Capítulo 1

Aspectos Preliminares

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

Como este livro trata também da implementação de linguagens de programação, este capítulo inclui uma visão geral das abordagens mais comuns para a implementação. Finalmente, descrevemos alguns exemplos de ambientes de programação e discutimos seu impacto na produção de software.

1.1 RAZÕES PARA ESTUDAR CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO

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

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

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

  Pode-se argumentar que aprender os recursos de outras linguagens não ajuda um programador forçado a usar uma determinada linguagem que não tem tais recursos. Esse argumento não se sustenta,

porque normalmente as construções das linguagens podem ser simuladas em outras que não as oferecem suporte diretamente. Por exemplo, um programador C que aprendeu a estrutura e os usos de matrizes associativas em Perl (Wall et al., 2000) pode projetar estruturas que as simulem em C. Em outras palavras, o estudo de conceitos de linguagens de programação constrói uma apreciação de recursos e construções valiosas das linguagens e incentiva os programadores a usá-las, mesmo quando a linguagem utilizada não oferece suporte direto para esses recursos e construções.

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

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

- *Habilidade aumentada para aprender novas linguagens*. A programação de computadores ainda é uma disciplina relativamente nova e as metodologias de projeto, ferramentas de desenvolvimento de software e linguagens de programação ainda estão em evolução. Isso torna o desenvolvimento de software uma profissão excitante, mas também exige aprendizado contínuo. O processo de aprender uma nova linguagem de programação pode ser longo e difícil, especialmente para alguém que esteja confortável com apenas uma ou duas e nunca examinou os conceitos de linguagens de programação de um modo geral. Uma vez que um entendimento preciso dos conceitos fundamentais das linguagens tenha sido adquirido, fica mais fácil ver como esses conceitos são incorporados no projeto da linguagem aprendida. Por exemplo, programadores que entendem os conceitos de programação orientada a objetos aprenderão

Java muito mais facilmente (Arnold et al., 2006) do que aqueles que nunca usaram tais conceitos.

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

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

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

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

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

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

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

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

  Por exemplo, muitas pessoas acreditam que teria sido melhor se o ALGOL 60 (Backus et al., 1963) tivesse substituído o Fortran (Metcalf et al., 2004) no início dos anos 1960, porque ele era mais elegante e tinha sentenças de controle muito melhores do que o Fortran, dentre outras razões. Ele não o substituiu, em parte por causa dos programadores e dos gerentes de desenvolvimento de software da época; muitos não entendiam o projeto conceitual do ALGOL 60. Eles achavam sua descrição difícil de ler (o que era verdade) e mais difícil ainda de entender. Eles não gostaram das vantagens da estrutura de blocos, da recursão e de estruturas de controle bem estruturadas, então falharam em ver as melhorias do ALGOL 60 ao relação ao Fortran.

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

  Em geral, se aqueles que escolhem as linguagens fossem mais bem informados, talvez linguagens melhores ganhassem de outras.

1.2 DOMÍNIOS DE PROGRAMAÇÃO

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

1.2.1 Aplicações científicas

Os primeiros computadores digitais, que apareceram nos anos 1940, foram inventados e usados para aplicações científicas. Normalmente, aplicações científicas têm estruturas de dados relativamente simples, mas requerem diversas computações de aritmética de ponto flutuante. As estruturas de dados mais comuns são os vetores e matrizes; as estruturas de controle mais comuns são os laços de contagem e as seleções. As primeiras linguagens de programação de alto nível inventadas para aplicações científicas foram projetadas para suprir tais necessidades. Sua competidora era a linguagem *assembly* – logo, a eficiência era uma preocupação primordial. A primeira linguagem para aplicações científicas foi o Fortran. O ALGOL 60 e a maioria de seus descendentes também tinham a intenção de serem usados nessa área, apesar de terem sido projetados também para outras áreas relacionadas. Para aplicações científicas nas quais a eficiência é a principal preocupação, como as que eram comuns nos anos 1950 e 1960, nenhuma linguagem subsequente é significativamente melhor do que o Fortran, o que explica por que o Fortran ainda é usado.

1.2.2 Aplicações empresariais

O uso de computadores para aplicações comerciais começou nos anos 1950. Computadores especiais foram desenvolvidos para esse propósito, com linguagens especiais. A primeira linguagem de alto nível para negócios a ser bem-sucedida foi o COBOL (ISO/IEC, 2002), com sua primeira versão aparecendo em 1960. O COBOL ainda é a linguagem mais utilizada para tais aplicações. Linguagens de negócios são caracterizadas por facilidades para a produção de relatórios elaborados, maneiras precisas de descrever e armazenar números decimais e caracteres, e a habilidade de especificar operações aritméticas decimais.

Poucos avanços ocorreram nas linguagens para aplicações empresariais fora do desenvolvimento e evolução do COBOL. Logo, este livro inclui apenas discussões limitadas das estruturas em COBOL.

1.2.3 Inteligência Artificial

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

A primeira linguagem de programação amplamente utilizada desenvolvida para aplicações de IA foi a linguagem funcional LISP (McCarthy et al.,

1965), que apareceu em 1959. A maioria das aplicações de IA desenvolvidas antes de 1990 foi escrita em LISP ou em uma de suas parentes próximas. No início dos anos 1970, entretanto, uma abordagem alternativa a algumas dessas aplicações apareceu – programação lógica usando a linguagem Prolog (Clocksin e Mellish, 2003). Mais recentemente, algumas aplicações de IA têm sido escritas em linguagens de sistemas como C. Scheme (Dybvig, 2003), um dialeto de LISP, e Prolog são introduzidas nos Capítulos 15 e 16, respectivamente.

1.2.4 Programação de sistemas

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

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

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

1.2.5 Software para a Web

A *World Wide Web* é mantida por uma eclética coleção de linguagens, que vão desde linguagens de marcação, como XHTML, que não é de programação, até linguagens de programação de propósito geral, como Java. Dada a necessidade universal por conteúdo dinâmico na Web, alguma capacidade de computação geralmente é incluída na tecnologia de apresentação de conteúdo. Essa funcionalidade pode ser fornecida por código de programação embarcado em um documento XHTML. Tal código é normalmente escrito com uma linguagem de *scripting*, como JavaScript ou PHP. Existem também algumas linguagens similares às de marcação que têm sido estendidas para incluir construções que controlam o processamento de documentos, as quais são discutidas na Seção 1.5 do Capítulo 2.

1.3 CRITÉRIOS DE AVALIAÇÃO DE LINGUAGENS

Conforme mencionado, o objetivo deste livro é examinar os conceitos fundamentais das diversas construções e capacidades das linguagens de programação. Iremos também avaliar esses recursos, focando em seu impacto no processo de desenvolvimento de software, incluindo a manutenção. Para isso, precisamos de um conjunto de critérios de avaliação. Tal lista é necessariamente controversa, porque é praticamente impossível fazer dois cientistas da computação concordarem com o valor de certas características das linguagens em relação às outras. Apesar dessas diferenças, a maioria concordaria que os critérios discutidos nas subseções a seguir são importantes.

Algumas das características que influenciam três dos quatro critérios mais importantes são mostradas na Tabela 1.1, e os critérios propriamente ditos são discutidos nas seções seguintes[1]. Note que apenas as características mais importantes são incluídas na tabela, espelhando a discussão nas subseções seguintes. Alguém poderia argumentar que, se fossem consideradas características menos importantes, praticamente todas as posições da tabela poderiam ser marcadas.

Note que alguns dos critérios são amplos – e, de certa forma, vagos – como a facilidade de escrita, enquanto outros são construções específicas de linguagens, como o tratamento de exceções. Apesar de a discussão parecer implicar que os critérios têm uma importância idêntica, não é o caso.

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

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

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

1.3.1 Legibilidade

Um dos critérios mais importantes para julgar uma linguagem de programação é a facilidade com a qual os programas podem ser lidos e entendidos. Antes de 1970, o desenvolvimento de software era amplamente pensado em termos da escrita de código. As principais características positivas das linguagens de programação eram a eficiência e a legibilidade de máquina. As construções das linguagens foram projetadas mais do ponto de vista do computador do que do dos usuários. Nos anos 1970, entretanto, o conceito de ciclo de vida de software (Booch, 1987) foi desenvolvido; a codificação foi relegada a um papel muito menor, e a manutenção foi reconhecida como uma parte principal do ciclo, particularmente em termos de custo. Como a facilidade de manutenção é determinada, em grande parte, pela legibilidade dos programas, a legibilidade se tornou uma medida importante da qualidade dos programas e das linguagens de programação, o que representou um marco na sua evolução. Ocorreu uma transição de foco bem definida: da orientação à máquina à orientação às pessoas

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

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

1.3.1.1 Simplicidade geral

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

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

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

Apesar de as últimas duas sentenças terem significados um pouco diferentes uma da outra e em relação às duas primeiras em alguns contextos, as quatro

têm o mesmo significado quando usadas em expressões isoladas. Essas variações são discutidas no Capítulo 7.

Um terceiro problema em potencial é a **sobrecarga de operadores**, na qual um operador tem mais de um significado. Apesar de ser útil, pode levar a uma redução da legibilidade se for permitido aos usuários criar suas próprias sobrecargas e eles não o fizerem de maneira sensata. Por exemplo, é aceitável sobrecarregar o operador + para usá-lo tanto para a adição de inteiros quanto para a adição de valores de ponto flutuante. Na verdade, essa sobrecarga simplifica uma linguagem ao reduzir o número de operadores. Entretanto, suponha que o programador definiu o símbolo + usado entre vetores de uma única dimensão para significar a soma de todos os elementos de ambos os vetores. Como o significado usual da adição de vetores é bastante diferente, isso tornaria o programa mais confuso, tanto para o autor quanto para os leitores do programa. Um exemplo ainda mais extremo da confusão em programas seria um usuário definir que o + entre dois operandos do tipo vetor seja a diferença entre os primeiros elementos de cada vetor. A sobrecarga de operadores é discutida com mais detalhes no Capítulo 7.

A simplicidade em linguagens pode, é claro, ser levada muito ao extremo. Por exemplo, a forma e o significado da maioria das sentenças de uma linguagem *assembly* são modelos de simplicidade, como você pode ver quando considera as sentenças que aparecem na próxima seção. Essa simplicidade extrema, entretanto, torna menos legíveis os programas escritos em linguagem *assembly*.

Devido à falta de sentenças de controle mais complexas, a estrutura de um programa é menos óbvia; e como as sentenças são simples, são necessárias mais do que em programas equivalentes escritos em uma linguagem de alto nível. Os mesmos argumentos se aplicam para os casos menos extremos de linguagens de alto nível com construções inadequadas de controle e de estruturas de dados.

1.3.1.2 Ortogonalidade

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

O significado de um recurso de linguagem ortogonal é independente do contexto de sua aparição em um programa (o nome ortogonal vem do conceito matemático de vetores ortogonais, independentes uns dos outros). A ortogonalidade vem de uma simetria de relacionamentos entre primitivas. Uma falta de ortogonalidade leva a exceções às regras de linguagem. Por exemplo, deve ser possível, em uma linguagem de programação que possibilita o uso de

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

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

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

onde `Reg1` e `Reg2` representam registradores. A semântica desses é

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

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

```
ADDL operand_1, operand_2
```

cuja semântica é

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

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

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

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

Como exemplos da falta de ortogonalidade em uma linguagem de alto nível, considere as seguintes regras e exceções em C. Apesar de C ter duas formas de tipos de dados estruturados, vetores e registros (**`structs`**), os registros podem ser retornados por funções, mas os vetores não. Um membro de uma

estrutura pode ser de qualquer tipo de dados, exceto **void** ou uma estrutura do mesmo tipo. Um elemento de um vetor pode ser qualquer tipo de dados, exceto **void** ou uma função. Parâmetros são passados por valor, a menos que sejam vetores, o que faz com que sejam passados por referência (porque a ocorrência de um nome de um vetor sem um índice em um programa em C é interpretada como o endereço do primeiro elemento desse vetor).

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

```
a + b
```

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

Muita ortogonalidade também pode causar problemas. Talvez a linguagem de programação mais ortogonal seja o ALGOL 68 (van Wijngaarden et al.,1969). Cada construção de linguagem no ALGOL 68 tem um tipo, e não existem restrições nesses tipos. Além disso, mais construções produzem valores. Essa liberdade de combinações permite construções extremamente complexas. Por exemplo, uma sentença condicional pode aparecer no lado esquerdo de uma atribuição, com declarações e outras sentenças diversas, desde que o resultado seja um endereço. Essa forma extrema de ortogonalidade leva a uma complexidade desnecessária. Como as linguagens necessitam de um grande número de tipos primitivos, um alto grau de ortogonalidade resulta em uma explosão de combinações. Então, mesmo se as combinações forem simples, seu número já leva à complexidade.

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

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

1.3.1.3 Tipos de dados

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

```
timeOut = 1
```

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

```
timeOut = true
```

O significado dessa sentença é perfeitamente claro.

1.3.1.4 Projeto da sintaxe

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

- *Formato dos identificadores*. Restringir os identificadores a tamanhos muito curtos piora a legibilidade. Se os identificadores podem ter no máximo seis caracteres, como no Fortran 77, é praticamente impossível usar nomes conotativos para variáveis. Um exemplo mais extremo é o ANSI* BASIC (ANSI, 1978b), no qual um identificador poderia ser formado por apenas uma letra ou por uma letra seguida de um dígito. Outras questões de projeto relacionadas ao formato dos identificadores são discutidas no Capítulo 5.
- *Palavras especiais*. A aparência de um programa e sua legibilidade são fortemente influenciadas pela forma das palavras especiais de uma linguagem (por exemplo, **`while`**, **`class`** e **`for`**). O método para formar sentenças compostas, ou grupos de sentenças, é especialmente importante, principalmente em construções de controle. Algumas linguagens têm usado pares casados de palavras especiais ou de símbolos para formar grupos. C e seus descendentes usam chaves para especificar sentenças compostas. Todas essas linguagens sofrem porque os grupos de sentenças são sempre terminados da mesma forma, o que torna difícil determinar qual grupo está sendo finalizado quando um **`end`** ou um } aparece. O Fortran 95 e Ada tornaram isso claro ao usar uma sintaxe distinta para cada tipo de grupo de sentenças. Por exemplo, Ada utiliza **`end if`** para terminar uma construção de seleção e **`end loop`** para terminar uma construção de repetição. Esse é um exemplo do conflito entre a simpli-

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

cidade resultante ao serem usadas menos palavras reservadas, como em C++, e a maior legibilidade que pode resultar do uso de mais palavras reservadas, como em Ada.

Outra questão importante é se as palavras especiais de uma linguagem podem ser usadas como nomes de variáveis de programas. Se puderem, os programas resultantes podem ser bastante confusos. Por exemplo, no Fortran 95, palavras especiais como `Do` e `End` são nomes válidos de variáveis, logo a ocorrência dessas palavras em um programa pode ou não conotar algo especial.

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

Uma das principais reclamações sobre os comandos de *shell* do UNIX (Raymond, 2004) é que a sua aparência nem sempre sugere sua funcionalidade. Por exemplo, o significado do comando UNIX `grep` pode ser decifrado apenas com conhecimento prévio, ou talvez com certa esperteza e familiaridade com o editor UNIX `ed`. A aparência do `grep` não tem conotação alguma para iniciantes no UNIX. (No `ed`, o comando */regular_expression/* busca uma subcadeia que casa com a expressão regular. Preceder isso com `g` faz ele ser um comando global, especificando que o escopo da busca é o arquivo inteiro que está sendo editado. Seguir o comando com `p` especifica que as linhas contendo a subcadeia casada devem ser impressas. Logo `g`*/regular_expression/*`p`, que pode ser abreviado como `grep`, imprime todas as linhas em um arquivo que contenham subcadeias que casem com a expressão regular.)

1.3.2 Facilidade de escrita

A facilidade de escrita é a medida do quão facilmente uma linguagem pode ser usada para criar programas para um domínio. A maioria das características de linguagem que afetam a legibilidade também afeta a facilidade de escrita. Isso é derivado do fato de que o processo de escrita de um programa requer que o programador releia sua parte já escrita.

Como ocorre com a legibilidade, a facilidade de escrita deve ser considerada no contexto do domínio de problema alvo de uma linguagem. Não é razoável comparar a facilidade de escrita de duas linguagens no contexto de uma aplicação em particular quando uma delas foi projetada para tal aplicação

e a outra não. Por exemplo, as facilidades de escrita do Visual BASIC (VB) e do C são drasticamente diferentes para criar um programa com uma interface gráfica com o usuário, para o qual o VB é ideal. Suas facilidades de escrita também são bastante diferentes para a escrita de programas de sistema, como um sistema operacional, para os quais a linguagem C foi projetada.

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

1.3.2.1 Simplicidade e ortogonalidade

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

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

1.3.2.2 Suporte à abstração

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

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

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

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

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

1.3.2.3 Expressividade

A expressividade em uma linguagem pode se referir a diversas características. Em uma linguagem como APL (Gilman e Rose, 1976), expressividade significa a existência de operadores muito poderosos que permitem muitas computações com um programa muito pequeno. Em geral, uma linguagem expressiva especifica computações de uma forma conveniente, em vez de desselegante. Por exemplo, em C, a notação `count++` é mais conveniente e menor do que `count = count + 1`. Além disso, o operador booleano **`and then`** em Ada é uma maneira conveniente de especificar avaliação em curto-circuito de uma expressão booleana. A inclusão da sentença **`for`** em Java torna a escrita de laços de contagem mais fácil do que com o uso do **`while`**, também possível. Todas essas construções aumentam a facilidade de escrita de uma linguagem.

1.3.3 Confiabilidade

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

1.3.3.1 Verificação de tipos

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

Um exemplo de como a falha em verificar tipos, tanto em tempo de compilação quanto em tempo de execução, tem levado a incontáveis erros de

programa é o uso de parâmetros de subprogramas na linguagem C original (Kernighan e Ritchie, 1978). Nessa linguagem, o tipo de um parâmetro real em uma chamada à função não era verificado para determinar se seu tipo casava com aquele do parâmetro formal correspondente na função. Uma variável do tipo `int` poderia ser usada como um parâmetro real em uma chamada à uma função que esperava um tipo `float` como seu parâmetro formal, e nem o compilador nem o sistema de tempo de execução teriam detectado a inconsistência. Por exemplo, como a sequência de bits que representa o inteiro 23 é essencialmente não relacionada com a sequência de bits que representa o ponto flutuante 23, se um inteiro 23 fosse enviado a uma função que espera um parâmetro de ponto flutuante, quaisquer usos desse parâmetro na função produziriam resultados sem sentido. Além disso, tais problemas são difíceis de serem diagnosticados[2]. A versão atual de C eliminou esse problema ao requerer que todos os parâmetros sejam verificados em relação aos seus tipos. Subprogramas e técnicas de passagem de parâmetros são discutidos no Capítulo 9.

1.3.3.2 Tratamento de exceções

A habilidade de um programa de interceptar erros em tempo de execução (além de outras condições não usuais detectáveis pelo programa), tomar medidas corretivas e então continuar é uma ajuda óbvia para a confiabilidade. Tal facilidade é chamada de **tratamento de exceções**. Ada, C++ e Java incluem diversas capacidades para tratamento de exceções, mas tais facilidades são praticamente inexistentes em muitas linguagens amplamente usadas, como C e Fortran. O tratamento de exceções é discutido no Capítulo 14.

1.3.3.3 Utilização de apelidos

Em uma definição bastante informal, **apelidos** são permitidos quando é possível ter um ou mais nomes para acessar a mesma célula de memória. Atualmente, é amplamente aceito que o uso de apelidos é um recurso perigoso em uma linguagem de programação. A maioria das linguagens permite algum tipo de apelido – por exemplo, dois ponteiros configurados para apontarem para a mesma variável, o que é possível na maioria das linguagens. O programador deve sempre lembrar que trocar o valor apontado por um dos dois ponteiros modifica o valor referenciado pelo outro. Alguns tipos de uso de apelidos, conforme descrito nos Capítulos 5 e 9, podem ser proibidos pelo projeto de uma linguagem.

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

---

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

1.3.3.4 Legibilidade e facilidade de escrita

Tanto a legibilidade quanto a facilidade de escrita influenciam a confiabilidade. Um programa escrito em uma linguagem que não oferece maneiras naturais para expressar os algoritmos requeridos irá necessariamente usar abordagens não naturais, menos prováveis de serem corretas em todas as situações possíveis. Quanto mais fácil é escrever um programa, mais provavelmente ele estará correto.

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

1.3.4 Custo

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

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

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

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

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

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

Uma escolha simples pode ser feita entre o custo de compilação e a velocidade de execução do código compilado. **Otimização** é o nome dado à coleção de técnicas que os compiladores podem usar para diminuir o tamanho e/ou aumentar a velocidade do código que produzem. Se pouca ou nenhuma otimização é feita, a compilação pode ser feita muito mais rapidamente do que se um esforço significativo for feito para produzir código otimizado. A escolha entre as duas alternativas é influenciada pelo ambiente no qual o compilador será usado. Em um laboratório para estudantes que estão iniciando a programação – os quais geralmente compilam seus programas

diversas vezes durante o desenvolvimento, mas que usam pouco tempo de execução de código (seus programas são pequenos e precisam ser executados corretamente apenas uma vez) – pouca ou nenhuma otimização deve ser feita. Em um ambiente de produção, onde os programas compilados são executados muitas vezes após o desenvolvimento, é melhor pagar o custo extra de otimizar o código.

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

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

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

A importância da facilidade de manutenção de software não pode ser subestimada. Tem sido estimado que, para grandes sistemas de software com tempos de vida relativamente longos, os custos de manutenção podem ser tão grandes como o dobro ou o quádruplo dos custos de desenvolvimento (Sommerville, 2005).

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

É claro, outros critérios podem ser usados para avaliar linguagens de programação. Um exemplo é a **portabilidade**, a facilidade com a qual os programas podem ser movidos de uma implementação para outra. A portabilidade é, na maioria das vezes, fortemente influenciada pelo grau de padronização da linguagem. Algumas, como o BASIC, não são padronizadas, fazendo os programas escritos nessas linguagens serem difíceis de mover de uma implementação para outra.

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

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

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

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

1.4 INFLUÊNCIAS NO PROJETO DE LINGUAGENS

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

1.4.1 Arquitetura de computadores

A arquitetura básica dos computadores tem um efeito profundo no projeto de linguagens. A maioria das linguagens populares dos últimos 50 anos tem sido projetada considerando a principal arquitetura de computadores, chamada de **arquitetura de von Neumann**, cujo nome é derivado de um de seus criadores, John von Neumann (pronuncia-se "von Noyman"). Elas são chamadas de linguagens **imperativas**. Em um computador von Neumann, tanto os dados quanto os programas são armazenados na mesma memória. A unidade central de processamento (CPU), que executa instruções, é separada da memória. Logo, instruções e dados devem ser transmitidos da memória para a CPU. Resultados de operações na CPU devem ser retornados para a memória. Praticamente todos os computadores digitais construídos desde os anos 1940 têm sido baseados nessa arquitetura. A estrutura geral de um computador von Neumann é mostrada na Figura 1.1.

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

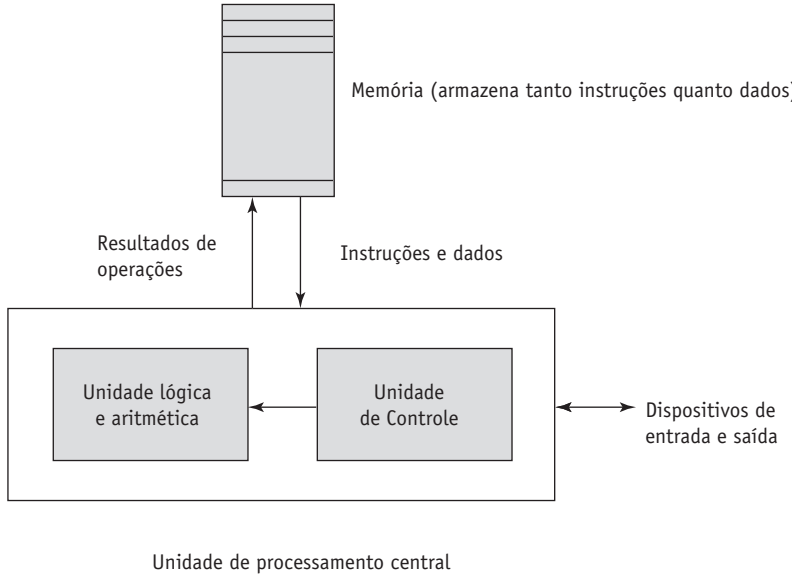

Figura 1.1 A arquitetura de computadores de von Neumann.

A execução de um programa em código de máquina em uma arquitetura de computadores von Neumann ocorre em um processo chamado de **ciclo de obtenção e execução**. Conforme mencionado, os programas residem na memória, mas são executados na CPU. Cada instrução a ser executada deve ser movida da memória para o processador. O endereço da próxima instrução a ser executada é mantido em um registrador chamado de **contador de programa**. Esse ciclo pode ser descrito de maneira simples pelo algoritmo:

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

O passo "decodifique a instrução" no algoritmo significa que a instrução é examinada para determinar que ação ela especifica. A execução de um programa termina quando uma instrução de parada é encontrada, apesar de, em um computador real, uma instrução de parada raramente ser executada. Em vez disso, o controle é transferido do sistema operacional a um programa de usuário para sua execução e retorna para o sistema operacional quando a execução do programa de usuário estiver completa. Em um sistema de

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

Deve ser possível, em uma linguagem de programação, definir um ponteiro para qualquer tipo específico definido na linguagem. Conforme mencionado, uma linguagem funcional, ou aplicativa, é uma na qual a principal forma de computação é a aplicação de funções para parâmetros fornecidos. A programação pode ser feita em uma linguagem funcional sem os tipos de variáveis usados nas linguagens imperativas, sentenças de atribuição e iteração. Apesar de muitos cientistas da computação terem explicado a infinidade de vantagens oriundas das linguagens funcionais, como Scheme, é improvável que elas substituam as linguagens imperativas até que um computador que não use a arquitetura von Neumann seja projetado e permita a execução eficiente de programas em linguagens funcionais. Dentre aqueles que já reclamaram desse fato, o mais eloquente foi John Backus (1978), o principal projetista da versão original do Fortran.

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

1.4.2 Metodologias de projeto de programas

O final dos anos 1960 e o início dos anos 1970 trouxeram uma análise intensa, iniciada em grande parte pelo movimento da programação estruturada, tanto no processo de desenvolvimento de software quanto no projeto de linguagens de programação.

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

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

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

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

O último grande passo na evolução do desenvolvimento de software orientado a dados, que começou no início dos anos 1980, é o projeto orientado a objetos. As metodologias orientadas a objetos começam com a abstração de dados, que encapsula o processamento com os objetos de dados e controla o acesso aos dados, e também adiciona mecanismos de herança e vinculação dinâmica de métodos. A herança é um conceito poderoso que melhora o potencial reúso de software existente, fornecendo a possibilidade de melhorias significativas na produtividade no contexto de desenvolvimento de software. Esse é um fator importante no aumento da popularidade das linguagens orientadas a objetos. A vinculação dinâmica de métodos (em tempo de execução) permite um uso mais flexível da herança.

A programação orientada a objetos se desenvolveu com uma linguagem que oferecia suporte para seus conceitos: Smalltalk (Goldberg e Robson, 1989). Apesar de Smalltalk nunca ter se tornado amplamente usada como muitas outras linguagens, o suporte à programação orientada a objetos é agora parte da maioria das linguagens imperativas populares, incluindo Ada 95 (ARM, 1995), Java e C++. Conceitos de orientação a objetos também encontraram seu caminho em linguagens funcionais como em CLOS (Bobrow et al., 1988) e na programação lógica em Prolog++ (Moss, 1994). O suporte à programação orientada a objetos é discutido em detalhes no Capítulo 12.

A programação orientada a procedimentos é, de certa forma, o oposto da orientada a dados. Apesar de os métodos orientados a dados dominarem o desenvolvimento de software atualmente, os métodos orientados a procedimentos não foram abandonados. Boa parte da pesquisa vem ocorrendo em programação orientada a procedimentos nos últimos anos, especialmente na área de concorrência. Esses esforços de pesquisa trouxeram a necessidade de recursos de linguagem para criar e controlar unidades de programas concorrentes. Ada, Java e C# incluem tais capacidades, que serão discutidas em detalhes no Capítulo 13.

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

1.5 CATEGORIAS DE LINGUAGENS

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

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

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

Uma linguagem de programação lógica é um exemplo de uma baseada em regras. Em uma linguagem imperativa, um algoritmo é especificado em muitos detalhes, e a ordem de execução específica das instruções ou sentenças deve ser incluída. Em uma linguagem baseada em regras, entretanto, estas são especificadas sem uma ordem em particular, e o sistema de implementação da linguagem deve escolher uma ordem na qual elas são usadas para produzir os resultados desejados. Essa abordagem para o desenvolvimento de software é radicalmente diferente daquelas usadas nas outras três categorias de linguagens e requer um tipo completamente diferente de linguagem. Prolog, a linguagem de programação lógica mais usada, e a programação lógica propriamente dita são discutidas no Capítulo 16.

Nos últimos anos, surgiu uma nova categoria: as linguagens de marcação/linguagens de programação híbridas. As de marcação não são de programação. Por exemplo, XHTML, a linguagem de marcação mais utilizada, es-

pecifica a disposição da informação em documentos Web. Entretanto, algumas capacidades de programação foram colocadas em extensões de XHTML e XML. Dentre essas, estão a biblioteca padrão de JSP, chamada de Java Server Pages Standard Tag Library (JSTL) e a linguagem chamada eXtensible Stylesheet Language Transformations (XSLT). Ambas são brevemente introduzidas no Capítulo 2. Elas não podem ser comparadas a qualquer linguagem de programação completa e, dessa forma, não serão discutidas após esse capítulo.

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

1.6 *TRADE-OFFS* NO PROJETO DE LINGUAGENS

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

Como outro exemplo de critérios conflitantes que levam diretamente aos *trade-offs* de projeto, considere o caso de APL, que inclui um poderoso conjunto de operadores para operandos do tipo matriz. Dado o grande número de operadores, um número significativo de novos símbolos teve de ser incluído em APL para apresentar esses operadores. Além disso, muitos operadores APL podem ser usados em uma expressão longa e complexa. Um resultado desse alto grau de expressividade é que, para aplicações envolvendo muitas operações de matrizes, APL tem uma facilidade de escrita muito grande. De fato, uma enorme quantidade de computação pode ser especificada em um programa muito pequeno. Outro resultado é que os programas APL têm uma legibilidade muito pobre. Uma expressão compacta e concisa tem certa beleza matemática, mas é difícil para qualquer um, exceto o autor, entendê-la. O renomado autor Daniel McCraken (1970) afirmou em certa ocasião que levou quatro

horas para ler e entender um programa APL de quatro linhas. O projetista trocou legibilidade pela facilidade de escrita.

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

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

1.7 MÉTODOS DE IMPLEMENTAÇÃO

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

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

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

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

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

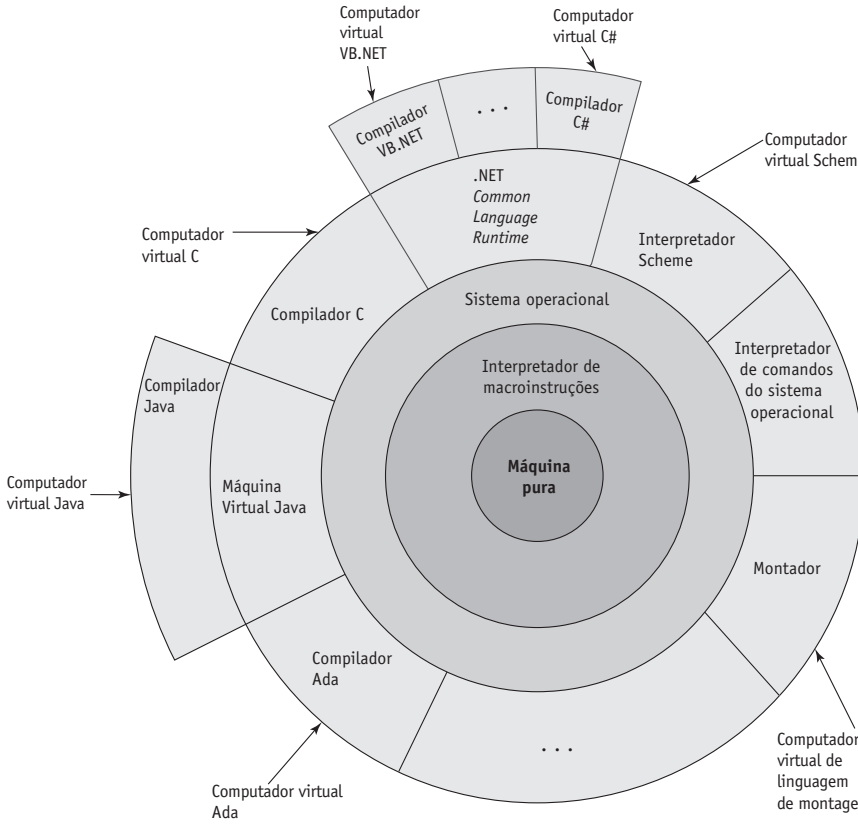

Figura 1.2 Interface em camadas de computadores virtuais, fornecida por um sistema de computação típico.

de implementação é uma aplicação de partes das teorias de autômatos e da de linguagens formais que eram bem entendidas.

1.7.1 Compilação

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

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

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

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

O gerador de código intermediário produz um programa em uma linguagem diferente, em um nível intermediário entre o programa fonte e a saída final do compilador: o programa em linguagem de máquina[3]. Linguagens intermediárias algumas vezes se parecem muito com as de montagem e, de fato, algumas vezes são linguagens de montagem propriamente ditas. Em outros casos, o código intermediário está a um nível um pouco mais alto do que uma linguagem de montagem. O analisador semântico é parte do gerador de código intermediário, que verifica erros difíceis (ou impossíveis) de ser detectados durante a análise sintática, como erros de tipos.

A otimização – que melhora os programas (normalmente, em sua versão de código intermediário) tornando-os menores, mais rápidos ou ambos –, é uma parte opcional da compilação. Na verdade, alguns compiladores são incapazes de fazer quaisquer otimizações significativas. Esse tipo de compilador seria usado em situações nas quais a velocidade de execução do programa traduzido é bem menos importante do que a velocidade de com-

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

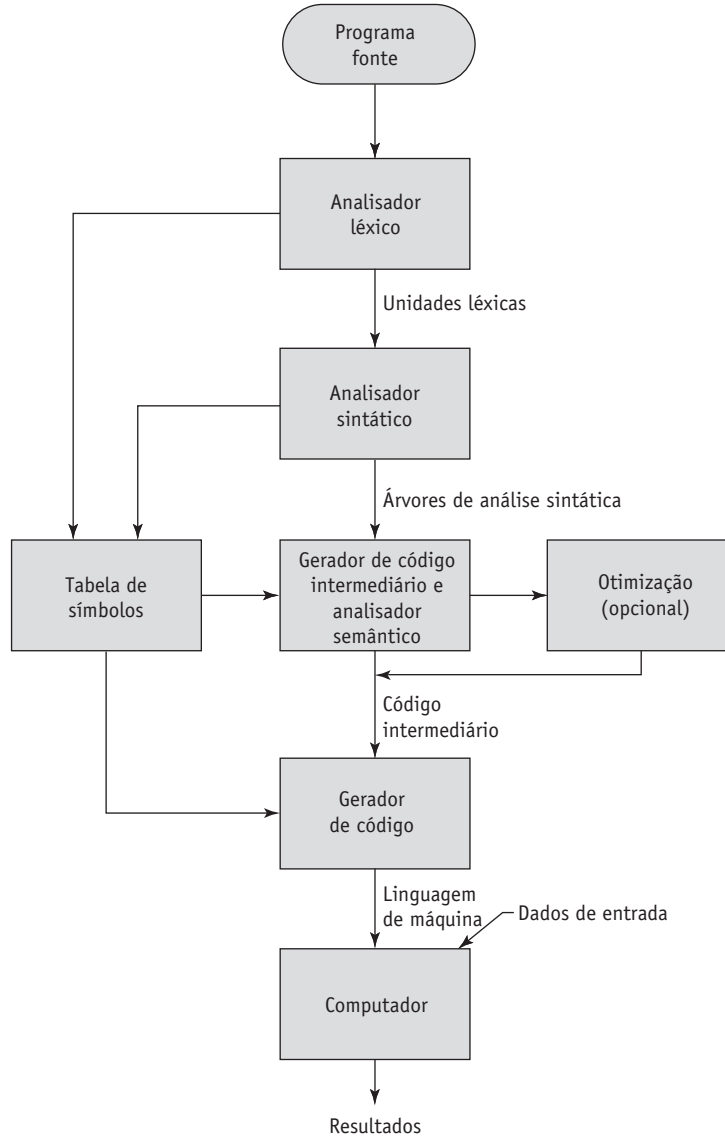

Figura 1.3 O processo de compilação.

pilação. Um exemplo de tal situação é um laboratório de computação para programadores iniciantes. Na maioria das situações comerciais e industriais, a velocidade de execução é mais importante do que a como de compilação, logo a otimização é rotineiramente desejável. Como muitos tipos de otimização são difíceis de serem feitas em linguagem de máquina, a maioria é feita em código intermediário.

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

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

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

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

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

1.7.2 Interpretação pura

A interpretação pura reside no oposto (em relação à compilação) dos métodos de implementação. Com essa abordagem, os programas são interpretados por outro, chamado **interpretador**, sem tradução. O interpretador age como uma simulação em software de uma máquina cujo ciclo de obtenção-execução trata de sentenças de programa de alto nível em vez de instruções de máquina. Essa simulação em software fornece uma máquina virtual para a linguagem.

A interpretação pura tem a vantagem de permitir uma fácil implementação de muitas operações de depuração em código fonte, pois todas as mensagens de erro em tempo de execução podem referenciar unidades de código

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

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

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

1.7.3 Sistemas de implementação híbridos

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

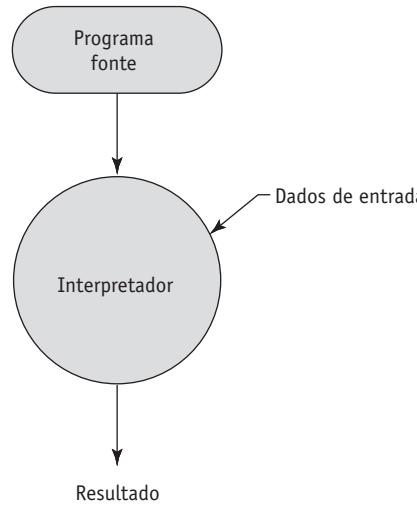

Figura 1.4 Interpretação pura.

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

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

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

As primeiras implementações de Java eram todas híbridas. Seu formato intermediário, chamado de ***bytecode***, fornece portabilidade para qualquer má-

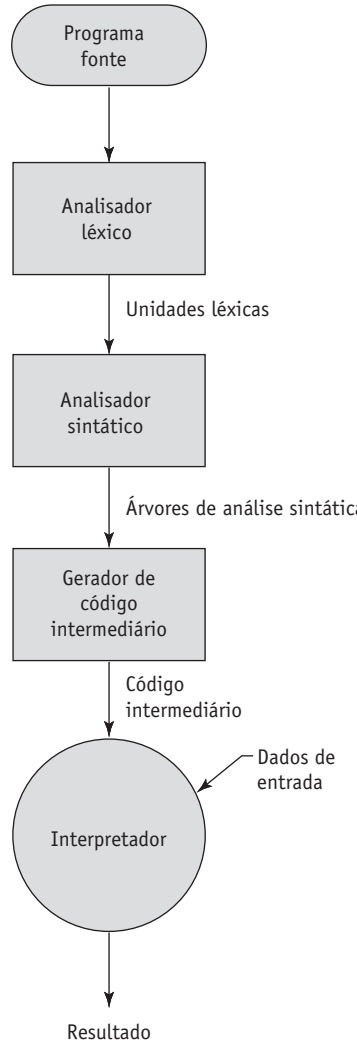

Figura 1.5 Sistema de implementação híbrido.

quina que tenha um interpretador de *bytecodes* e um sistema de tempo de execução associado. Juntos, eles são chamados de Máquina Virtual Java. Existem agora sistemas que traduzem *bytecodes* Java para código de máquina de forma a possibilitar uma execução mais rápida.

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

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

1.7.4 Pré-processadores

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

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

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

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

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

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

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

seria expandida pelo pré-processador para

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

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

1.8 AMBIENTES DE PROGRAMAÇÃO

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

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

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

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

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

RESUMO

O estudo de linguagens de programação é valioso por diversas razões: aumenta nossa capacidade de usar diferentes construções ao escrever programas, permite que escolhamos linguagens para os projetos de forma mais inteligente e torna mais fácil o aprendizado de novas linguagens.

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

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

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

Projetar uma linguagem de programação é primariamente um esforço de engenharia, no qual uma longa lista de *trade-offs* deve ser levada em consideração na escolha de recursos, construções e capacidades.

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

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

QUESTÕES DE REVISÃO

1. Por que é útil para um programador ter alguma experiência no projeto de linguagens, mesmo que ele nunca projete uma linguagem de programação?
2. Como o conhecimento de linguagens de programação pode beneficiar toda a comunidade da computação?
3. Que linguagem de programação tem dominado a computação científica nos últimos 50 anos?
4. Que linguagem de programação tem dominado as aplicações de negócios nos últimos 50 anos?
5. Que linguagem de programação tem dominado a Inteligência Artificial nos últimos 50 anos?
6. Em que linguagem o UNIX é escrito?
7. Qual é a desvantagem de ter muitas características em uma linguagem?
8. Como a sobrecarga de operador definida pelo usuário pode prejudicar a legibilidade de um programa?
9. Cite um exemplo da falta de ortogonalidade no projeto da linguagem C.
10. Qual linguagem usou a ortogonalidade como um critério de projeto primário?
11. Que sentença de controle primitiva é usada para construir sentenças de controle mais complicadas em linguagens que não as têm?
12. Que construção de uma linguagem de programação fornece abstração de processos?
13. O que significa para um programa ser confiável?
14. Por que verificar os tipos dos parâmetros de um subprograma é importante?
15. O que são apelidos?
16. O que é o tratamento de exceções?
17. Por que a legibilidade é importante para a facilidade de escrita?
18. Como o custo de compiladores para uma linguagem está relacionado ao projeto dela?

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

CONJUNTO DE PROBLEMAS

1. Você acredita que nossa capacidade de abstração é influenciada por nosso domínio de linguagens? Defenda sua opinião.
2. Cite alguns dos recursos de linguagens de programação específicas que você conhece cujo objetivo seja um mistério para você.
3. Que argumentos você pode dar a favor da ideia de uma única linguagem para todos os domínios de programação?
4. Que argumentos você pode dar contra a ideia de uma única linguagem para todos os domínios de programação?
5. Nomeie e explique outro critério pelo qual as linguagens podem ser julgadas (além dos discutidos neste capítulo).
6. Que sentença comum das linguagens de programação, em sua opinião, é mais prejudicial à legibilidade?
7. Java usa um símbolo de fechamento de chaves para marcar o término de todas as sentenças compostas. Quais são os argumentos a favor e contra essa decisão de projeto?
8. Muitas linguagens distinguem entre letras minúsculas e maiúsculas em nomes definidos pelo usuário. Quais são as vantagens e desvantagens dessa decisão de projeto?
9. Explique os diferentes aspectos do custo de uma linguagem de programação.
10. Quais são os argumentos para escrever programas eficientes mesmo sabendo que os sistemas de hardware são relativamente baratos?
11. Descreva alguns *trade-offs* de projeto entre a eficiência e a segurança em alguma linguagem que você conheça.
12. Quais recursos principais uma linguagem de programação perfeita deveria incluir, em sua opinião?

13. A primeira linguagem de programação de alto nível que você aprendeu era implementada com um interpretador puro, um sistema de implementação híbrido ou um compilador? (Você não necessariamente saberá isso sem pesquisar).
14. Descreva as vantagens e desvantagens de alguns ambientes de programação que você já tenha usado.
15. Como sentenças de declaração de tipos para variáveis simples afetam a legibilidade de uma linguagem, considerando que algumas não precisam de tais declarações?
16. Escreva uma avaliação de alguma linguagem de programação que você conheça, usando os critérios descritos neste capítulo.
17. Algumas linguagens de programação – por exemplo, Pascal – têm usado o ponto e vírgula para separar sentenças, enquanto Java os utiliza para terminar sentenças. Qual desses usos, em sua opinião, é mais natural e menos provável de resultar em erros de sintaxe? Justifique sua resposta.
18. Muitas linguagens contemporâneas permitem dois tipos de comentários: um no qual os delimitadores são usados em ambas as extremidades (comentários de múltiplas linhas) e um no qual um delimitador marca apenas o início do comentário (comentário de uma linha). Discuta as vantagens e desvantagens de cada um dos tipos de acordo com nossos critérios.