e aos tipos e a ordem dos seus parâmetros. Às vezes, isto é chamado de *protótipo da função*. 287

Associatividade A *associatividade* especifica se operadores com a mesma precedência devem ser executados da esquerda para a direita ou da direita para a esquerda. 30

Autômato de estado finito Um *autômato de estado finito* possui:
1 Um conjunto de estados representado por nodos de um grafo.
2 Um alfabeto de entrada aumentado por um símbolo único que representa o final da entrada.
3 Uma função de transição de estado representada por arcos direcionados de um nodo para o outro, rotuladas com um ou mais símbolos do alfabeto.
4 Um estado inicial único.
5 Um conjunto de um ou mais *estados finais* (estados sem arestas de saída). 63

Autômato de estado finito determinístico Um autômato de estado finito é *determinístico* se, para cada estado e símbolo de entrada, houver no máximo um arco de saída do estado rotulado com esse símbolo de entrada. 63

Avaliação de curto-circuito Avalia uma expressão boleana da esquerda para a direita e pára assim que a verdade da expressão puder ser determinada. 158

Avaliação lenta Uma alternativa à estratégia da avaliação rápida é denominada *avaliação lenta*, na qual um argumento para uma função não é avaliado até que ele seja necessário. 365

Avaliação rápida *Avaliação rápida* em linguagens funcionais se refere à estratégia de avaliar todos os argumentos de uma função no momento da chamada. 365

BNF expandida A *BNF expandida* (ou simplesmente EBNF) simplifica a escrita de uma gramática introduzindo meta-símbolos para iteração, opção e escolha. 35

Campos Os elementos individuais de uma estrutura são freqüentemente chamados de *campos*. 120

Capturar uma exceção *Capturar uma exceção* significa transferir o controle para um manipulador de exceções, o qual define a resposta que o programa executa quando a exceção ocorre. 180

Classe Uma *classe* é um tipo de declaração que encapsula constantes, variáveis e funções para manipulação dessas variáveis. 316

Classe abstrata Uma *classe abstrata* é aquela declarada como abstrata ou que tenha um ou mais métodos abstratos. 326

Classe imutável Objetos de uma *classe imutável* não podem ser alterados após terem sido criados. 338

Classe interna Uma *classe interna* é uma definição de classe totalmente contida dentro de outra classe. 316

Cláusula de Horn Uma *cláusula Horn* possui um cabeçalho *h*, que é um predicado, e um corpo, que é uma lista de predicados $p_1, p_2, ..., p_n$. 414

Cliente de uma classe Um *cliente de uma classe* C é outra classe ou um método que declare ou use um objeto da classe C. 317

Coerção de tipo O termo *coerção de tipo* é usado para denotar uma mudança implícita do tipo de um valor para outro, podendo ou não envolver uma conversão de tipo. 110

Coleta de lixo O termo *coleta de lixo* se refere a qualquer estratégia para recuperar blocos do *heap* de memória que não serão mais usados posteriormente pelo programa. 269

Compartilhamento do tempo O *compartilhamento do tempo* permite a duas ou mais pessoas usarem teclados e monitores para se comunicar simultaneamente com um computador. 485

Compilador Um *compilador* traduz um programa da sua representação em linguagem fonte para código de máquina. 42

Completa quanto a Turing Uma linguagem de programação é *completa quanto a Turing* se os seus programas forem capazes de calcular qualquer função calculável. 164

Comunicação interprocessos A *comunicação interprocessos (IPC)* ocorre quando dois ou mais programas diferentes se comunicam simultaneamente por uma rede. 506

Concorrência *Concorrência* ocorre quando o programa possui um conjunto de elementos assíncronos, os quais podem compartilhar informações ou sincronizarem-se em intervalos de tempo. 5

Clareza nas ligações Um elemento de linguagem é *conectado* a uma propriedade quando essa propriedade é definida para ele. 15

Conversão ampliadora Uma conversão de tipo é denominada *conversão ampliadora* se o valor resultante não requerer menos bits do que o valor original (geralmente sem perda de informação). 109

Conversão de tipo Para simular um modo de operação misto, um dos operandos deve ter seu valor convertido para o tipo do outro. Tal conversão é chamada de *conversão de tipo*, já que cria um padrão de bits diferente para representar o valor em um tipo diferente. 109

Conversão limitante Uma *conversão* de tipo é chamada *limitante* se o valor resultante permitir menos bits do que o original (dessa forma potencialmente perdendo informações). 109

Correção de programa Um programa está *correto* se satisfizer suas especificações formais para todas as suas entradas possíveis. 5

Correção parcial Um programa (ou comando) está *parcialmente correto* quanto a asserções P e Q se, sempre que s começar a execução em um estado que satisfizer a P, o estado resultante satisfizer a Q (supondo que o comando s termine). 523

Correção total Um programa (ou comando) s está *totalmente correto* quanto às asserções P e Q se s estiver parcialmente correto quanto às asserções P e Q e se, sempre que s começar a executar em um estado que satisfaça P, s com certeza terminar. 523

Corrida Uma condição de *corrida* (às vezes chamada de *corrida crítica*) ocorre quando o valor resultante de uma variável pode diferir, dependendo de qual de duas ou mais *threads* grava nela primeiro. 487

Deadlock Um *deadlock* ocorre quando uma *thread* está esperando por um evento que nunca acontecerá. 489

Derivação Uma *derivação* é uma seqüência de *strings* separada pelo símbolo $\Rightarrow$ na qual, em cada etapa, um não-terminal é substituído pelo lado direito de uma de suas produções. 27

Derivação mais à direita Uma *derivação mais à direita* é uma derivação na qual em cada etapa substituímos o não-terminal mais à direita de uma de suas alternativas. 28

Derivação mais à esquerda Uma *derivação mais à esquerda* é uma derivação na qual, em cada etapa, substituímos o não terminal mais à esquerda por uma das suas alternativas. 27

Diagrama de sintaxe Uma forma de gramática que usa diagramas. 36

Domínio de semântica Um *domínio de semântica* é um conjunto de valores cujas propriedades e operações são bem compreendidas independentemente, e nos quais as regras que definem a semântica de uma linguagem podem se basear. 198

Efeito colateral Um *efeito colateral* ocorre durante a avaliação de uma expressão se, além de retornar um valor, a expressão também alterar o estado da computação. 205

Encapsulamento *Encapsulamento* é um mecanismo que permite que constantes, tipos, variáveis, métodos, e assim por diante, relacionados logicamente, sejam agrupados em uma nova entidade. Exemplos incluem procedimentos, pacotes e classes. 310

Endereço de memória *Endereços de memória* normalmente são um conjunto contíguo {0, ..., n} dos inteiros, e muitas vezes são chamados de *espaço de endereçamento* do programa. 255

Equivalência de nome Sob *equivalência de nome,* dois tipos são o mesmo, se tiverem o mesmo nome. 126

Equivalência estrutural Na *equivalência estrutural*, dois tipos são o mesmo se tiverem a mesma estrutura. Para tipos *registro*, ter a mesma estrutura inclui o número e a ordem dos campos, assim como o nome e o tipo de cada campo. 126

Erro de tipo Um *erro de tipo* é qualquer erro que surja devido a uma tentativa de realização de operação sobre um tipo de dado para o qual ela não está definida. 103

Escopo O *escopo* de um nome é o conjunto de comandos que podem acessar a ligação desse nome. 89

Escopo de definição O escopo em que um nome é definido ou declarado é chamado *escopo de definição*. 90

Escopo dinâmico No *escopo dinâmico*, um nome é conectado à sua declaração mais recente com base no histórico de execução do programa. 94

Escopo estático No *escopo estático*, um nome é ligado a um conjunto de comandos de acordo com sua posição no programa-fonte. 89

Espaço em branco Um *espaço em branco* comumente inclui os caracteres de espaço e tabulação, o caractere ou os caracteres de final de linha e os comentários. 39

Estado O *estado* de um programa é o conjunto de conexões de todos os objetos ativos com seus valores correntes. 160

Estrutura Uma *estrutura* ou *registro* é um conjunto de elementos com tipos potencialmente diferentes. 120

Exceção Uma *exceção* é uma condição de erro gerada por uma operação que não pode ser resolvida pela própria operação. 179

Fase Uma *fase* do compilador é uma operação logicamente coesa que transforma o programa fonte de uma representação para outra. 42

Fatia Uma *fatia* de uma lista ou matriz é uma série contígua de entradas especificada pelo seu índice inicial e comprimento. 119

Fatiamento de tempo O *fatiamento de tempo* divide esse tempo em pequenos blocos e os distribui entre os processos de uma forma imparcial. 485

Forma sentencial Uma *string* que contém tanto símbolos terminais quanto não-terminais. 27

Fortemente tipada Uma linguagem de programação é *fortemente tipada* se o seu sistema de tipos permitir que todos os erros relativos a tipos em um programa sejam detectados em tempo de compilação ou em tempo de execução. 104

Frame de pilha Um *frame de pilha* é um sinônimo de registro de ativação quando este último é alocado e desalocado dinamicamente com o uso de uma pilha. 237

Funções não-avaliadas Uma função não-avaliada é uma função na qual alguns dos seus *n* argumentos são fixados. 390

Função genérica Uma *função genérica* ou *procedimento genérico* é um modelo que pode ser instanciado em tempo de compilação com operadores e tipos concretos. 128

Função parcial Uma *função parcial* é a função que não está bem definida para todos os valores possíveis do seu domínio (estado de entrada). 198

Função polimórfica Uma *função polimórfica* é uma função cuja definição se aplica igualmente bem a argumentos de diversos tipos, dentro de restrições dadas pela assinatura da função. 396

Função total Uma função é dita *total* se for definida para todos os elementos no seu domínio e *parcial,* caso contrário. 362

Genéricos Veja *modelos.* 325

Gerar uma exceção *Gerar uma exceção* é sinalizar que a condição que ela representa ocorreu. 180

Gramática ambígua Uma gramática é *ambígua* se a sua linguagem contiver pelo menos uma *string* com duas ou mais árvores de análise distintas. 31

Gramática livre de contexto Uma *gramática livre de contexto* possui um conjunto de produções *P*, um conjunto de símbolos terminais *T* e um conjunto de símbolos não terminais *N*; um dos quais, *S*, é distinguido como o *símbolo inicial*. 25

Heap O *heap de memória* contém valores que são alocados dinamicamente enquanto o programa está sendo executado, como *strings*, matrizes dinâmicas, objetos e diversas estruturas dinâmicas de dados como listas encadeadas. 264

Herança Suponha que uma subclasse *D* simples estenda a classe *C*; a subclasse *D* é dita ter herdado todas variáveis e os métodos de instância de *C*, incluindo construtores e destrutores. 320

Herança múltipla Em contraste com linguagens de herança simples, algumas linguagens orientadas a objeto suportam *herança múltipla*, o que permite que uma classe seja uma subclasse de uma ou *mais* superclasses. 321

Identificação de tipo em tempo de execução *Identificação de tipo em tempo de execução* (da sigla em inglês RTTI) é a capacidade da linguagem de identificar em tempo de execução o tipo ou a classe de um objeto. 330

Inanição A aquisição de recursos deve ser administrada de modo que nenhuma *thread* seja indevidamente atrasada ou tenha negado o seu acesso a um recurso de que precise. Uma ocorrência desta última é muitas vezes chamada de *lockout* ou *inanição*. 488

Instanciação (durante a resolução) A atribuição de valores a variáveis durante a resolução é chamada de *instanciação*. 417

Interface Uma *interface* encapsula um conjunto de constantes e assinaturas de métodos abstratos. Uma interface pode não incluir variáveis, construtores ou métodos não abstratos. 327

Interface gráfica de usuário (GUI) Uma aplicação de *interface gráfica de usuário (GUI)* é um programa executado dentro da sua própria janela, que se comunica com os usuários utilizando elementos gráficos como botões e menus. 453

Interpretador Em contraste com um compilador, um *interpretador* é uma forma de tradutor na qual as duas últimas fases do compilador são substituídas por um programa que executa diretamente (ou interpreta) o código intermediário. 47

Invariante do laço *Um invariante do laço* é uma asserção que permanece *verdadeira* antes e depois de cada iteração de um laço. 527

Iterador Um *iterador* é qualquer conjunto finito de valores sobre os quais um laço pode ser repetido. 167

Junção natural A *junção natural* $estado_1 \otimes estado_2$ é o conjunto de todos os pares em $estado_1$ e $estado_2$ que possuam o mesmo primeiro membro. 215

Lexer Veja analisador léxico. 42

Ligação dinâmica Uma conexão é *dinâmica* se a associação ocorrer em tempo de execução. 86

Ligação estática Uma ligação é *estática* se a associação ocorrer antes do tempo de execução. 86

Linguagem baseada em objetos O conjunto de comportamentos que permite a um programa criar qualquer número de instância de um tipo de dado abstrato é muitas vezes caracterizado como uma *linguagem baseada em objeto*. 318

Linguagem de uso geral Algumas linguagens são intencionalmente de uso mais geral no seu projeto, objetivando servir aos interesses de uma ampla gama de aplicações. 12

Linguagem funcional pura Uma linguagem funcional é *pura* se não houver algum conceito de operador de atribuição ou célula de memória; caso contrário, ela é denominada *impura*. 363

Linguagem L A *linguagem L* definida por uma gramática *G* é o conjunto de todas as *strings* terminais que podem ser derivadas do símbolo inicial. 27

Linguagem orientada a objetos Uma linguagem é *orientada a objeto* se ela suportar um mecanismo de encapsulamento que esconda informações para definir tipos abstratos de dados, métodos virtuais e herança. 323

Linguagem tipada dinamicamente Uma linguagem é *tipada dinamicamente* se o tipo de uma variável puder variar em tempo de execução, dependendo do seu valor. 104

Linguagem tipada estaticamente Uma linguagem é *tipada estaticamente* se os tipos de todas as variáveis forem fixados quando forem declarados em tempo de compilação. 104

Lixo *Lixo* é qualquer bloco do *heap* de memória que não possa ser acessado pelo programa. 267

Manipulação de eventos *Manipulação de eventos* ocorre com programas que respondem a eventos que sejam gerados em uma ordem imprevisível. 5

Mapa de tipo Um *mapa de tipo* é uma função que liga cada nome de variável declarado ao seu tipo. 137

Mapa de tipo para uma função O *mapa de tipo* para uma função *f* é definido como um conjunto de pares e triplos, cada um representando uma variável global, uma função global, um parâmetro ou uma variável local. 252

Matriz heterogênea Os valores de diferentes entradas podem ter tipos diferentes. 119

Memória μ A *memória* μ é um conjunto de pares que relacionam endereços com valores. 255

Memória estática A *memória estática* contém valores cujos requisitos de armazenamento são conhecidos antes do tempo de execução e permanecem constantes durante todo o tempo de execução do programa. 264

Metalinguagem Uma *metalinguagem* é uma linguagem usada para descrever outras linguagens. 24

Metassímbolo Um *metassímbolo* é um símbolo que faz parte da metalinguagem e não faz parte da linguagem que está sendo definida. 25

Método abstrato Um *método abstrato* é aquele que não contém código além da sua assinatura. 326

Modelo Um *modelo* define uma família de classes ou funções parametrizadas por um ou mais tipos. 325

Modelo View-Controler (MVC) No *MVC*, o modelo é o objeto real sendo implementado, o controlador é o mecanismo de entrada (botões, menus, caixas de combinação etc) e a visão é a representação da saída do modelo. 449

Notação de prefixo de Cambridge Uma notação para escrever expressões nas quais um operador sempre precede seus operandos e a expressão inteira fica entre parênteses. 156

Notação interfixada Uma notação para escrever expressões nas quais cada operador binário é escrito entre seus operandos. 155

Notação polonesa pré-fixada Uma notação para a escrita de expressões nas quais um operador sempre precede seus operandos. 156

Notação polonesa pós-fixada Uma notação para a escrita de expressões nas quais um operador sempre segue seus operandos. 156

Objeto Cada instância de uma classe é um *objeto*. 316

Ortogonal Uma linguagem é dita *ortogonal* se os seus comandos e as características forem construídos sobre um conjunto pequeno e mutuamente independente de operações primitivas. 17

Overflow do heap *Overflow do heap* ocorre quando uma chamada a *new* ocorre e o *heap* não contém um bloco contíguo de palavras de memória não usadas grande o suficiente para satisfazer à chamada. 265

Palavras reservadas A maioria das linguagens possui um conjunto pré-definido de nomes chamados de *palavras reservadas* ou *palavras-chave*, que possuem significado especial e não podem ser usados como identificadores. 86

Paradigma Um *paradigma* de programação é um padrão de resolução de problemas que se relaciona a um determinado tipo de programas e linguagens. 3

Parâmetro Um identificador que aparece em uma declaração de função ou procedimento é denominado de *parâmetro*. 228

Passagem por endereço Veja: Passagem por referência. 231

Passagem por nome Um argumento passado por *nome* se comporta como se fosse textualmente substituído para cada ocorrência do parâmetro. 234

Passagem por referência Passar um argumento *por referência* (ou *por endereço*) significa que o *endereço de memória* do argumento é copiado para o parâmetro correspondente, de modo que o parâmetro se torna uma *referência* indireta (ponteiro) ao argumento real. 231

Passagem por resultado Um argumento passado *por resultado* é implementado pela cópia do valor final calculado para o parâmetro, para o argumento no final da chamada. 233

Passagem por resultado-valor Um argumento passado por *resultado-valor* é implementado pela cópia do valor do argumento para o parâmetro, no início da chamada, e depois tem copiado seu resultado calculado de volta para o argumento correspondente, no final da chamada. 233

Passagem por valor Passar um argumento *por valor* significa que o valor do argumento é calculado quando da chamada e copiado para o parâmetro correspondente. 229

Passo Um *passo* do compilador corresponde a uma leitura completa da representação corrente do programa fonte. 42

Pilha de tempo de execução Uma *pilha de tempo de execução* é uma pilha de ativação de registros usada para modelar a semântica de chamadas e retornos de funções. 239

Polimorfismo Uma função ou operação é *polimórfica* se puder ser aplicada a qualquer um dos diversos tipos relacionados e obter o mesmo resultado. 127

Polimorfismo, linguagens orientadas a objeto Em linguagens orientadas a objeto, *polimorfismo* se refere à ligação tardia de uma chamada a uma de diversas implementações de um método em uma hierarquia de herança. 323

Ponteiro Um *ponteiro* é um valor que representa um endereço de memória ou uma referência. 113

Pós-condição Uma *pós-condição* de um programa é uma asserção que declara o resultado desse programa. 522

Precedência Um operador possui *precedência* mais alta do que outro operador se o primeiro tiver de ser avaliado mais cedo em todas as expressões sem parênteses envolvendo apenas os dois operadores. 30

Pré-condição Uma *pré-condição* de um programa é uma asserção que expressa o que deve ser *verdadeiro* antes que a execução do programa comece para que a pós-condição seja válida. 522

Princípio da substituição Um método de uma subclasse é *substituível* por um método da classe mãe se o método da subclasse executar a mesma função geral. 325

Processo Um *processo* denota uma seqüência de comandos em execução. 485

Produção Uma *produção* gramatical possui a forma $A \rightarrow w$, em que A é um símbolo não-terminal e w é uma *string* de símbolos não-terminais e terminais. 25

Programa concorrente Um *programa concorrente* é projetado para ter dois ou mais contextos de execução. Tal programa é dito *multithread*, já que mais de um contexto de execução pode estar ativo simultaneamente durante a sua execução. 486

Programação funcional A *programação funcional* modela um problema computacional na forma de um conjunto de funções matemáticas, cada um com uma entrada (domínio) e saída (faixa). 4

Programação imperativa *Programação imperativa* é o paradigma mais antigo, já que está baseado no modelo clássico de computação de "von Neumann-Eckert". 4

Programação lógica A *programação lógica (declarativa)* permite a um programa modelar um problema declarando qual saída ele deve produzir, em vez de como ela deve ser produzida. 5

Programação múltipla Na *programação múltipla*, diversos programas seriam carregados na memória e executados de uma forma intercalada, com um escalonador sendo usado para alternar o controle de um programa para outro. 485

Programação orientada a eventos *Programas orientados a eventos* não controlam a seqüência na qual a entrada de eventos ocorre; em vez disso, eles são escritos para reagir a qualquer seqüência razoável de eventos. 447

Programação orientada a objeto A *programação orientada a objeto (POO)* fornece um modelo no qual o programa é um conjunto de objetos que interagem entre si passando mensagens que transformam seus estados. 4

Projeto por contrato *Projeto por contrato* é um *framework* formal que estabelece as assim denominadas obrigações e os benefícios (responsabilidades e direitos) entre cada par fornecedor/cliente que ocorre durante a vida de um produto de software. 540

Referência *forward* Uma referência *forward* é uma referência a um nome que ocorre antes que o nome tenha sido declarado. 91

Referência local Uma referência a um nome é *não local* se ocorrer em um escopo aninhado ao escopo da definição, caso contrário, a referência é *local*. 90

Refinamento gradual O processo do *refinamento gradual* utiliza abstração procedural para desenvolver um algoritmo a partir da sua forma mais geral, chegando a uma implementação específica. 280

Reflexão Mecanismo por meio do qual um programa pode descobrir e usar os métodos de qualquer um dos seus objetos e classes. 331

Registro de ativação Um *registro de ativação* é o bloco de informações associado à ativação de uma função, incluindo os parâmetros da função e as variáveis locais. 236

Registro variante Um *registro variante* (ou *união*) é um registro ou uma estrutura em que dois ou mais campos compartilham o mesmo bloco de memória. 121

Resolução Quando aplicadas a cláusulas de Horn, a *resolução* diz que, se *h* é o cabeçalho de uma cláusula de Horn e corresponde a um dos termos de outra cláusula de Horn, esse termo pode ser substituído por *h*. 416

Robusteza Aplicações são *robustas* quando continuam a operar sob todas as situações de erro concebíveis. 179

Seção crítica A seção de código que requer acesso exclusivo a uma variável compartilhada é chamada de seção crítica de uma *thread*. 487

Semáforo Um *semáforo* é uma variável inteira e um mecanismo associado para o enfileiramento de *threads*. 490

Semântica O significado de um programa é chamado de *semântica*. 23

Semântica de cópia *Semântica de cópia* ocorre em uma atribuição, quando uma cópia do valor da expressão fonte é atribuída à variável alvo. 163

Semântica de referência *Semântica de referência* ocorre em uma atribuição de objeto quando uma referência (ponteiro) para o valor da expressão fonte é atribuído à variável alvo. 163

Semântica denotacional *A semântica denotacional* de uma linguagem define os significados de elementos abstratos da linguagem como um conjunto de funções de transformação de estado. 198

Senão pendente Uma ambigüidade sintática na qual um "*senão*" pode ser associado com dois ou mais "*se*". 33

Serialização *Serialização* é um processo pelo qual objetos podem ser arquivados para instanciação posterior no mesmo programa ou para transmissão entre computadores via *sockets*. 172

Símbolo não-terminal Quando uma gramática BNF é usada para definir a sintaxe de linguagens de programação, os *não-terminais* N identificam categorias gramaticais da linguagem como *Identificador, Inteiro, Expressão, Comando e Programa*. 25

Símbolo terminal Os *símbolos terminais T* formam o alfabeto básico a partir do qual os programas são construídos. 25

Sintaxe A *sintaxe* de uma linguagem de programação é uma descrição precisa de todos os seus programas gramaticalmente corretos. 24

Sintaxe abstrata A *sintaxe abstrata* de uma linguagem carrega apenas as informações essenciais do programa, sem se preocupar com idiossincrasias sintáticas como pontuação ou parêntese. 24

Sintaxe concreta A *sintaxe concreta* de uma linguagem se refere à representação real do seu programa, usando símbolos léxicos como seu alfabeto. 24

Sintaxe léxica A *sintaxe léxica* de uma linguagem define as regras para os símbolos básicos, incluindo identificadores, literais (por exemplo, números inteiros e de ponto flutuante), operadores e pontuação. 24

Sistema de tipo Um *sistema de tipo* é uma definição precisa das ligações entre o tipo de uma variável, seus valores e as operações possíveis sobre esses valores. 103

Sistemas legados *Sistemas legados* são aqueles artefatos de software que foram projetados e implementados pela equipe anterior de programação, mas que cuja manutenção e atualização são feitas pela equipe atual. 14

Sobrecarga A *sobrecarga* usa o número ou tipo de argumentos para distinguir entre nomes idênticos de funções ou operadores. 96

Sobrecarga de operador Um *operador* ou uma função é denominado *sobrecarregado* quando seu significado variar dependendo do tipo dos seus operandos ou argumentos ou resultados. 108

Socket Um *socket* é um ponto extremo de uma conexão dedicada de comunicação entre dois programas sendo executados em uma rede. 507

Subtipo Um *subtipo* pode ser visto como um tipo que possui determinadas restrições colocadas nos seus valores ou em suas operações. 126

Tabela de símbolos Uma *tabela de símbolos* é uma estrutura de dados mantida por um tradutor, que permite a ele registrar cada nome declarado e suas ligações. 92

Tempo de vida O *tempo de vida* de uma variável é o intervalo de tempo no qual essa variável possuiu um bloco de memória alocado para ela. 98

Tipo Um *tipo* é um conjunto de valores e de operações sobre esses valores. 102

Tipo união Em linguagens do tipo C, um tipo *união* realiza o objetivo de um registro variante, já que permite duas ou mais formas de visualização do mesmo local de armazenamento. 122

Token Um *token* é uma seqüência logicamente coesa de caracteres que representam um único símbolo. 60

Transparência referencial Uma função possui *transparência referencial* se o seu valor depender apenas dos valores dos seus parâmetros. 363

União de substituição A *união de substituição* de X e Y, escrita $X \ddot{U} Y$, é o resultado da substituição em X de todos os pares $<x,v>$ cujo primeiro membro relacione um par $<x,w>$ de Y por $<x,w>$ e então adicione a *X* todos os pares restantes em Y. 215

Unificação *Unificação* é um processo de relacionamento de padrões que determina quais instanciações específicas podem ser feitas a variáveis durante a produção de uma série de resoluções simultâneas. 417

Validação Um programa é *válido* se as suas declarações de variáveis globais e funções forem válidas e cada uma das suas funções (incluindo *main*) for válida quanto ao seu mapa de tipo. 254

Valor L O *valor L* de uma variável é o uso do seu nome para denotar seu endereço, e no seu uso no lado esquerdo de uma atribuição. 88

Valor R O *valor R* de uma variável é o uso do seu nome para denotar seu valor, como no seu uso no lado direito de uma atribuição. 88

Variável Uma *variável* é uma ligação de um nome a um endereço de memória. Além disso, uma variável possui um tipo, um valor e um tempo de vida. 88

Variável ligada Uma variável ligada é uma variável cujo nome é igual ao nome do parâmetro; caso contrário, dizemos que a variável é livre (free) 364

Verificação de tipo A detecção de erros de tipo, em tempo de compilação ou em tempo de execução, é chamada de *verificação de tipo*. 135

Vetor *dope* Um *vetor dope* de uma matriz é um bloco de informações usadas para facilitar verificações como faixa de índices entre outras. 116

Visibilidade de nome Um nome é *visível* a uma referência se a sua referenciação incluir essa referência e o nome não for redeclarado em um escopo mais interno. 95

Bibliografia

ADAMS, J. C. et al. (1997). *Fortran 95 HandBook: Complete ISO/ANSI Reference*. MIT Press.

AHERN, D. et al. (2004). *CMMI Distilled: A Practical Approach to Integrated Process Improvement*. Addison-Wesley.

AHO, A. V., SETHI, R. e ULLMAN J. D. (1986). *Compilers, Principles, Techniques, and Tools*. Addison-Wesley.

ALVES-FOSS, J. (ed.) (1999). *Formal Syntax and Semantics of Java*. Lecture Notes in Computer Science, 1523. Springer.

ANDREWS, G. R. e OLSSON, R. A. (1993). *The SR Programming Language*. Benjamin Cummings.

APPEL, A. W. (1998). *Modern Compiler Implementation in Java*. Cambridge University Press.

ARNOLD, K. e GOSLING , J. (1998). *The Java Programming Language*. Addison-Wesley.

BACKUS, J. et al. (1954). *Preliminary Report: Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN*. IBM Corporation.

BARNES, J. (1997). *High Integrity Ada: The Spark Approach*. Addison-Wesley.

BARNES, J. (2003). *High Integrity Software: The Spark Approach to Safety and Security*. Addison-Wesley.

BEN-ARI, M. (1994). *Principles of Concurrent Programming*. Prentice-Hall International.

BLAHA, M. e RUMBAUGH, J. (2005). *Object-Oriented Modeling and Design with UML*. Prentice-Hall, 2ª edição.

BÖHM, C. e JACOPINI, G. (1996). Flow diagrams, Turing machines, and languages with only two formation rules. *Communications of ACM* 29 (Junho 1966), p. 471–483.

BROWN, G. D. (1977). *Advanced ANS Cobol with Structured Programming*.Wiley-Interscience.

BUDD, T. A. (1995). *Multiparadigm Programming in Leda*. Addison-Wesley.

BUDD, T. (2000). *Understanding Object-Oriented Programming with Java*. Addison-Wesley.

CARRIERO, N. e GELENTER, D. (1989). Linda in context. *Communications of ACM* 32 (Abril 1989), p. 444–458.

CC2001 (2001). *Computing Curriculum 2001: Computer Science*. ACM/IEEE Joint Task Force on Computing Curricula.

CHAMBERLAND, L. (1995). *Fortran 90 A Reference Guide*. Addison-Wesley.

CHOMSKY, N. (1957). *Syntactic Structures*. Mouton.

CHURCH, A. (1941). *The Calculi of Lambda Conversion*. Princeton University Press.

CLOCKSIN,W. F. e MELLISH, C. S. (1997). *Programming in Prolog*. Springer, 4ª edição.

COFFMAN, E. G., ELPHNICK, M. J. e SHOSANI, A. (1971). System deadlocks. *Computing Surveys* 3 (Junho 1971) p. 67–68.

COHEN, N. H. (1996). *Ada as a Second Language*. McGraw-Hill, 2ª edição.

COOPER, D. e CLANCY Michael (1985). *Oh! Pascal!* W.W. Norton & Company, 2ª edição.

CROXFORD, M. (2005). The challenge of low defect, secure software–too difficult and too expensive? *DOD Software Tech News* 82, (Julho 2005), p. 8–10.

Department of Defense (1983). Military standard Ada programming language. ANSI/MIL-STD-1815A (Fevereiro).

DERSHEM, H. L. e JIPPING, M. J. (1990). *Programming Languages: Structures and Models*. Wadsworth.

DIJKSTRA, E.W. (1965). Solution of a problem in concurrent control. *Communications of ACM* 8 (Setembro 1965), p. 569.

DIJKSTRA, E. W. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*. Academic Press, 1968, p. 43–112.

DIJKSTRA, E. W. Go to statement considered harmful. *Communications of ACM* 11, Março 1968, p. 147–148.

DIJKSTRA, E. W. Hierarchical ordering of sequential processes. *Acta Informatica* 1, 1971, p. 115–138.

DIJKSTRA, E. W. (1972). "Notes on structured programming." In O.-J. Dahl, E. W. Dijkstra e C. A. R. Hoare, editors, *Structured Programming*. Academic Press.

DYBVIG, R. Kent (1996). *The Scheme Programming Language, ANSI Scheme*. Prentice-Hall.

ELLIS, M. A. e STROUSTRUP B. (1990). *The Annotated C++ Reference Manual*. Addison-Wesley.

FLANAGAN, C., RUSTAN, K., LEINO, M., LILLIBRIDGE M., NELSON, Greg, J. NELSON, B. SAXE e STATA, R. (2002). Extended static checking for Java. In *Conference on Programming Language Design and Implementation*. ACM SIGPLAN (Junho 2002), p. 234–235.

FLANAGAN, D. (1996). *Java in a Nutshell*. O'Reilly.

FOWLER, M. (2000). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

FUCHI, K., KOWALSKI, R., FURUKAWA K., UEDA, K., CHIAYAMA, T. e TICK E. (1993). Launching the new era. *Communications of ACM* 3 (Março 1993) p. 49–100.

GAMMA, E., HELM, R., JOHNSON, R. e VLISSIDES J. (1995). *Design Patterns*. Addison-Wesley.

GOLDBERG, A. e ROBSON, D. (1989). *Smalltalk-80 the Language*. Addison-Wesley.

GOLDBERG, D. (1991). What every computer scientist should know about floating-point arithmetic. *Computing Surveys* 23 (Março 1991), p. 5–48.

GOSLING, J., JOY, B. e STEELE, G. (1996). *The Java Language Specification*. Addison-Wesley.

GRIES, David (1981). *The Science of Programming*. Springer.

GUZDIAL, M. e ROSE, K. (2000). *Squeak: Open Personal Computing and Multimedia*. Prentice-Hall.

HALL, A. e CHAPMAN, R. (2002). Correctness by construction: Developing a commercial secure system. *IEEE Software* (Janeiro 2002), p. 18–25.

HENNESSY, J. e PATTERSON, D. A. (1998). *Computer Organization and Design: The Hardware/-Software Interface*. Morgan Kaufmann.

HOARE, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of ACM* 12 (Outubro 1969), p. 576–580, 583.

HOARE, C. A. R. (1974). An operating system structuring concept. *Communications of ACM* 17 (Outubro 1974), p. 549–557.

HOARE, C. A. R. (1978). Communicating sequential processes. *Communications of ACM* 21 (Agosto 1978), p. 666–677.

HOARE, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall International.

HOPCROFT, J. E. e ULLMAN, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

HORSTMANN, C. (2004). *Object-Oriented Design & Patterns*. JohnWiley.

HUDAK, P. (2000). *The Haskell School of Expression*. Cambridge University Press.

HUGHES, J. (1989). Why functional programming matters. *The Computer Journal* 32, 2 (1989), p. 98–107.

HUGHES, S. (2001). *PHP Developer's Cookbook*. SAMS.

IVERSON, K. E. (1962). *A Programming Language*. JohnWiley.

JENSEN, K. e WIRTH, N. (1975). *Pascal User Manual*. Springer, 2ª edição.

JONES, R. e LINS, R. (1996). *Garbage Collection*. JohnWiley.

JOSUTTIS, N. (1999). *The Standard C++ Library*. Addison-Wesley.

KELSEY, R. CLINGER, W. e REES, J. (ed.) (1998). Revised report on the algorithmic language Scheme. *SIGPLAN Notices* 33, 9 (Setembro 1998), p. 26–27.

KERNIGHAN, B. W. e RITCHIE, D. M. (1978). *The C Programming Language*. Prentice-Hall.

KERNIGHAN, B. W. e PIKE, R. (1984). *The Unix Programming Environment*. Prentice-Hall.

KERNIGHAN, B. W. e RITCHIE, D. M. (1988). *The C Programming Language*. Prentice-Hall, 2ª edição.

KNUTH, D. E. (1967). The remaining trouble spots in ALGOL 60. *CACM* 10 (Outubro 1967), p. 611–618.

KNUTH, D. E. (1971). Top down syntax analysis. *Acta Informatica* 1, 2 (1971), p. 79–110.

KOWALSKI, R. e KUEHNER, D. (1970). Resolution with selection function. *Artificial Intelligence* 3, 3 (1970), p. 227–260.

KOWALSKI, R. A. The early years of logic programming. *Communications of ACM* 31 (Janeiro 1988), p. 38–43.

LEAVENS, G. T., BAKER, A. L. e RUBY, C. JML:AJava modeling language. In *Formal Underpinnings of Java Workshop*, Outubro, 1998.

LEAVENS, G. e CHEON, Y. (2004). Design by contract with JML: An on-line tutorial. http://www.cs.iastate.edu/leavens/JML/teaching.shtml.

LINDHOLM, Tim e YELLIN, F. (1997). *The Java Virtual Machine Specification*. Addison-Wesley.

LISKOV, B.e SNYDER, A. Exception handling in CLU. *IEEE Transactions on Software Engineering,* SE-5 (1979), p. 546–558.

LISKOV, B. et al. (1981). *CLU Reference Manual*. Springer.

LISKOV, B. e GUTTAG, J. (2001). *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley.

LUTZ, M. e ASCHER, D. (1999). *Learning Python*. O'Reilly.

LUTZ, M. (2001). *Programming Python*. O'Reilly, 2ª edição.

MASON, T., LEVINE, J. e BROWN, D. *lex & yacc*. O'Reilly, 1992, 2ª edição.

Mccarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM* 3 (Abril 1960), p. 184–185.

MCCARTHY, J. et al. (1965). *Lisp 1.5 Programmer's Manual*. MIT Press.

MEYER, B. (1988). *Object-Oriented Software Construction*. Prentice-Hall.

MEYER, B. (1992). *Eiffel: The Language*. Prentice-Hall.

MEYER, B. (1997). *Object-Oriented Software Construction*. Prentice-Hall, 2ª edição.

DE MILLO, R. A., LIPTON, R. J. e PERLIS, A. J. . Social processes and proofs of theorems and programs. *Communications of ACM* 22, Maio 1979, p. 271–280.

MITCHELL, J. G.,W. MAYBURY e R. SWEET (1979). *Mesa Language Manual, Version 5*. Xerox Palo Alto Research Center.

NAUR, P. (ed.) Revised report on the algorithmic language Algol 60. *Communications of ACM* 6 Janeiro 1963, p. 1–23.

NIEMEYER, P., e Jonathan KNUDSEN (2002). *Learning Java*. O'Reilly, 2ª edição.

NOONAN, R. E. (2000). An object-oriented view of backtracking. In *Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education*, Março 2000, p. 362–366.

OUSTERHOUT, J. K. (1994). *Tcl and the Tk Toolkit*. Addison-Wesley.

OWRE, S., J. M. RUSHBY e N. SHANKAR*Lecture Notes in Artificial Intelligence*, volume 607, capítulo PVS: A Prototype Verification System. Springer 1992, p. 748–752.

PATIL, S. *Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes*. MIT. Fevereiro 1971, Technical report.

PRATT, P. J. (1990). *A Guide to SQL*. Boyd & Fraser.

ROBINSON, J. A. A machine-oriented logic based on the resolution principle. *Journal ACM* 12 ,1965, p. 23–41.

RUBIN, F. goto statement considered harmful. *Communications of ACM* 30. Março 1987, p. 195–196.

SCHWARTZ, R. L. (1993). *Learning Perl*. O'Reilly. 590 Bibliography

SHAPIRO, E.Y. (1983). The fifth generation project – a report. *Communications ofACM* 26 Setembro 1983, p. 637–641.

STEELE, G. L. (1990). *Common Lisp – The Language*. Digital Press, Cambridge, MA. 2ª edição.

STEIN, L. A. (1999). Challenging the computational metaphor: implications for how we think. *Cybernetics and Systems* 30 Setembro 1999, p. 30.

STROUSTRUP, B. (1991). *The C++ Programming Language*. Addison-Wesley, 2ª edição.

STROUSTRUP, B.(1994). *The design and evolution of C++*. Addison-Wesley.

STROUSTRUP, B.(1997). *The C++ Programming Language*. Addison-Wesley, 3ª edição.

TENNENT, R. D. (1981). *Principles of Programming Languages*. Prentice-Hall International.

THOMPSON, S. (1999). *Haskell: the Craft of Functional Programming*. Addison-Wesley, 2ª edição.

TUCKER, A. e Robert NOONAN (2002). *Programming Languages: Principles and Practices*. McGraw-Hill.

TURING, A. M On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* 2. 1936, p. 265. A correction: 43, 544–546.

ULLMAN, J. D. (1989). *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Volume II.

ULLMAN, J. D. (1998). *Elements of ML Programming*. Prentice-Hall, 197ª edição.

Unicode Consortium (2000). *The Unicode Standard Version 3.0*. Addison-Wesley. See also http://www.unicode.org.

van den BERG, J. e JACOBS, B. The LOOP compiler for Java and JML. In T. Magaria e W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, 2031. Springer 2001, p. 299–312.

van WIJNGAARDEN, A. (ed.), MAILLOUX, B. J., PECK, J. E. L. e KOSTER, C. H. A. Report on the algorithmic language ALGOL 68. *Numerische Mathematik*, 14 (1969), p. 79–218.

WAITE, W. D. e L. R. CARTER (1993). *An Introduction to Compiler Construction*. HarperCollins.

WALEY, J. e M. S. LAM (2004). Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Conference on Programming Language Design and Implementation*. ACM SIGPLAN (Junho 2004), p. 131–144.

WALKER, H. M. e G. M. SCHNEIDER (1996). Revised model curriculum for a liberal arts degree in computer science. *Communications of the ACM* 39 (Dezembro 1996), p. 85–95.

WALL, L., Tom CHRISTIANSEN e Randal L. SCHWARTZ (1996). *Programming Perl*. O'Reilly, 2ª edição.

WARMER, J. e A. KLEPPE (1999). *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley.

WIRTH, N. (1973). *Systematic Programming: an Introduction*. Prentice-Hall.

WIRTH, N. (1974). On the design of programming languages. In *Proceedings of IFIP Congress 74* (1974), p. 386–393.

WIRTH, N. (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall.

WIRTH, N. (1977). What can we do about the unnecessary diversity of notation for syntactic definitions? *CACM* 20 (Novembro 1977), p. 822–823.

WIRTH, N. (1982). *Programming in Modula-2*. Springer, 2ª edição.

Índice Remissivo