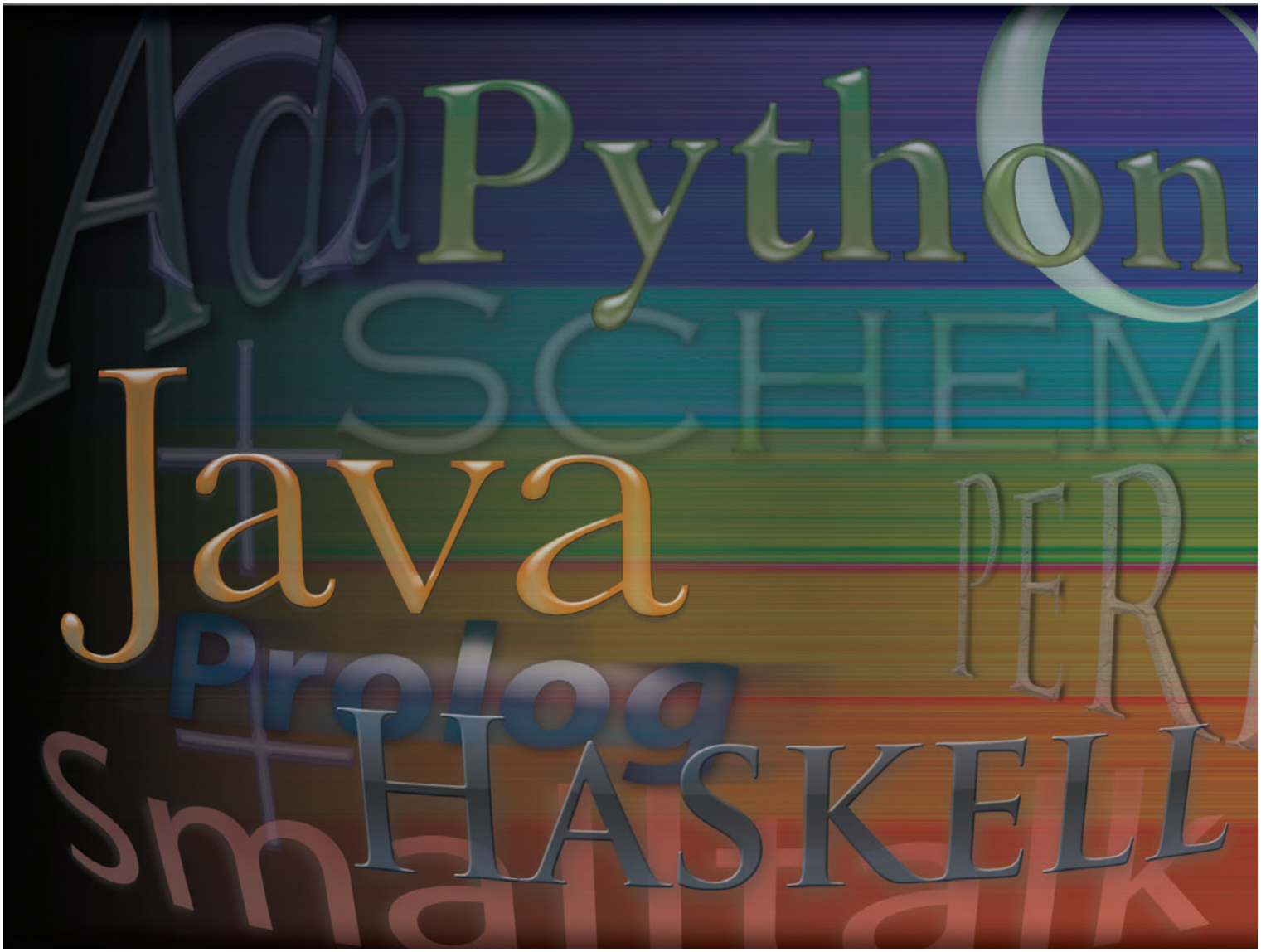
Linguagens de Programação

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

T891 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

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

12.1 O QUE TORNA UMA LINGUAGEM IMPERATIVA? 278
12.2 ABSTRAÇÃO PROCEDURAL 280
12.3 EXPRESSÕES E ATRIBUIÇÃO 281
12.4 SUPORTE DE BIBLIOTECA PARA ESTRUTURAS DE DADOS 283
12.5 PROGRAMAÇÃO IMPERATIVA E C 284
12.6 PROGRAMAÇÃO IMPERATIVA E ADA 290
12.7 PROGRAMAÇÃO IMPERATIVA E PERL 296
12.8 RESUMO 307
EXERCÍCIOS 307

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.

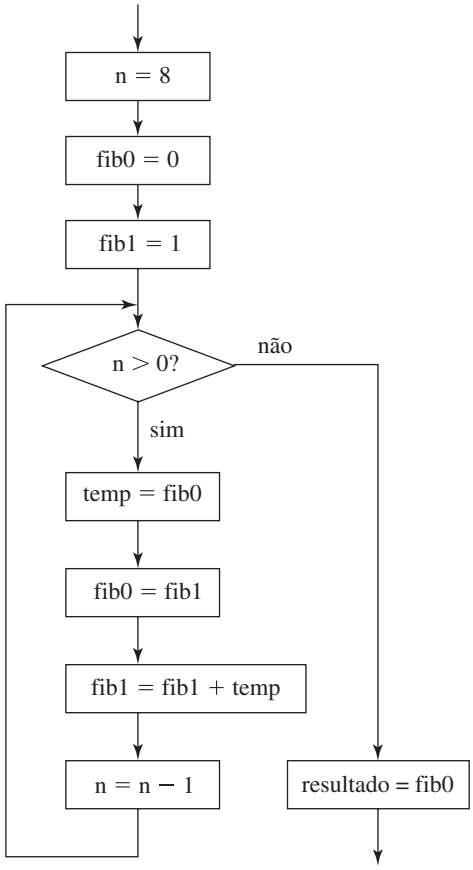


Figura 12.1
Exemplo de Diagrama de Fluxo para Calcular Números de Fibonacci

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 opl, 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(l) 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.

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

14.1	FUNÇÕES E O CÁLCULO LAMBDA	362
14.2	SCHEME	366
14.3	HASKELL	388
14.4	RESUMO	408
EXERCÍCIOS		408

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.

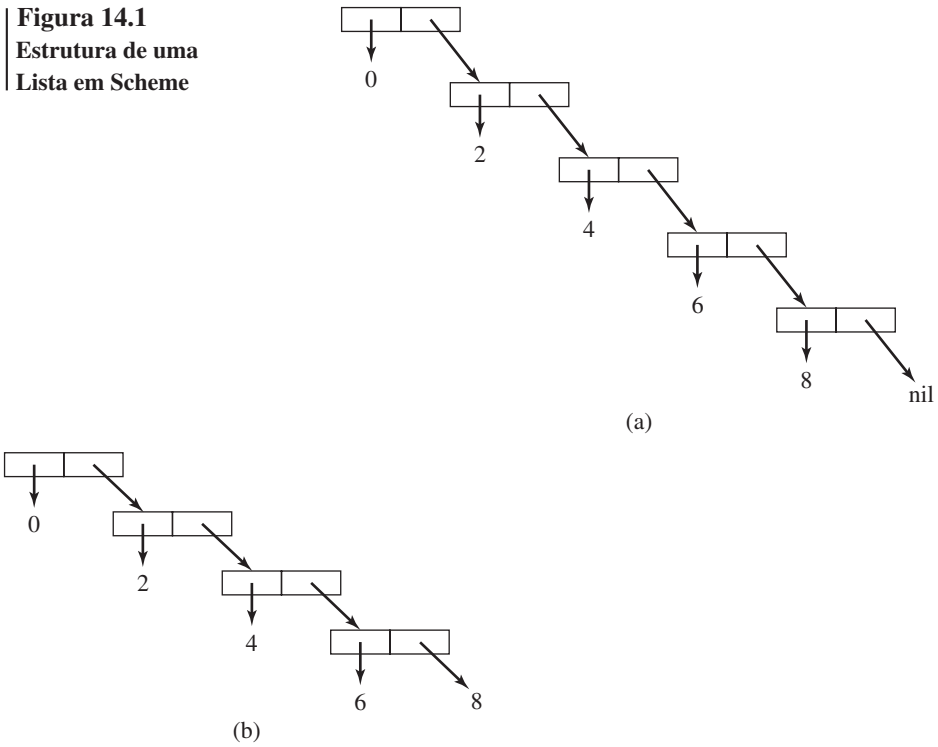


Figura 14.1
Estrutura de uma Lista em Scheme

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))
```

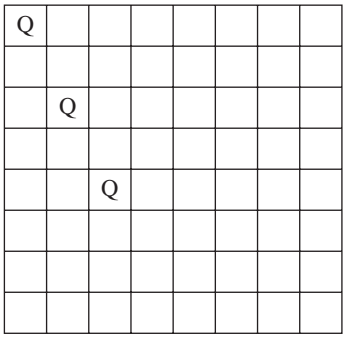


Figura 14.3
Três Rainhas em um Tabuleiro de Xadrez 8 × 8

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 (λ*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

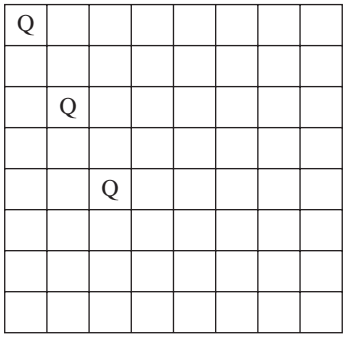


Figura 14.5
Três Rainhas em um Tabuleiro de Xadrez 8 × 8

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:

$$M(Block\ \text{b}, State\ \sigma) = \sigma \qquad \text{se b} = \phi$$
$$= M((Block)b_{2\ldots n}, M((Statement)b_1, \sigma)) \qquad \text{se b} = b_1 b_2 \ldots b_n$$

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.

15 Programação Lógica

"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

15.1 LÓGICA E CLÁUSULAS DE HORN 414
15.2 PROGRAMAÇÃO LÓGICA EM PROLOG 417
15.3 EXEMPLOS PROLOG 430
15.4 RESUMO 443
EXERCÍCIOS 443

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 $p_1, p_2, \ldots$ 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:

$\texttt{term:-term}_1\texttt{, term}_2\texttt{,..., term}_n\texttt{.}$

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.

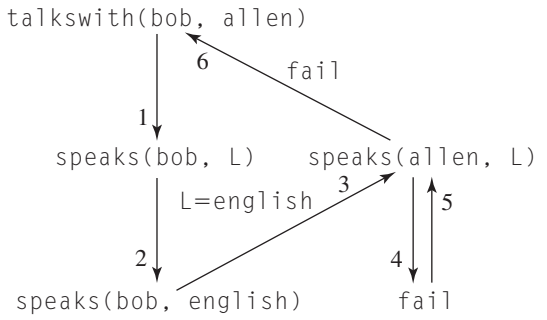
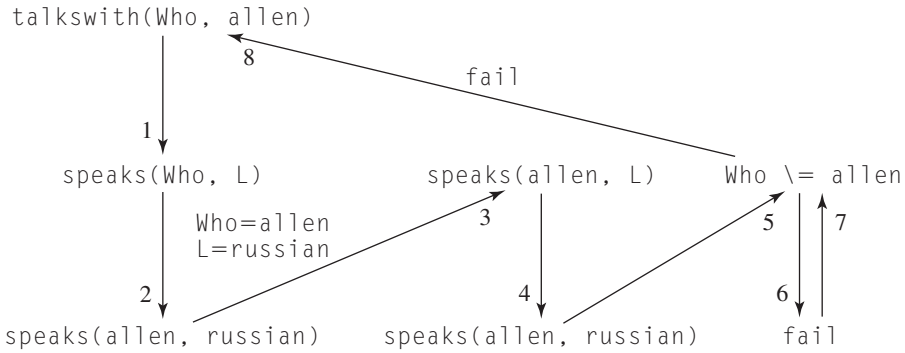


Figura 15.1
Tentando Satisfazer a Query `talkswith (bob, allen)`



| 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

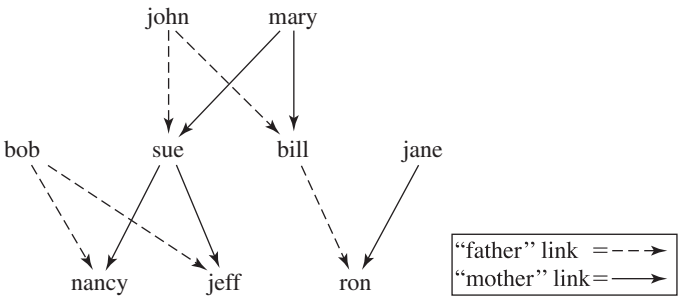
na árvore e "avô" (grandparent) relaciona duas pessoas que estão distantes entre si dois níveis.

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).
```

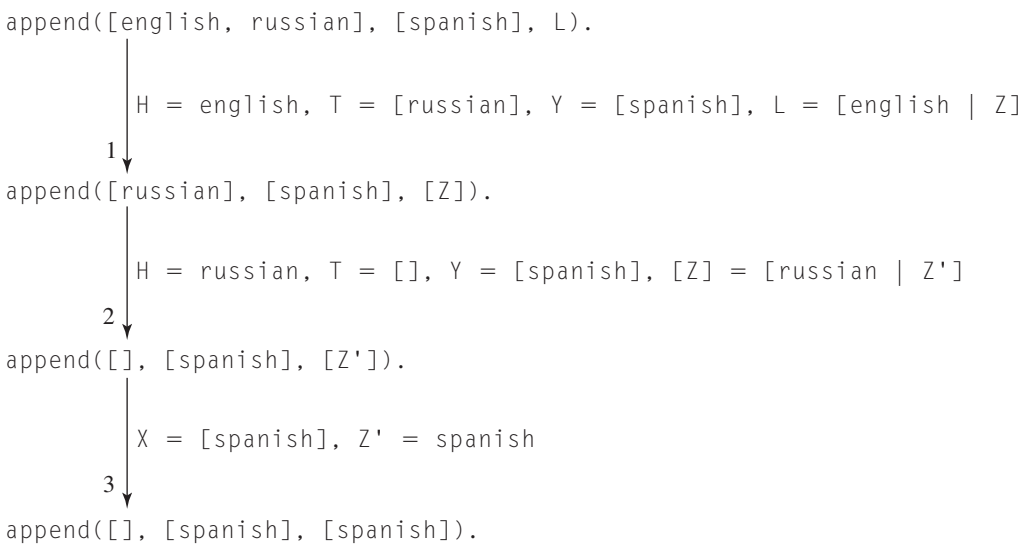
Essa definição repetitiva um tanto estranha tem duas partes. O "base case" é definido pela primeira linha, que simplesmente afirma que a lista vazia encadeada com qualquer outra lista retorna aquela outra lista como resultado. O case repetitivo, definido pela segunda linha, diz que se `Z` é o resultado 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.

$$\frac{dc}{dx} = 0 \qquad c \text{ é uma constante}$$

$$\frac{dx}{dx} = 1$$

$$\frac{d}{dx}(u + v) = \frac{du}{dx} + \frac{dv}{dx} \qquad u \text{ e } v \text{ são funções de } x$$

$$\frac{d}{dx}(u - v) = \frac{du}{dx} - \frac{dv}{dx}$$

$$\frac{d}{dx}(uv) = u\frac{dv}{dx} + v\frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{u}{v}\right) = \left(v\frac{du}{dx} - u\frac{dv}{dx}\right) \Big/ v^2$$

| Figura 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

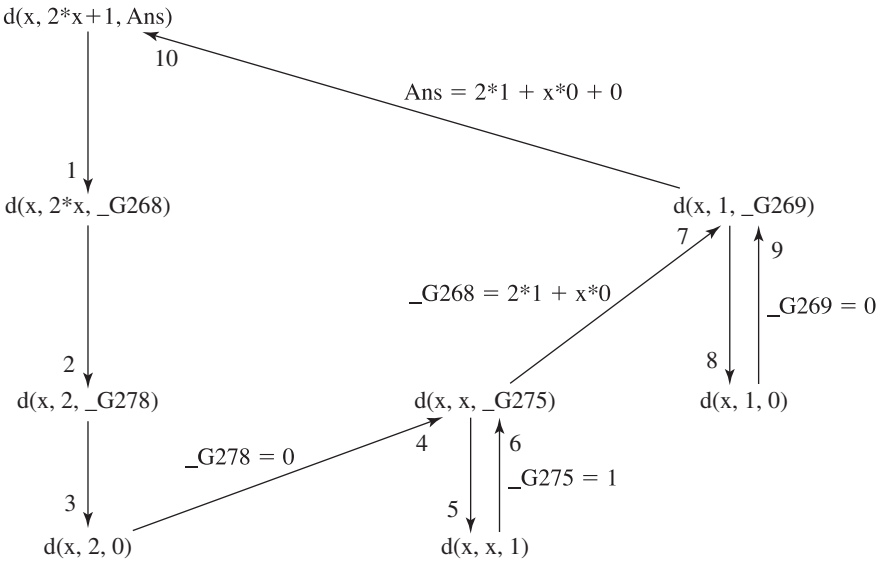
```
?- d(x, 2*x, 2).
```

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.

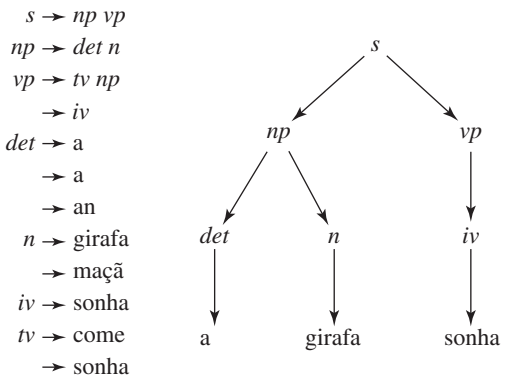
```
print_floors([A | B]) :- write(A), nl, print_floors(B).
print_floors([]).

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**).

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

17.1 CONCEITOS DE CONCORRÊNCIA 484
17.2 ESTRATÉGIAS DE SINCRONIZAÇÃO 490
17.3 SINCRONIZAÇÃO EM JAVA 494
17.4 COMUNICAÇÃO INTERPROCESSOS 506
17.5 CONCORRÊNCIA EM OUTRAS LINGUAGENS 513
17.6 RESUMO 515
EXERCÍCIOS 516

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

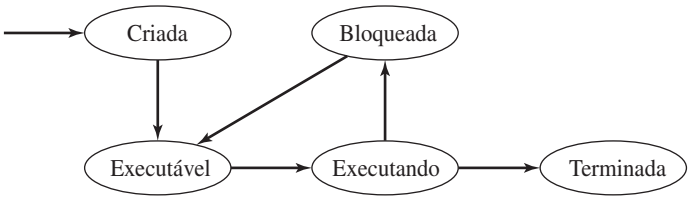
Tanto em Java quanto em Ada, uma *thread* está associada a um método separado (função), em vez de a uma única instrução. No entanto, para iniciar uma *thread* são necessárias ações adicionais, além de chamar uma função. Primeiro, quem chama uma nova *thread* não espera que ela se complete antes de continuar; 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.

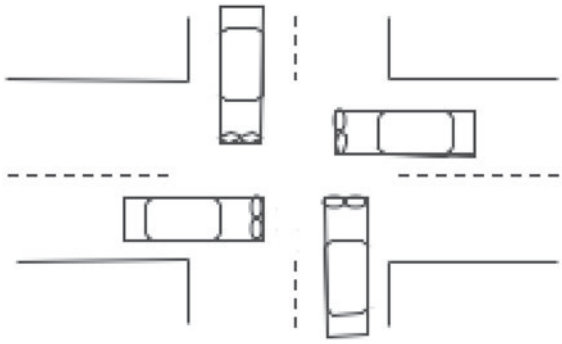
De fato, à medida que aumenta o número de *threads* tentando executar esse código, o número resultante de valores distintos computados para `c` pode variar entre 1 e o número de *threads*! Essa é uma situação computacional inaceitável.

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`:

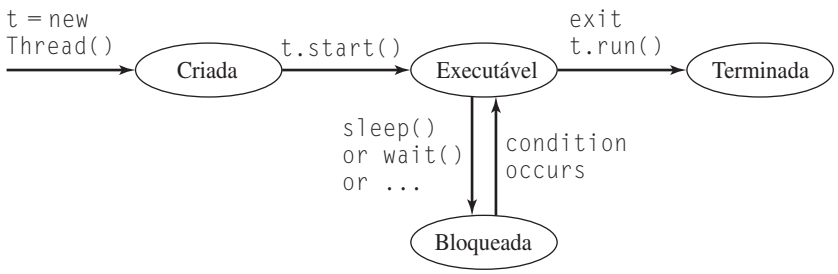
```
Thread thread = new MyThread( );
```

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.

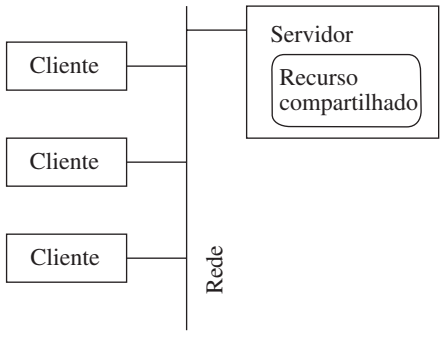
Servidores são computadores poderosos ou processos dedicados ao gerenciamento de bases de dados (servidores de arquivo), impressoras (servidores de impressão) ou de tráfego de rede (servidores de rede). Clientes são PCs ou estações de trabalho (*workstations*) nas quais os usuários rodam aplicativos. 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.

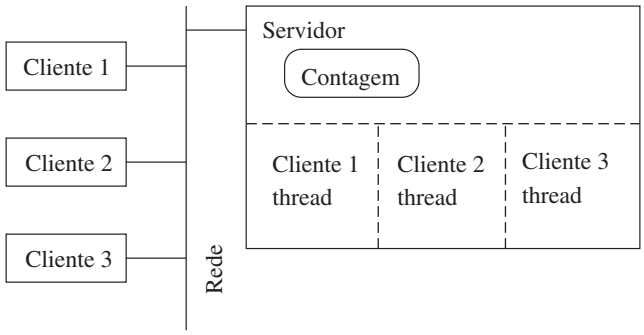


Figura 17.20 Arquitetura Cliente-Servidor para Contagem de Votos

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.