algoritmos e programação

com exemplos em Pascal e C

nina edelweiss
maria aparecida castro livi

bookman

as autoras

Nina Edelweiss é engenheira eletricista e doutora em Ciência da Computação pela Universidade Federal do Rio Grande do Sul. Durante muitos anos, lecionou em cursos de Engenharia e de Ciência da Computação na UFRGS, na UFSC e na PUCRS. Foi, ainda, orientadora do Programa de Pós-Graduação em Ciência da Computação da UFRGS. É coautora de três livros, tendo publicado diversos artigos em periódicos e em anais de congressos nacionais e internacionais. Participou de diversos projetos de pesquisa financiados por agências de fomento como CNPq e FAPERGS, desenvolvendo pesquisas nas áreas de bancos de dados e desenvolvimento de software.

Maria Aparecida Castro Livi é licenciada e bacharel em Letras, e mestre em Ciência da Computação pela Universidade Federal do Rio Grande do Sul. Desenvolveu sua carreira profissional na UFRGS, onde foi programadora e analista de sistema, antes de ingressar na carreira docente. Ministrou por vários anos a disciplina de Algoritmos e Programação para alunos dos cursos de Engenharia da Computação e Ciência da Computação. Sua área de interesse prioritário é o ensino de Linguagens de Programação, tanto de forma presencial quanto a distância.

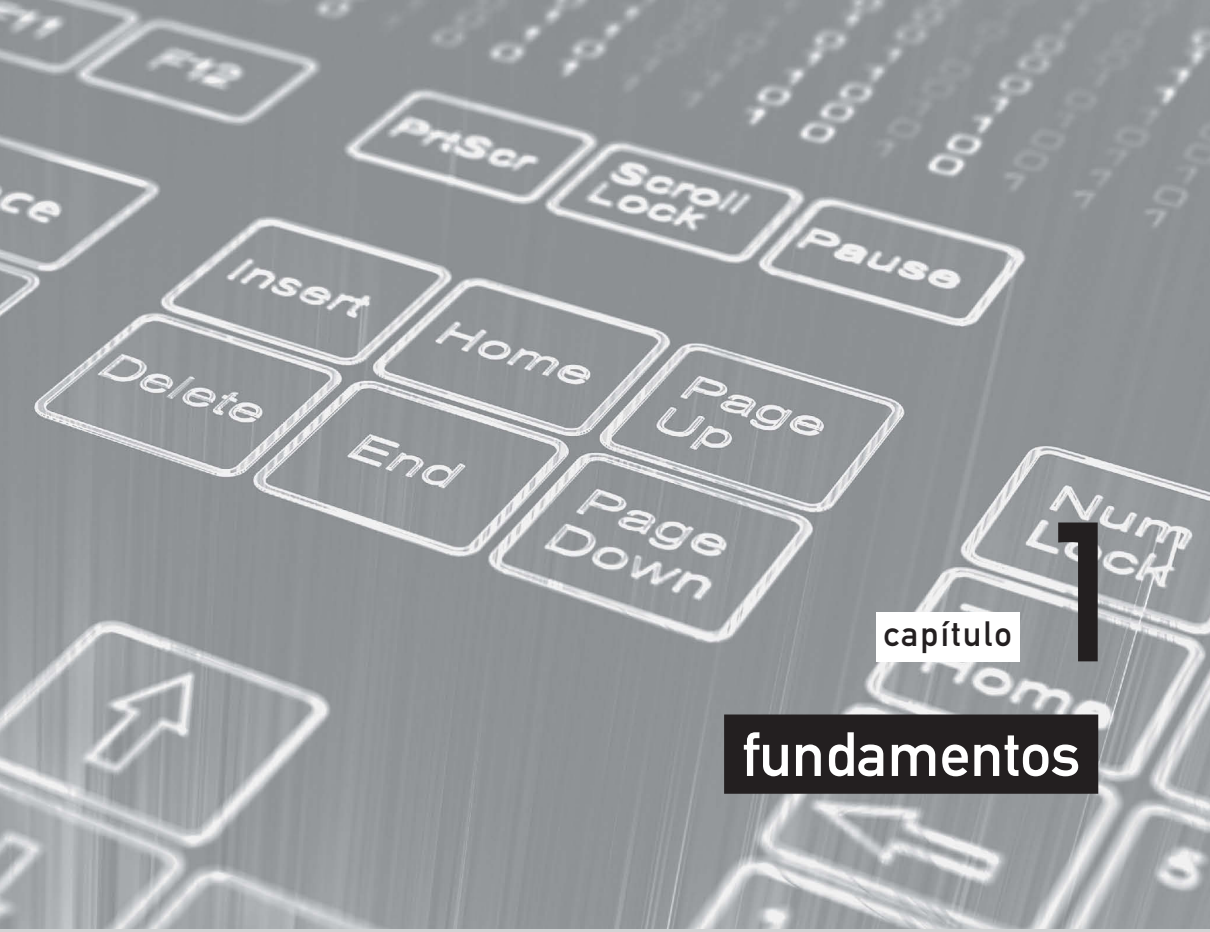
E22a Edelweiss, Nina.
Algoritmos e programação com exemplos em Pascal e C [recurso eletrônico] / Nina Edelweiss, Maria Aparecida Castro Livi. – Dados eletrônicos. – Porto Alegre : Bookman, 2014.

Editado também como livro impresso em 2014.
ISBN 978-85-8260-190-7

1. Informática. 2. Algoritmos – Programação. I. Livi, Maria Aparecida Castro. II. Título.

CDU 004.421

Catalogação na publicação: Ana Paula M. Magnus – CRB 10/2052



capítulo 1

fundamentos

Este primeiro capítulo discute algoritmos, formas de expressar algoritmos, etapas para a construção de um algoritmo e de um programa, paradigmas de programação, programação estruturada e fundamentos de representação interna de dados. Introduz, ainda, as linguagens de programação Pascal e C, utilizadas no livro.

Computadores constituem uma poderosa ferramenta para auxiliar o trabalho do homem. O uso mais comum dos computadores é por meio de aplicativos já desenvolvidos e disponíveis, tais como editores de texto, planilhas eletrônicas, sistemas de gerenciamento de bancos de dados, programas de acesso à Internet e jogos. Entretanto, por vezes, as pessoas desenvolvem soluções específicas para determinadas aplicações, de modo a permitir que as informações dessas aplicações possam ser acessadas e manipuladas de forma mais segura, rápida e eficiente ou com um custo mais baixo. Este livro trata dessa segunda forma de uso dos computadores, ou seja, de como um usuário pode projetar e desenvolver soluções próprias para resolver problemas específicos de seu interesse.

Este primeiro capítulo apresenta alguns conceitos básicos utilizados no restante do livro: o que vem a ser um algoritmo, formas de expressar algoritmos, etapas para a construção de um algoritmo e de um programa, algumas considerações a respeito das linguagens de programação utilizadas, o que vem a ser a programação estruturada, que é a técnica de programação adotada no desenvolvimento dos programas aqui apresentados, e alguns fundamentos de representação interna de dados.

1.1 → o que é um algoritmo

Vejamos como são solucionados alguns problemas do cotidiano.

exemplo 1: Telefone público. Para utilizar um telefone público como um "orelhão" ou similar, as operações que devemos realizar estão especificadas junto a esse telefone, sendo mais ou menos assim:

1. leve o fone ao ouvido;
2. insira seu cartão no orifício apropriado;
3. espere o sinal para discar;
4. assim que ouvir o sinal, disque o número desejado;
5. ao final da ligação, retorne o fone para a posição em que se encontrava;
6. retire seu cartão.

Esse conjunto de operações é o que se denomina algoritmo. Qualquer pessoa pode executar essas operações, na ordem especificada, para fazer suas ligações telefônicas, desde que possua um cartão específico e conheça o número para o qual quer telefonar.

exemplo 2: Compra de um livro. Uma compra em um estabelecimento comercial também obedece a uma sequência de ações predeterminadas. Por exemplo, para comprar um livro em uma livraria deve-se:

1. entrar na livraria;
2. verificar se o livro está disponível. Para isso, precisa-se conhecer (1) o título e o autor do livro e (2) ter disponibilidade financeira para a compra. Caso a compra venha a ser efetuada, deve-se:
   a. levar o livro até o balcão;
   b. esperar que a compra seja registrada no caixa;

   c. pagar o valor correspondente;
   d. esperar que seja feito o pacote;
   e. levar o livro comprado.
3. sair da livraria.

Os dois exemplos apresentados são resolvidos por uma sequência de ações bem definidas, que devem ser executadas em uma determinada ordem. Outras aplicações de nosso dia a dia podem ser detalhadas de forma semelhante: uma receita de um bolo, o acesso a terminais eletrônicos de bancos, a troca do pneu de um carro, etc.

definição de algoritmo. Um **algoritmo** é definido como uma sequência finita de operações que, quando executadas na ordem estabelecida, atingem um objetivo determinado em um tempo finito.

Um algoritmo deve atender aos seguintes requisitos:

- possuir um estado inicial;
- consistir de uma sequência lógica finita de ações claras e precisas;
- produzir dados de saída corretos;
- possuir estado final previsível (deve sempre terminar).

Além de definir algoritmos para resolver problemas do dia a dia, podemos também desenvolver algoritmos que podem ser transformados, total ou parcialmente, em programas e executados em computadores. Este livro concentra-se em problemas resolvidos através de algoritmos que podem ser integralmente executados por computadores.

1.1.1 algoritmos executados por um computador

Para que um algoritmo possa ser totalmente executado por um computador é necessário identificar claramente quais as ações que essa máquina pode executar. O exemplo a seguir permite identificar, através de uma simulação, algumas das ações básicas que um computador executa e como isso é feito.

Vamos supor que um professor, na sala de aula, mostre aos alunos como é calculada a média das notas de uma prova. Para simplificar, suponhamos que a turma tenha somente cinco alunos. As provas estão sobre sua mesa, já corrigidas. O professor desenha uma grade no quadro, dando nome a cada um dos espaços nos quais vai escrever a nota de cada aluno: `Nota1`, `Nota2`, etc. (Figura 1.1). Acrescenta mais um espaço para escrever os resultados de seus cálculos, que chama de `Resultado`.

Para formalizar o que está fazendo, ele escreveu a sequência de ações que está executando em uma folha sobre sua mesa. Ele inicia pegando a primeira prova, olha sua nota (vamos supor que seja 10) e escreve essa nota no espaço reservado para ela, que chamou de `Nota1`. Essa prova ele coloca em uma segunda pilha sobre a mesa, pois já foi usada e não deve ser considerada uma segunda vez para calcular a média. Em seguida, faz o mesmo para a segunda prova (nota 8), escrevendo seu valor em `Nota2` e colocando-a na pilha das já utilizadas. Ele repete essa operação para cada uma das provas restantes.

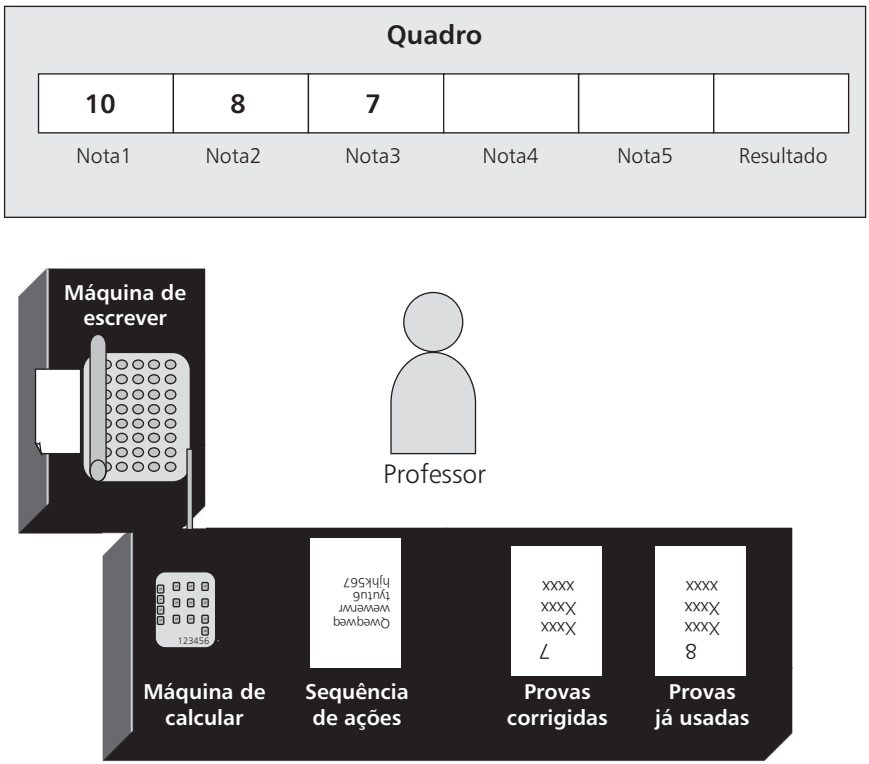


figura 1.1 Simulação de um algoritmo.

Obtidas todas as notas das provas, o professor passa a realizar as operações que vão calcular a média. Inicialmente, precisa somar todas as notas. Em cima da sua mesa, está uma calculadora. Ele consulta cada um dos valores das notas que escreveu no quadro e utiliza essa calculadora para fazer sua soma:

```
Soma = Nota1 + Nota2 + Nota3 + Nota4 + Nota5
```

O resultado da soma ele escreve no espaço que chamou de `Resultado` (Figura 1.2a).

Feita a soma, ela deve ser dividida por cinco para que seja obtida a média das notas. Utilizando novamente a calculadora, o professor consulta o que escreveu no espaço `Resultado` (onde está a soma) e divide este valor por cinco. Como não vai mais precisar do valor da soma, o professor utiliza o mesmo espaço, chamado `Resultado`, para escrever o valor obtido para a média, apagando o valor anterior (Figura 1.2b).

Finalizando, o professor escreve as cinco notas obtidas nas provas e a média em uma folha, utilizando uma máquina de escrever, para informar à direção da escola.

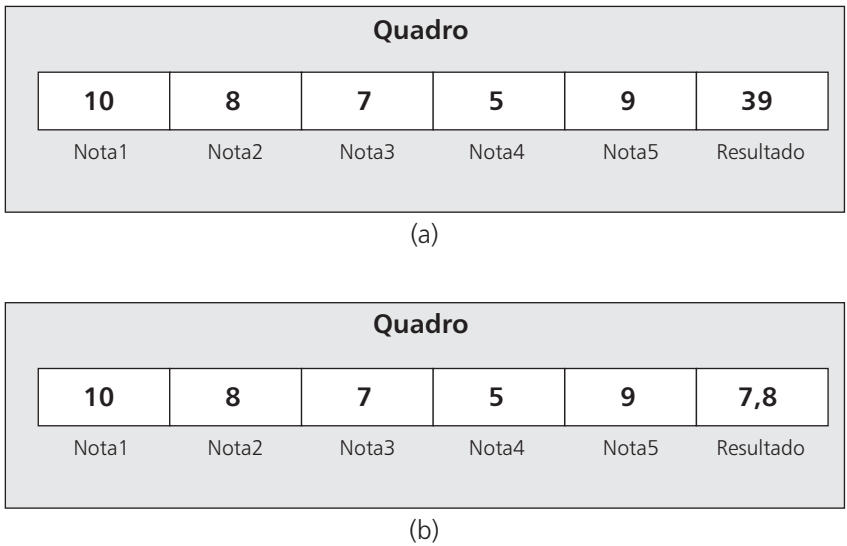


figura 1.2 Valores durante a simulação.

A sequência de ações que foram executadas foi a seguinte:

1. `ler a nota da primeira prova e escrevê-la em Nota1;`
2. `ler a nota da prova seguinte e escrevê-la em Nota2;`
3. `ler a nota da prova seguinte e escrevê-la em Nota3;`
4. `ler a nota da prova seguinte e escrevê-la em Nota4;`
5. `ler a nota da prova seguinte e escrevê-la em Nota5;`
6. `somar os valores escritos nos espaços Nota1, Nota2, Nota3, Nota4 e Nota5. Escrever o resultado da soma em Resultado;`
7. `dividir o valor escrito em Resultado por cinco e escrever o valor deste cálculo em Resultado;`
8. `usando a máquina de escrever, escrever os valores contidos em Nota1, Nota2, Nota3, Nota4, Nota5 e Resultado;`
9. `terminar a execução desta tarefa.`

Essa sequência de operações caracteriza um algoritmo, sendo que todas as ações realizadas nesse algoritmo podem ser executadas por um computador. A tradução desse algoritmo para uma linguagem que um computador possa interpretar gera o **programa** que deve ser executado pelo computador. O professor corresponde à unidade central de processamento (UCP ou, mais comumente, CPU, de *Central Processing Unit*), responsável pela execução desse programa. Essa unidade organiza o processamento e garante que as instruções sejam executadas na ordem correta.

Fazendo um paralelo entre o exemplo e um computador real, os espaços desenhados na grade do quadro constituem a memória principal do computador, que é composta por espaços acessados pelos programas através de nomes dados pelo programador.

Nesses espaços são guardadas, durante o processamento, informações lidas na entrada e resultados de processamentos, como no exemplo visto. Cada um desses espaços só pode conter um valor a cada momento, perdendo o valor anterior se um novo valor for armazenado nele, como ocorreu quando se escreveu a média em `Resultado`, apagando o valor da soma que lá estava. Denomina-se **variável** cada um desses espaços utilizados para guardar valores, denotando que seu valor pode variar ao longo do tempo. As instruções de um programa que está sendo executado também são armazenadas na memória principal. Todas as informações armazenadas nas variáveis da memória principal são perdidas no momento em que termina a execução do programa.

Unidades de memória secundária podem ser utilizadas para guardar informações (dados) a fim de serem utilizadas em outra ocasião. Exemplos de dispositivos de memória secundária são HDs (*Hard Disks*), CDs, DVDs e *pendrives*.

A comunicação do computador com o usuário durante o processamento e ao seu final é feita através de unidades de entrada e saída. No exemplo anterior, a pilha de provas corresponde à unidade de entrada do computador, através da qual são obtidos os valores que serão utilizados no processamento. A unidade de entrada mais usada para interação entre o usuário e o programa durante a execução é o teclado do computador. Quando se trata de imagens, a unidade de entrada pode ser, por exemplo, uma máquina fotográfica ou um *scanner*.

A máquina de escrever corresponde à unidade de saída, que informa aos usuários o resultado do processamento. Exemplos de unidades de saída são o vídeo do computador e uma impressora.

As unidades de entrada e saída de dados constituem as únicas interfaces do computador com seu usuário. Observe que, sem as unidades de entrada e saída, não é possível fornecer dados ao computador nem saber dos resultados produzidos.

A máquina de calcular corresponde à unidade aritmética e lógica do computador, responsável pelos cálculos e inferências necessários ao processamento. Sua utilização fica totalmente transparente ao usuário, que somente é informado dos resultados do processamento.

Resumindo, um computador processa dados. Processar compreende executar atividades como, por exemplo, comparações, realização de operações aritméticas, ordenações. A partir de dados (de entrada), processando-os, o computador produz resultados (saídas). Na figura 1.3, vê-se um esquema simplificado da organização funcional de um computador. Nela podem ser observados os sentidos em que as informações fluem durante a execução de um programa: o sistema central do computador compreende a CPU e a memória principal; na CPU estão as unidades de controle e de aritmética e lógica; a unidade de controle tem acesso à memória principal, às unidades de entrada e de saída de dados e aos dispositivos de memória secundária.

1.1.2 comandos básicos executados por um computador

Analisando o exemplo anterior, identificamos as primeiras ações que podem ser executadas por um computador:

- obter um dado de uma unidade de entrada de dados, também chamada de leitura de um dado;
- informar um resultado através de uma unidade de saída, também chamada de escrita de uma informação ou saída de um dado;
- resolver expressões aritméticas e lógicas;
- colocar o resultado de uma expressão em uma variável.

Essas ações são denominadas **instruções** ou **comandos**. Outros comandos serão vistos ao longo deste livro.

1.1.3 da necessidade do desenvolvimento de algoritmos para solucionar problemas computacionais

Nas atividades cotidianas já vistas, é sem dúvida necessária alguma organização por parte de quem vai realizar a tarefa. No uso do telefone, retirar o fone do gancho e digitar o número e, só depois, inserir o cartão não será uma boa estratégia, assim como, no caso da livraria, levar o livro sem passar pelo caixa também resultará em problemas. Nessas atividades, no entanto, grande parte das pessoas não necessita colocar por escrito os passos a realizar para cumprir a tarefa. Porém, quando se trata de problemas a solucionar por computador, a sequência de

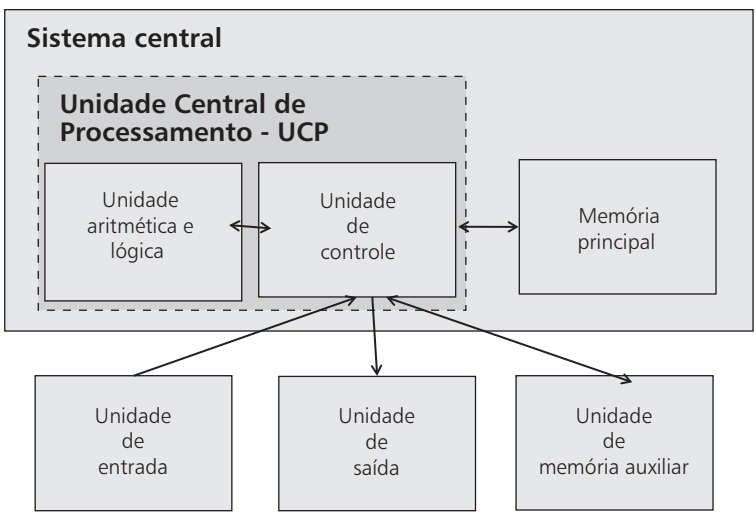


figura 1.3 Esquema simplificado de um computador.

ações que o computador deve realizar é por vezes bastante extensa e nem sempre conhecida e óbvia. Para a programação de computadores, a análise cuidadosa dos elementos envolvidos em um problema e a organização criteriosa da sequência de passos necessários à sua solução (algoritmo) devem obrigatoriamente preceder a escrita do programa que busque solucionar o problema. Para problemas mais complexos, o recomendável é desenvolver um algoritmo detalhado antes de passar à etapa de codificação, mas para problemas mais simples, o algoritmo pode especificar apenas os passos principais.

1.1.4 formas de expressar um algoritmo

Em geral, no desenvolvimento de algoritmos computacionais não são utilizadas nem as linguagens de programação nem a linguagem natural, mas formas mais simplificadas de linguagens. As formas mais usuais de representação de algoritmos são a linguagem textual, alguma pseudolinguagem e o fluxograma. Para exemplificar cada uma delas vamos usar o seguinte exemplo: *obter a soma de dois valores numéricos quaisquer*.

linguagem textual. Foi a forma utilizada para introduzir o conceito de algoritmo nos exemplos anteriores. Analisando o problema aqui colocado, para obter a soma de dois valores é preciso realizar três operações na ordem a seguir:

1. obter os dois valores
2. realizar a soma
3. informar o resultado

pseudolinguagem. Para padronizar a forma de expressar algoritmos são definidas pseudolinguagens. Uma pseudolinguagem geralmente é bastante semelhante a uma linguagem de programação, sem, entretanto, entrar em detalhes como, por exemplo, formatação de informações de entrada e de saída. As operações básicas que podem ser executadas pelo computador são representadas através de palavras padronizadas, expressas na linguagem falada (no nosso caso, em Português). Algumas construções também são padronizadas, como as que especificam onde armazenar valores obtidos e calculados, bem como a forma de calcular expressões aritméticas e lógicas.

Antecipando o que será visto nos capítulos a seguir, o algoritmo do exemplo recém-discutido é expresso na pseudolinguagem utilizada neste livro como:

```
Algoritmo 1.1 - Soma2
{INFORMAR A SOMA DE 2 VALORES}
  Entradas: valor1, valor2 (real)
  Saídas: soma (real)
início
  ler (valor1, valor2)            {ENTRADA DOS 2 VALORES}
  soma ← valor1 + valor2          {CALCULA A SOMA}
  escrever (soma)                 {INFORMA A SOMA}
fim
```

fluxograma. Trata-se de uma representação gráfica que possibilita uma interpretação visual do algoritmo. Cada ação é representada por um bloco, sendo os blocos interligados por linhas dirigidas (setas) que representam o fluxo de execução. Cada forma de bloco representa uma ação. A Figura 1.4 mostra alguns blocos utilizados em fluxogramas neste livro, juntamente com as ações que eles representam. São adotadas as formas propostas na padronização feita pela ANSI (*American National Standards Institute*) em 1963 (Chapin, 1970), com algumas adaptações. Outras formas de blocos serão introduzidas ao longo do texto. A representação do algoritmo do exemplo acima está na Figura 1.5.

A representação através de fluxogramas não é adequada para algoritmos muito extensos, com grande número de ações a executar. Utilizaremos a representação de fluxogramas somente como apoio para a compreensão das diferentes construções que podem ser utilizadas nos algoritmos.

1.1.5 eficácia e eficiência de algoritmos

Dois aspectos diferentes devem ser analisados quando se constrói um algoritmo para ser executado em um computador: sua eficácia (exatidão) e sua eficiência.

eficácia (corretude) de um algoritmo. Um algoritmo deve realizar corretamente a tarefa para a qual foi construído. Além de fazer o que se espera, o algoritmo deve fornecer o resultado correto para quaisquer que sejam os dados fornecidos como entrada. A eficácia de um algoritmo deve ser exaustivamente testada antes que ele seja implementado em um computador, o que levou ao desenvolvimento de diversas técnicas de testes, incluindo testes formais. A forma mais simples de testar um algoritmo é através de um "teste de mesa", no qual se si-

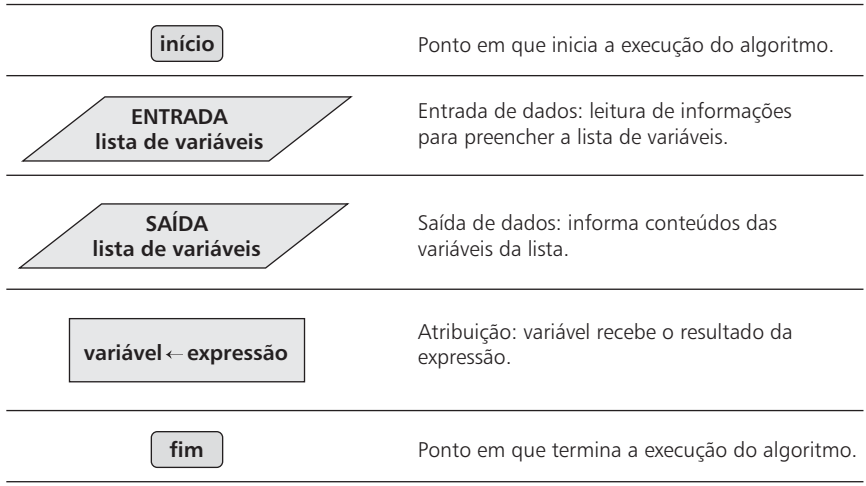


figura 1.4 Blocos de fluxograma.

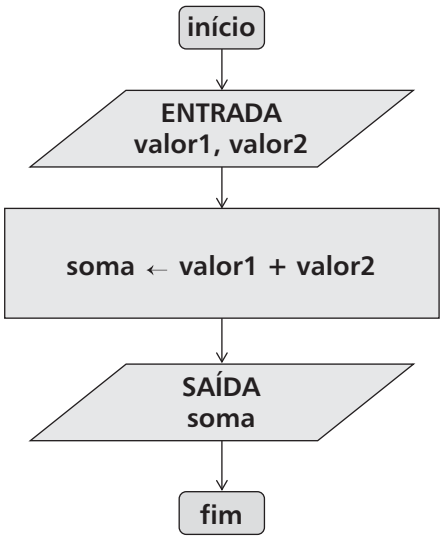


figura 1.5 Fluxograma da soma de dois números.

mula com lápis e papel sua execução, com conjuntos diferentes de dados de entrada. No final de cada capítulo deste livro, são indicados alguns cuidados a adotar para verificar a exatidão dos algoritmos durante os testes.

eficiência de um algoritmo. A solução de um problema através de um algoritmo não é necessariamente única. Na maioria dos casos, algoritmos diferentes podem ser construídos para realizar uma mesma tarefa. Neste livro será enfatizada a utilização de técnicas que levam à construção de algoritmos mais eficientes. Entretanto, em alguns casos não se pode dizer *a priori* qual a melhor solução. Pode-se, sim, calcular qual a forma mais eficiente, com base em dois critérios: tempo de execução e espaço de memória ocupado. Aspectos de eficiência de algoritmos são vistos em outro livro desta série (Toscani; Veloso, 2012).

Um exemplo da diferença entre eficácia e eficiência pode ser observado na receita de ovo mexido mostrada a seguir:

1. `ligar o fogão em fogo baixo;`
2. `separar 1 ovo, 1 colher de sobremesa de manteiga e sal a gosto;`
3. `quebrar o ovo em uma tigela;`
4. `colocar sal na tigela;`
5. `misturar levemente o ovo e o sal, com um garfo;`
6. `aquecer a manteiga na frigideira até que comece a derreter;`
7. `jogar o ovo na frigideira, mexendo com uma colher até ficar firme;`
8. `retirar da frigideira e servir.`

Se analisarmos o algoritmo acima, podemos observar que, embora o ovo mexido seja obtido, garantindo a eficácia da receita, existe uma clara ineficiência em relação ao gasto de gás, uma vez que ligar o fogão não é pré-requisito para a quebra do ovo e mistura do ovo e do sal. Já em outras ações, como as especificadas nos passos 3 e 4, a sequência não é relevante.

Se modificarmos apenas a sequência das ações, conforme indicado abaixo, então teremos um algoritmo eficaz e mais eficiente:

1. `separar 1 ovo, 1 colher de sobremesa de manteiga e sal a gosto;`
2. `quebrar o ovo em uma tigela;`
3. `colocar sal nesta tigela;`
4. `misturar levemente o ovo e o sal, com um garfo;`
5. `ligar o fogão em fogo baixo;`
6. `aquecer a manteiga na frigideira até que comece a derreter;`
7. `jogar o ovo na frigideira, misturando com uma colher até ficar firme;`
8. `retirar da frigideira e servir.`

1.2 → etapas de construção de um programa

A construção de um algoritmo para dar suporte computacional a uma aplicação do mundo real deve ser feita com todo cuidado para que ele realmente execute as tarefas que se quer de forma correta e em tempo razoável. Programar não é uma atividade trivial, muito antes pelo contrário, requer muito cuidado e atenção. A dificuldade em gerar bons programas levou à definição de técnicas específicas que iniciam frequentemente com a construção de um algoritmo.

A forma mais simples de garantir a qualidade de um programa é construí-lo seguindo uma série de etapas. Parte-se de uma análise inicial da realidade envolvida na aplicação, desenvolvendo a solução de forma gradual, e chega-se ao produto final: um programa que executa as funcionalidades necessárias à aplicação.

A seguir, são explicadas as etapas que devem ser cumpridas para assegurar a construção de um programa correto (Figura 1.6). Observe que este processo não é puramente sequencial, mas, em cada etapa, pode ser necessário voltar a alguma etapa anterior para desenvolver com mais detalhes algum aspecto.

- **análise detalhada do problema.** Inicia-se com uma análise detalhada do problema, identificando os aspectos que são relevantes para a sua solução. No Algoritmo 1.1, o problema é:

  `Informar a soma de dois valores.`

- **especificação dos requisitos do problema.** Nessa etapa são identificados e especificados os resultados que deverão ser produzidos (`saídas`) e os dados que serão necessários para a execução da tarefa requerida (`entradas`). No Algoritmo 1.1, os dados de entrada e saída são:

```
Entradas: dois valores numéricos, digitados via teclado.
Saída: a soma dos dois valores, mostrada na tela.
```

Esse é um problema simples, adequado à introdução dos conceitos iniciais. Contudo, na prática, não apenas os objetivos podem ser mais complexos como a identificação de entradas e saídas pode incluir formatos e valores válidos, bem como quantidades de valores e a especificação de outros dispositivos de entrada e saída.

- **construção de um algoritmo.** A etapa seguinte é o projeto de um algoritmo que solucione o problema, ou seja, de um conjunto finito de ações que, quando executadas na ordem estabelecida, levem ao resultado desejado em um tempo finito. É importante notar que mesmo os problemas mais simples tendem a ter mais de uma solução possível, devendo ser determinada a solução que será adotada. Nesta etapa já devem ser criados os nomes de variáveis que irão armazenar os valores de entrada e os valores gerados durante o processamento. O Algoritmo 1.1. representa uma possível solução alcançada nesta etapa.
- **validação do algoritmo.** Em seguida, deve ser feita a validação lógica do algoritmo desenvolvido. Essa validação muitas vezes é feita através de um teste de mesa, ou seja, simulando sua execução com dados virtuais. Procura-se, através desses testes, verificar se a solução proposta atinge o objetivo. Devem ser feitos testes tanto com valores corretos como incorretos. No exemplo que está sendo utilizado aqui, os dados para testes devem incluir valores nulos, positivos e negativos, como por exemplo:

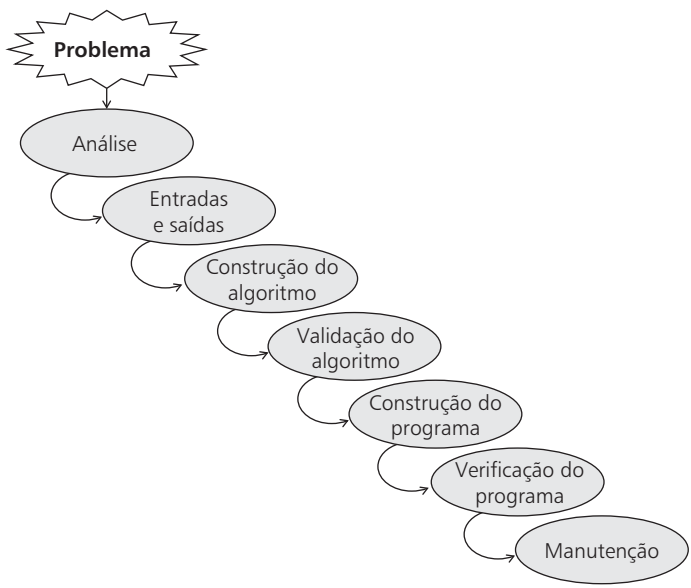


figura 1.6 Etapas da construção de um programa.

Valor1	Valor2	Soma
0	0	0
26	12	38
-4	-10	-14
12	-10	2
-5	2	-3

- **codificação do programa.** É a tradução do algoritmo criado para resolver o problema para uma linguagem de programação. Os programas em Pascal e C gerados a partir do algoritmo desenvolvido para o exemplo (Algoritmo 1.1) são apresentados no Capítulo 3.
- **verificação do programa.** Consiste nas verificações sintática (compilação) e semântica (teste e depuração) do programa gerado. Os mesmos valores utilizados no teste de mesa podem ser utilizados para testar o programa gerado.
- **manutenção.** Uma vez considerado pronto, o programa passa a ser utilizado por usuários. A etapa de manutenção do programa inicia no momento em que ele é liberado para execução, e acompanha todo seu tempo de vida útil. A manutenção tem por finalidade corrigir eventuais erros detectados, assim como adicionar novas funcionalidades.

Cada uma dessas fases é importante, devendo ser respeitada e valorizada para se chegar a programas de qualidade. Neste sentido, aqui são fornecidos subsídios a fim de que todas as etapas sejam consideradas durante a escrita de programas como, por exemplo, a indicação de valores que devem ser utilizados nos testes de cada comando e conselhos para deixar os programas legíveis, de modo a facilitar a sua manutenção.

1.3 → paradigmas de programação

O programa realmente executado por um computador é escrito em uma linguagem compreendida pela máquina, por isso denominada linguagem de máquina, na qual as instruções são codificadas no sistema de numeração binário. A utilização direta de linguagem de máquina é bastante complicada. Para tornar a escrita de programas mais acessível a usuários comuns foram desenvolvidas linguagens de mais alto nível, denominadas linguagens de programação. São linguagens que permitem a especificação das instruções que deverão ser executadas pelo computador através de uma linguagem mais próxima da linguagem natural.

Um programa escrito numa linguagem de programação, denominado programa-fonte, deve ser primeiro traduzido para linguagem de máquina, para só então ser executado pelo computador. A tradução do programa-fonte para o programa em linguagem de máquina correspondente é feita por um outro programa, específico para a linguagem utilizada, denominado compilador (Figura 1.7).

Na busca de uma forma simples, clara e precisa de escrever programas, diversas linguagens de programação foram desenvolvidas nos últimos anos. Toda linguagem possui uma sintaxe bem definida, que determina as construções corretas a serem utilizadas para a elaboração de programas. Além disso, cada linguagem de programação utiliza um conjunto de concei-

tos adotados na solução de problemas, o qual corresponde à semântica desta linguagem, ou seja, à forma como construções sintaticamente corretas são executadas. Esses conceitos possibilitam diferentes abordagens de problemas e formulações de soluções, isto é, seguem diferentes **paradigmas de programação**.

A palavra "paradigma" corresponde a um modelo ou padrão de como uma realidade é entendida e de como se interage com essa realidade. Aqui, um paradigma de programação corresponde à forma como a solução está estruturada e será executada no programa gerado, incluindo técnicas e conceitos específicos, bem como os recursos disponibilizados. Os principais paradigmas das linguagens de programação são (Ghezzi; Jazayeri, 1987; Melo, Silva, 2003; Sebesta, 2003):

- **imperativo ou procedural**, no qual um programa é composto por uma sequência de comandos a serem executados pelo computador em uma determinada ordem. Dentre as linguagens de programação voltadas a esse paradigma destacam-se Pascal, C, Fortran, Cobol, PL/1, Basic, Algol, Modula e Ada, entre outras;
- **funcional**, em que um programa é composto pela declaração de funções que transformam a(s) entrada(s) na(s) saída(s) desejada(s). Exemplos de linguagens funcionais são Lisp, ML, Miranda, Haskell e OCaml;
- **lógico**, que utiliza a avaliação de condições lógicas como base para a escrita dos programas. Um programa é composto por regras que disparam ações a partir da identificação de premissas. Um exemplo desse paradigma é a linguagem Prolog;
- **orientação a objetos**, em que o mundo real é representado por meio de classes de objetos e das operações que podem ser realizadas sobre eles, as quais definem seu comportamento. Herança e polimorfismo são conceitos básicos adotados nesse paradigma. Smalltalk, C++, Java, PascalOO, Delphi, C#, Eiffel e Simula são exemplos de linguagens orientadas a objetos.

A forma de escrever um programa em cada um desses paradigmas é bastante diferente. Neste livro será considerado somente o **paradigma imperativo** ou **procedural**. Essa opção, para um primeiro curso em programação, justifica-se pelas seguintes razões:

- o paradigma imperativo permite representar de uma forma intuitiva os problemas do dia a dia, que geralmente são executados através de sequências de ações;

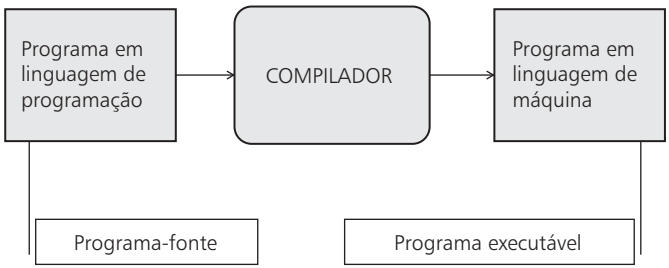


figura 1.7 Tradução de programa-fonte para executável.

- historicamente, os primeiros programas foram desenvolvidos utilizando linguagens imperativas, sendo esse um paradigma dominante e bem estabelecido;
- existe um grande número de algoritmos e de sistemas implementados em linguagens que seguem esse paradigma, os quais podem ser utilizados como base para o desenvolvimento de novos programas.

A opção de utilizar as linguagens Pascal e C neste livro deu-se por serem essas as linguagens mais utilizadas como introdutórias à programação na maior parte dos cursos brasileiros de ciência da computação, informática e engenharia da computação.

A **linguagem Pascal** foi definida por Niklaus Wirth em 1970 (Wirth, 1971, 1972, 1978) com a finalidade de ser utilizada em aplicações de propósito geral e, principalmente, para ensino de programação. Uma característica importante de Pascal é que foi, desde sua criação, pensada para dar suporte à programação estruturada. Pascal serviu de base para o desenvolvimento de diversas outras linguagens de programação (Ghezzi; Jazayeri, 1987). Portanto, o aprendizado de novas linguagens de programação, sobretudo as que seguem o paradigma imperativo, se torna mais fácil para quem conhece Pascal.

A **linguagem C** foi desenvolvida por Dennis Ritchie nos anos 1970 (Kernighan; Ritchie, 1988) com o propósito de ser uma linguagem para a programação de sistemas. É hoje largamente utilizada em universidades e no desenvolvimento de *softwar*e básico.

1.4 → programação estruturada

A **programação estruturada** (Jackson, 1975) pode ser vista como um subconjunto do paradigma imperativo. Baseia-se no princípio de que o fluxo do programa deve ser estruturado, devendo esse fluxo ficar evidente a partir da estrutura sintática do programa. A estruturação deve ser garantida em dois níveis: de comandos e de unidades.

No nível de comandos, a programação estruturada fundamenta-se no princípio básico de que um programa deve possuir um único ponto de entrada e um único ponto de saída, existindo de "1 a n" caminhos definidos desde o princípio até o fim do programa e sendo todas as instruções executáveis, sem que apareçam repetições (*loops*) infinitas de alguns comandos. Nesse ambiente, o programa deve ser composto por blocos elementares de instruções (comandos), interconectados através de apenas três mecanismos de controle de fluxo de execução: sequência, seleção e iteração. Cada bloco elementar, por sua vez, é delimitado por um ponto de início – necessariamente no topo do bloco – e por um ponto de término – necessariamente no fim do bloco – de execução, ambos muito bem definidos. Os três mecanismos de controle do fluxo de execução estão representados na Figura 1.8 através de fluxogramas, nos quais se observa claramente os pontos de entrada e de saída de cada bloco de instruções. Alguns dos blocos que constam nessa figura serão vistos nos próximos capítulos deste livro.

Uma característica fundamental da programação estruturada é que o uso de desvios incondicionais no programa, implementados pelo comando `GOTO` (`VÁ PARA`), é totalmente proibido.

Embora esse tipo de comando, em alguns casos, possa facilitar a construção de um programa, dificulta enormemente sua compreensão e manutenção.

No nível de unidades, a programação estruturada baseia-se na ideia proposta em 1972 pelo cientista de computação E. W. Dijkstra: "A arte de programar consiste na arte de organizar e dominar a complexidade dos sistemas". A programação estruturada enfatiza a utilização de unidades separadas de programas, chamadas de módulos, que são ativadas através de comandos especiais. Propõe que os programas sejam divididos em um conjunto de subprogramas menores, cada um com seu objetivo específico e bem definido, mais fáceis de implementar e de testar (seguindo a tática de "dividir para conquistar").

O desenvolvimento de programas deve ser feito de forma descendente, com a decomposição do problema inicial em módulos ou estruturas hierárquicas, de modo a dividir ações complexas em uma sequência de ações mais simples, desenvolvidas de forma mais fácil. Essa técnica decorre da programação estruturada e é também conhecida como **programação modular**.

Resumindo, a programação estruturada consiste em:

- uso de um número muito limitado de estruturas de controle;
- desenvolvimento de algoritmos por fases ou refinamentos sucessivos;
- decomposição do algoritmo total em módulos.

Essas técnicas para a solução de problemas visam à correção da solução desenvolvida, bem como à simplicidade dessa solução, garantindo uma melhor compreensão do que é feito e facilitando a manutenção dos programas por outras pessoas além do desenvolvedor inicial.

Neste texto será utilizada a programação estruturada, incentivando o desenvolvimento de programas através de módulos, de forma a garantir a qualidade dos programas construídos (Farrer et al., 1999). Seguindo os preceitos da programação estruturada, comandos do tipo `GOTO` (`VÁ PARA`), que alteram o fluxo de execução incondicionalmente, não serão tratados neste livro.

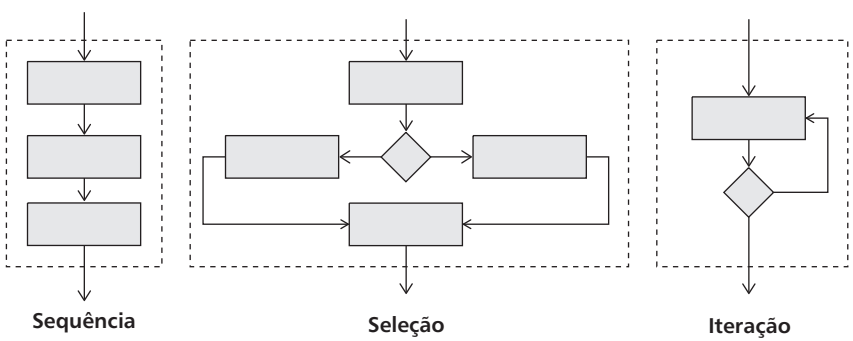


figura 1.8 Estruturas de controle de fluxo de execução na programação estruturada.

1.5 → elementos de representação interna de dados

Internamente, os computadores digitais operam usando o sistema numérico binário, que utiliza apenas os símbolos 0 e 1. Na memória e nos dispositivos de armazenamento, o componente conceitual básico e a menor unidade de armazenamento de informação é o bit. Bit vem do Inglês *binary digit*, ou seja, dígito binário, e um bit pode memorizar somente um entre dois valores: zero ou um. Qualquer valor numérico pode ser expresso por uma sucessão de bits usando o sistema de numeração binário.

Para representar caracteres, são utilizados códigos armazenados em conjuntos de bits. Os códigos mais comuns armazenam os caracteres em *bytes*, que são conjuntos de 8 bits. Nos códigos de representação de caracteres, cada caractere tem associado a si, por convenção, uma sequência específica de zeros e uns. Três códigos de representação de caracteres são bastante utilizados: ASCII (7 bits por caractere), EBCDIC (8 bits por caractere) e UNICODE (16, 32 ou mais bits).

Tanto o ASCII (*American Standard Code for Information Interchange*), que é o código utilizado pela maioria dos microcomputadores e em alguns periféricos de equipamentos de grande porte, quanto o EBCDIC (*Extended Binary Coded Decimal Interchange Code*) utilizam um *byte* para representar cada caractere, sendo que, na representação do conjunto de caracteres ASCII padrão, o bit mais significativo (bit mais à esquerda) do *byte* é sempre igual a 0. A representação dos caracteres A e Z nos dois códigos é:

Caracteres	EBCDIC	ASCII
A	1100 0001	0100 0001
Z	1110 1001	0101 1010

O UNICODE é promovido e desenvolvido pelo *Unicode Consortium*. Busca permitir aos computadores representar e manipular textos de forma consistente nos múltiplos sistemas de escrita existentes. Atualmente, ele compreende mais de 100.000 caracteres. Dependendo do conjunto de caracteres que esteja em uso em uma aplicação, um, dois ou mais *bytes* podem ser utilizados na representação dos caracteres.

As unidades de medida utilizadas para quantificar a memória principal e indicar a capacidade de armazenamento de dispositivos são:

K	quilo	(mil)	10^3
M	mega	(milhão)	10^6
G	giga	(bilhão)	10^9
T	tera	(trilhão)	10^{12}

O sistema métrico de unidades de medida utiliza os mesmos prefixos, mas o valor exato de cada um deles em informática é levemente superior. Como o sistema de numeração utilizado

internamente em computadores é o binário (base 2), as capacidades são representadas como potências de 2:

K	1.024	2^{10}
M	1.048.576	2^{20}
		etc...

A grafia dos valores expressos em múltiplos de *bytes* pode variar. Assim, por exemplo, 512 quilobytes podem ser escritos como 512K, 512KB, 512kB ou 512Kb. Já os valores expressos em bits, via de regra, são escritos por extenso, como em 512 quilobits.

1.6 → dicas

Critérios que devem ser observados ao construir um algoritmo:

- procurar soluções simples para proporcionar clareza e facilidade de entendimento do algoritmo;
- construir o algoritmo através de refinamentos sucessivos;
- seguir todas as etapas necessárias para a construção de um algoritmo de qualidade;
- identificar o algoritmo, definindo sempre um nome para ele no cabeçalho. Este nome deve traduzir, de forma concisa, seu objetivo. Por exemplo: `Algoritmo 1.1 – Soma2` indica, através do nome, que será feita a soma de dois valores;
- definir, também no cabeçalho, o objetivo do algoritmo, suas entradas e suas saídas;
- nunca utilizar desvios incondicionais, como `GOTO` (`VÁ PARA`).

1.7 → testes

Testes de mesa. É importante efetuar, sempre que possível, testes de mesa para verificar a eficácia (corretude) de um algoritmo antes de implementá-lo em uma linguagem de programação. Nestes testes, deve-se utilizar diferentes conjuntos de dados de entrada, procurando usar dados que cubram a maior quantidade possível de situações que poderão ocorrer durante a utilização do algoritmo. Quando o algoritmo deve funcionar apenas para um intervalo definido de valores, é recomendável que se simule a execução para valores válidos, valores limítrofes válidos e inválidos e valores inválidos acima e abaixo do limite estabelecido. Por exemplo, se um determinado algoritmo deve funcionar para valores inteiros, no intervalo de 1 a 10, inclusive, o teste de mesa deveria incluir a simulação da execução para, no mínimo, os valores 0, 1, 10, 11 e um valor negativo.


Enter
Shift
Start
menu
Ctrl

capítulo 2

unidades léxicas, variáveis, constantes e expressões

Este capítulo apresenta as unidades léxicas de linguagens de programação imperativas, como Pascal e C. Discute as declarações de variáveis, de constantes e de tipos, bem como a representação de expressões aritméticas e lógicas. Neste capítulo, e nos que o sucedem, todos os conceitos são apresentados e analisados inicialmente em linguagem algorítmica, sendo a seguir comentados e exemplificados em Pascal e C.

Para que um algoritmo se transforme em um programa executável, é necessário que esse seja inicialmente traduzido para uma linguagem de programação pelo compilador correspondente, que irá gerar o programa a ser executado. Essa tradução é feita com base na gramática da linguagem. Nesse processo, cada símbolo, cada palavra e cada construção sintática utilizados no programa devem ser reconhecidos pelo compilador. Isso é possível porque toda linguagem de programação possui uma gramática bem definida que rege a escrita dos programas, ou seja, que define sua sintaxe.

A primeira representação da gramática de uma linguagem de programação foi apresentada por John Backus, em 1959, para expressar a gramática da linguagem Algol. Esta notação deu origem à **BNF** (**Backus-Naur Form** ou **Backus Normal Form**) (Knuth, 2003; Wiki, 2012), que se tornou a forma mais utilizada para representar a gramática de linguagens de programação. Neste livro, é utilizada uma forma simplificada da BNF (ver Apêndice) para representar a gramática da pseudolinguagem e das linguagens Pascal e C.

As gramáticas das linguagens de programação imperativas são bastante parecidas no que se refere a unidades léxicas e comandos disponíveis. Esses elementos são apresentados a partir deste capítulo, que inicia apresentando as unidades léxicas de linguagens de programação imperativas. A seguir, ele mostra como devem ser feitas as declarações de variáveis, de constantes e de tipos, incluindo a análise de diferentes tipos de variáveis e dos valores que podem conter. Por fim, esse capítulo apresenta as expressões aritméticas e lógicas e suas representações. Outros tipos de declarações serão vistos mais adiante neste livro. Todos os conceitos são apresentados e analisados na linguagem algorítmica, sendo depois traduzidos para as linguagens de programação Pascal e C.

2.1 ⇢ componentes das linguagens de programação

Os componentes básicos de uma linguagem de programação são denominados **unidades léxicas**. As unidades léxicas mais simples, analisadas a seguir, são valores literais, identificadores, palavras reservadas, símbolos especiais e comentários.

2.1.1 literais

Literais são valores representados explicitamente no programa e que não mudam durante a execução. Podem ser números, valores lógicos, caracteres ou *strings*.

números. Usualmente é utilizada a notação decimal para representar números nos programas, embora se saiba que internamente eles sejam representados na forma binária. Podem ser utilizados valores numéricos inteiros ou fracionários (chamados de reais), positivos ou negativos. Os números são representados na linguagem algorítmica exatamente como aparecem nas expressões aritméticas em português.

Ex.: 123 –45 +6,7

As linguagens de programação geralmente também permitem uma forma alternativa de escrita, mais compacta, de números muito grandes ou muito pequenos, denominada notação

exponencial, científica ou de ponto flutuante. Nessa notação, um número real é representado por um valor inteiro (denominado mantissa) multiplicado por potências de 10 (indicadas pelo seu expoente). Por exemplo, o valor 3.000.000.000.000 seria representado como 3×10^{11}. Tanto a mantissa como o expoente podem ter sinal (positivo ou negativo). Cada linguagem de programação define uma forma para a representação de números em notação exponencial, conforme será visto nas seções correspondentes a Pascal e C.

valores lógicos. Os valores lógicos (ou booleanos) `verdadeiro` e `falso` podem ser utilizados diretamente nos programas quando for feita alguma comparação.

caracteres. Permitem representar um símbolo ASCII qualquer, como uma letra do alfabeto, um dígito numérico (aqui, sem conotação quantitativa, apenas como representação de um símbolo) ou um caractere especial (um espaço em branco também corresponde a um caractere especial). Nos programas, os caracteres são geralmente representados entre apóstrofos. Essa é também a forma utilizada na pseudolinguagem aqui empregada.

Ex.: 'A' 'b' '4' '+'

***strings*.** São sequências de um ou mais caracteres. Quaisquer caracteres podem ser utilizados (letras, dígitos e símbolos), incluindo o símbolo que representa um espaço em branco. *Strings* normalmente são representadas entre apóstrofos em um programa, forma também utilizada na pseudolinguagem.

Ex.: 'Ana Maria' 'A12B3' 'a$b' '91340-330/1'

2.1.2 identificadores

São as palavras criadas pelo programador para denominar o próprio programa ou elementos dentro do mesmo, tais como: variáveis, constantes ou subprogramas. Toda linguagem de programação define regras específicas para a formação de **identificadores**, para que eles possam ser reconhecidos pelo compilador.

Na pseudolinguagem utilizada neste livro, um identificador deve sempre iniciar por uma letra, seguida de qualquer número de letras e dígitos, incluindo o símbolo "_" (sublinhado), por ser essa a forma mais frequentemente utilizada em linguagens de programação. Tratando-se de uma pseudolinguagem, a acentuação e a letra "ç" também podem ser utilizadas de forma a traduzir de forma mais fiel os valores que devem ser representados. A pseudolinguagem não diferencia letras maiúsculas de minúsculas, mas se recomenda que sejam usadas apenas minúsculas nos nomes de identificadores, reservando as maiúsculas para iniciais e para casos específicos que serão destacados oportunamente.

Exemplos de identificadores:

```
valor
número1
a7b21
Nome_Sobrenome
```

2.1.3 palavras reservadas

São identificadores que têm um significado especial na linguagem, representando comandos e operadores, ou identificando subprogramas já embutidos na linguagem. As **palavras reservadas** não podem ser utilizadas como identificadores definidos pelo programador.

Algumas das palavras reservadas definidas na pseudolinguagem são:

```
início
fim
se
então
escrever
ler
função
```

2.1.4 símbolos especiais

Símbolos especiais servem para delimitar ações, separar elementos, efetuar operações ou indicar ações específicas. Na pseudolinguagem aqui adotada, também existem alguns símbolos especiais com significado específico, como:

```
←     +     (     )     <     >     :     ;
```

2.1.5 comentários

Comentários são recursos oferecidos pelas linguagens de programação que permitem, por exemplo, a inclusão de esclarecimentos sobre o que o programa faz e como isso é feito. Os comentários são identificados e delimitados por símbolos especiais e podem compreender quaisquer sequências de caracteres. Todo o conteúdo compreendido entre os símbolos delimitadores de comentários é ignorado pelo compilador durante a tradução do programa, servindo apenas para documentar e facilitar o entendimento pelos seres humanos que tiverem acesso ao código do programa. Na pseudolinguagem, os comentários são delimitados pelos símbolos "{" e "}".

Exemplo de comentário: `{ Este é um comentário @#$% }`

2.2 → declarações

Todos os itens utilizados em um programa devem ser declarados antes de sua utilização. Os nomes e as características desses itens são definidos através de **declarações**. Nesta seção, são analisadas declarações de variáveis, de tipos de dados e de constantes. Outras declarações são vistas em capítulos subsequentes.

2.2.1 declaração de variáveis

Uma variável representa um espaço de memória identificado e reservado para guardar um valor durante o processamento. Ressalte-se que somente um valor pode estar armazenado em uma variável em um determinado momento. Caso seja definido um novo valor para uma variável, o anterior será perdido.

Sempre que um programador decidir utilizar uma variável em seu programa, ele deverá informar seu nome e o tipo de valores que ela irá armazenar. Isso faz com que, ao final da compilação do programa, exista um espaço reservado para essa variável na memória principal do computador, com um determinado endereço físico. O tamanho do espaço alocado para a variável depende do tipo definido para ela. Uma vez alocada a variável, ela passa a ser referenciada no programa através do nome dado pelo programador, não sendo necessário saber seu endereço físico.

Variáveis de dois tipos, bastante diferentes no seu conteúdo e forma de uso, podem ser utilizadas em um programa: (1) variáveis que armazenam os valores manipulados no programa e (2) variáveis que guardam endereços físicos de memória, denominadas ponteiros. Esse segundo tipo de variável será tratado mais adiante, no Capítulo 14. Até lá, sempre que forem feitas referências a variáveis se estará tratando das que armazenam valores e não endereços de memória.

Toda variável utilizada pelo programa deve ser declarada no seu início, através de uma **declaração de variáveis**, em que são definidos seu nome e o tipo de dados que poderá armazenar.

Os **tipos de dados** utilizados nas linguagens de programação se classificam, de acordo com os valores que podem armazenar, em:

- tipos simples:
  - numéricos;
  - alfanuméricos;
  - lógicos ou booleanos;
  - ponteiros.
- tipos compostos:
  - arranjos;
  - registros;
  - enumerações;
  - conjuntos;
  - arquivos.

Inicialmente serão analisados somente os três primeiros tipos de dados simples. O tipo ponteiro e os tipos compostos serão gradualmente apresentados ao longo deste livro.

Os nomes dados aos tipos de dados simples na pseudolinguagem são:

- **`inteiro`**, para armazenar somente valores numéricos inteiros;
- **`real`**, em que são armazenados valores numéricos fracionários;
- **`caractere`**, para armazenar somente um caractere alfanumérico, utilizando a codificação de caracteres ASCII, que representa qualquer caractere em 8 bits;

- **string**, em que são armazenadas cadeias de caracteres alfanuméricos;
- **lógico**, para variáveis que podem armazenar somente um dos dois valores lógicos, `verdadeiro` ou `falso`.

Uma opção na declaração de uma variável simples do tipo `string` é definir o número de caracteres que poderá conter por meio de um inteiro entre colchetes. Por exemplo, uma variável definida com o tipo `string[3]` poderá conter somente três caracteres, enquanto que uma variável definida com o tipo `string`, sem limitação de tamanho, poderá ter o número de caracteres permitido na linguagem de programação utilizada.

No Capítulo 1 ressaltou-se a importância de identificar os valores de entrada e de saída de um algoritmo. Esses valores são armazenados em variáveis. Na pseudolinguagem, sugere-se que as variáveis sejam definidas em conjuntos separados, identificando (1) as variáveis de entrada, que servirão para valores fornecidos ao programa, (2) as de saída, que vão armazenar os valores que serão informados pelo programa, resultantes de seu processamento, e (3) as variáveis auxiliares, que servirão somente para guardar valores durante o processamento. A sintaxe da declaração de variáveis é a seguinte:

```
Entradas: <lista de nomes de variáveis com seus tipos>
Saídas: <lista de nomes de variáveis com seus tipos>
Variáveis auxiliares: <lista de nomes de variáveis com seus tipos>
```

Os nomes escolhidos pelo programador para cada variável devem ser seguidos do tipo da variável entre parênteses:

```
nome (string)
valor (real)
```

Várias variáveis do mesmo tipo podem ser agrupadas em uma lista de nomes separados por vírgula, seguida pelo tipo correspondente:

```
int1, int2, int3 (inteiro)
```

Um exemplo do cabeçalho de um algoritmo, incluindo as declarações das variáveis já identificadas conforme sua futura utilização, é mostrado a seguir:

```
Algoritmo - MédiaEMaiorValor
 {INFORMA A MÉDIA DE 2 VALORES E QUAL O MAIOR DELES}
  Entradas: valor1, valor2 (real)
  Saídas: média (real)
          maior (real)
  Variáveis auxiliares: aux (real)
```

2.2.2 declaração de tipos de dados

As linguagens de programação geralmente permitem a **definição de novos tipos de dados**. Um novo tipo é identificado através de um nome dado pelo programador. Sua definição se baseia em um tipo-base, anteriormente definido ou predefinido na linguagem. Na

Encerra aqui o trecho do livro disponibilizado para esta Unidade de Aprendizagem. Na Biblioteca Virtual da Instituição, você encontra a obra na íntegra.