INOVAÇÃO
E TECNOLOGIA

unidade
1

Algorítimos e PROGRAMAÇÃO

UNIVERSIDADE LaSalle EAD

Introdução a Algoritmos de Programação

Prezado(a) estudante

Estamos começando uma unidade desta disciplina. Os textos que a compõem foram organizados com cuidado e atenção, para que você tenha contato com um conteúdo completo e atualizado tanto quanto possível. Leia com dedicação, realize as atividades e tire suas dúvidas com os tutores. Dessa forma, você com certeza alcançará os objetivos propostos para essa disciplina.

OBJETIVO GERAL

Conceituar, identificar, exemplificar e codificar com a utilização da ferramenta VisuAlg, ordenar dados, realizar comandos condicionais simples, compostos e de múltiplas escolha.

OBJETIVOS ESPECÍFICOS

- Definir o conceito de algoritmos e lógica de programação.
- Diferenciar os componentes básicos de funcionamento de um computador na execução de algoritmos.
- Aplicar as etapas de construção de um algoritmo na solução de problemas.
- Identificar as diferentes formas de representação de algoritmos.
- Construir algoritmos básicos utilizando linguagem descritiva, fluxogramas e pseudocódigo.
- Resolver problemas através da construção de algoritmos.
- Reconhecer a representação interna de dados em um computador.
- Diferenciar variáveis e constantes.
- Demonstrar de forma correta os tipos de dados na solução de problemas.
- Construir a tabela verdade dos conectivos lógicos e, ou e não.
- Usar os conectivos e, ou e não na avaliação de proposições.
- Avaliar proposições simples e compostas.
- Construir equações matemáticas na forma algorítmica.

OBJETIVOS ESPECÍFICOS

- Aplicar corretamente os operadores aritméticos, relacionais e lógicos na solução de problemas.
- Resolver corretamente os operadores aritméticos, relacionais e lógicos na solução de problemas.
- Analisar algoritmos simples com comandos de entrada e saída em pseudolinguagem.
- Identificar algoritmos simples com comandos de entrada e saída em fluxograma.
- Aplicar comandos de entrada e saída na solução de problemas.
- Identificar os sistemas de numeração decimal, binário, octal e hexadecimal.
- Converter os sistemas de numeração.
- Reconhecer o sistema de medida na computação, assim como sua importância na prática.
- Identificar a estrutura básica de um algoritmo sequencial em fluxograma.
- Analisar algoritmos sequenciais em forma de fluxograma.
- Construir algoritmos sequenciais através de fluxogramas.
- Analisar algoritmos sequenciais em forma de fluxograma.
- Desenvolver algoritmos sequenciais através de fluxogramas.
- Usar os fluxogramas para representar a resolução de problemas.
- Reconhecer a estrutura básica de um algoritmo sequencial em pseudocódigo/pseudolinguagem.
- Analisar algoritmos sequenciais em forma de pseudocódigo/pseudolinguagem.
- Construir algoritmos sequenciais em pseudocódigo/pseudolinguagem.

Parte 1

Conceitos Básicos e Tipos de Dados

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.

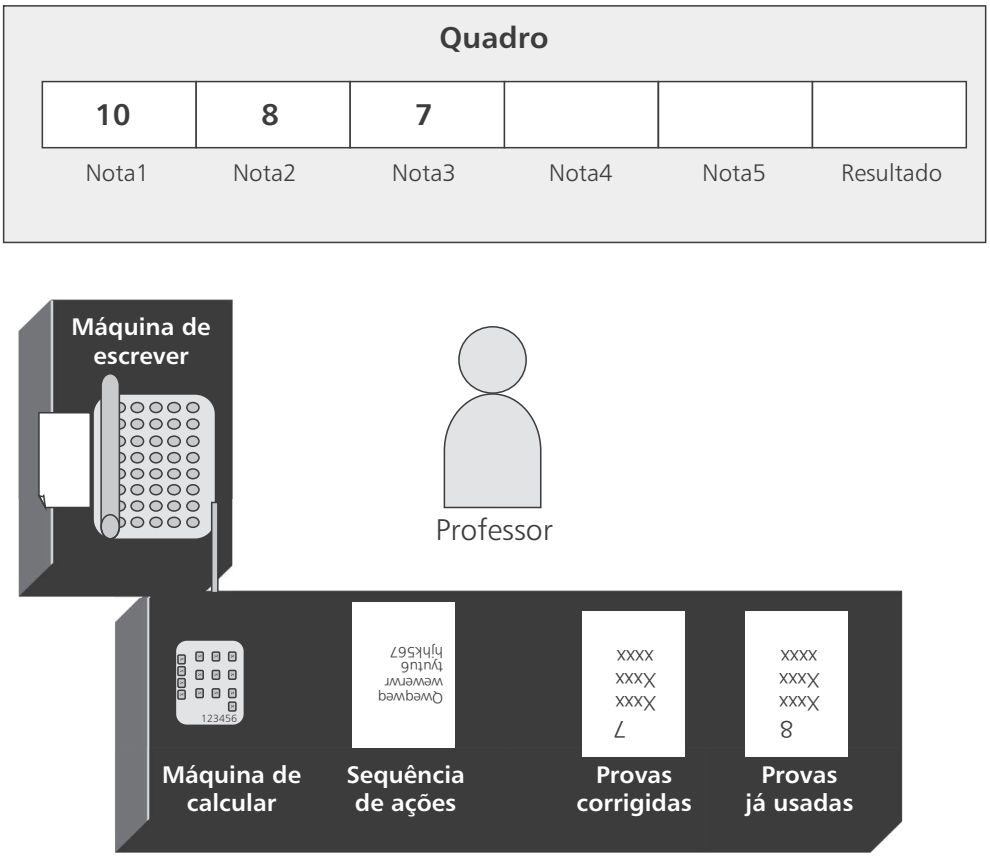


figura 1.1 Simulação de um algoritmo.

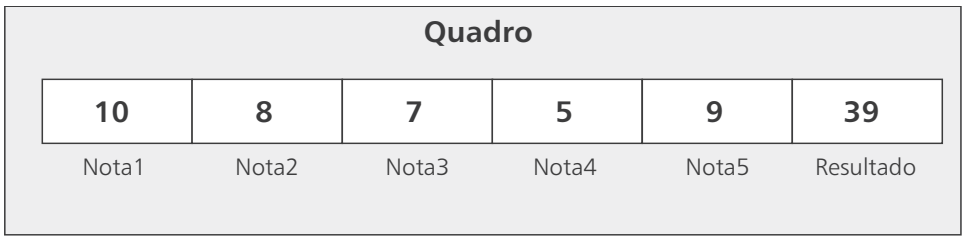
Obtidas todas as notas das provas, o professor passa a realizar as operações que vão calcular a média. Inicialmente, precisa somar todas as notas. Em cima da sua mesa, está uma calculadora. Ele consulta cada um dos valores das notas que escreveu no quadro e utiliza essa calculadora para fazer sua soma:

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

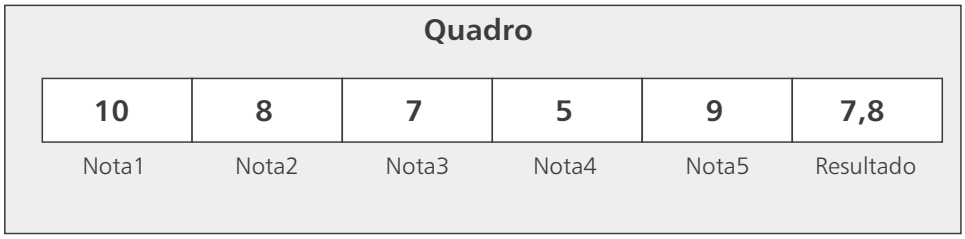
O resultado da soma ele escreve no espaço que chamou de `Resultado` (Figura 1.2a).

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.



(a)



(b)

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

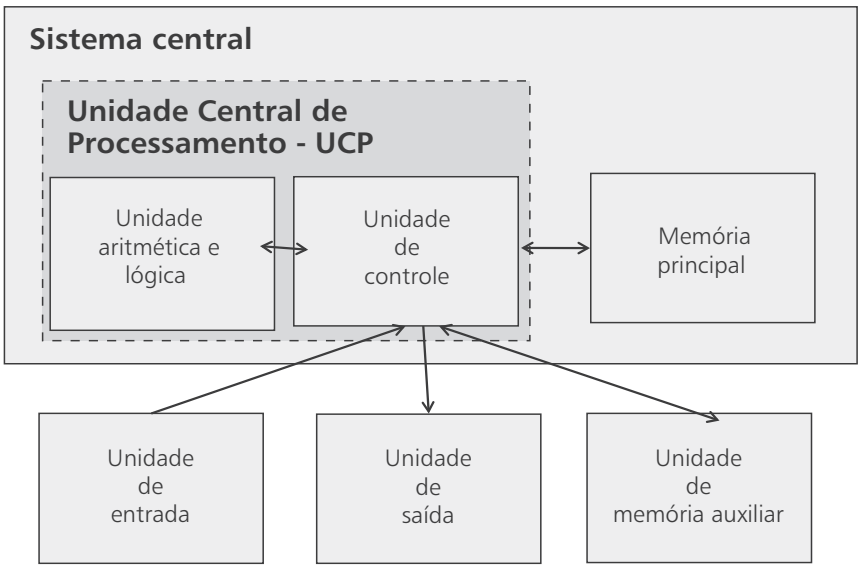


figura 1.3 Esquema simplificado de um computador.

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-

início	Ponto em que inicia a execução do algoritmo.
ENTRADA **lista de variáveis**	Entrada de dados: leitura de informações para preencher a lista de variáveis.
SAÍDA **lista de variáveis**	Saída de dados: informa conteúdos das variáveis da lista.
variável ← expressão	Atribuição: variável recebe o resultado da expressão.
fim	Ponto em que termina a execução do algoritmo.

figura 1.4 Blocos de fluxograma.

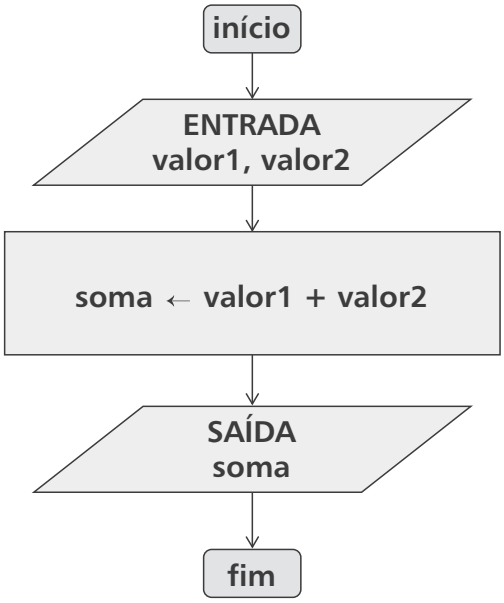


figura 1.5 Fluxograma da soma de dois números.

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 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:

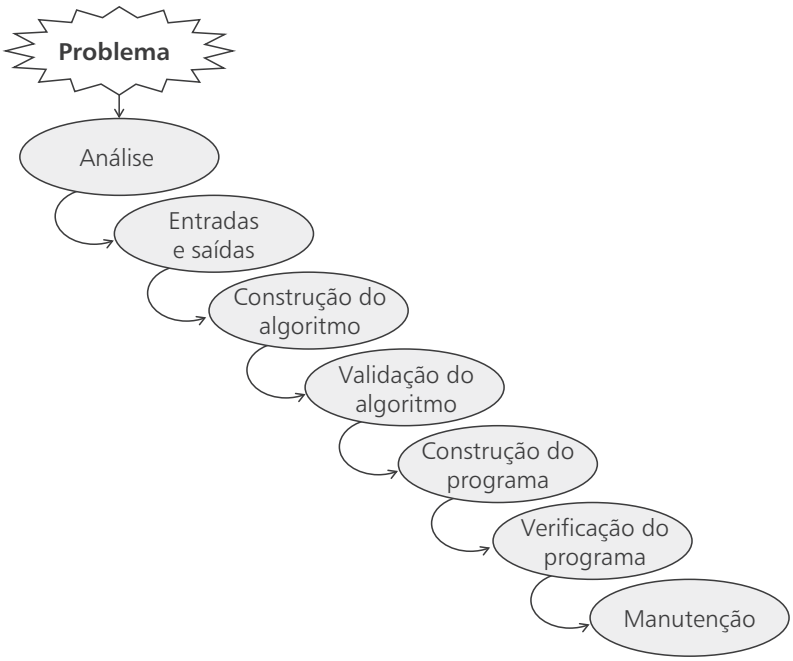


figura 1.6 Etapas da construção de um programa.

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;

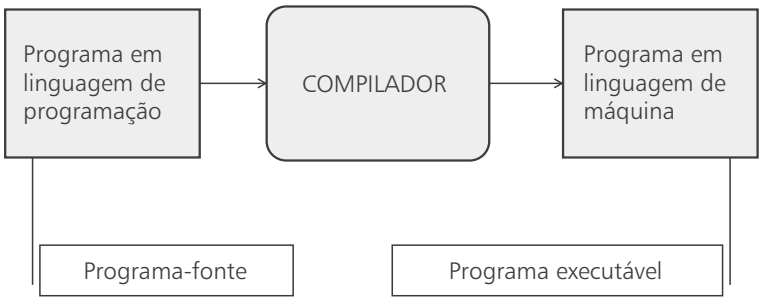


figura 1.7 Tradução de programa-fonte para executável.

- 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 *software* 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.

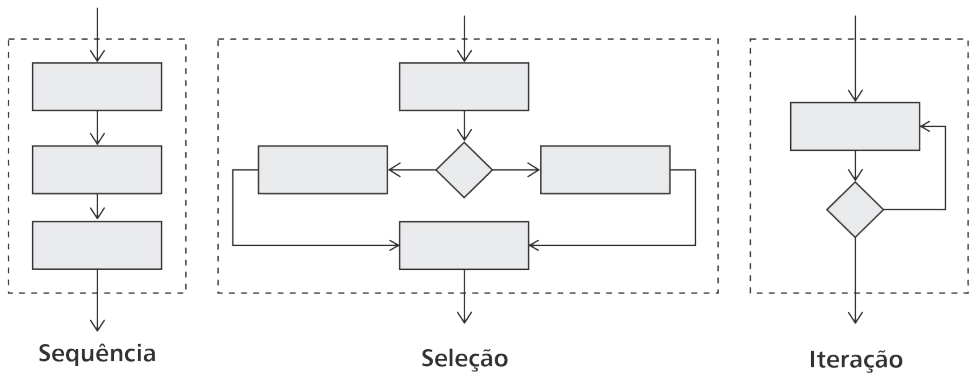


figura 1.8 Estruturas de controle de fluxo de execução na programação estruturada.

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.

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)
```

Parte 2

Representação de Problemas através de Algoritmos em Fluxograma e Descritiva

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-

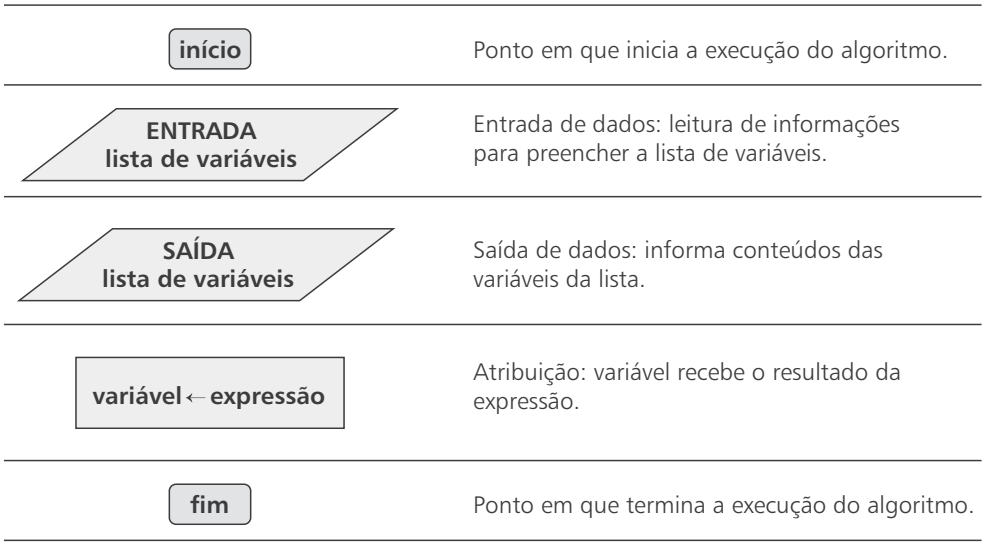


figura 1.4 Blocos de fluxograma.

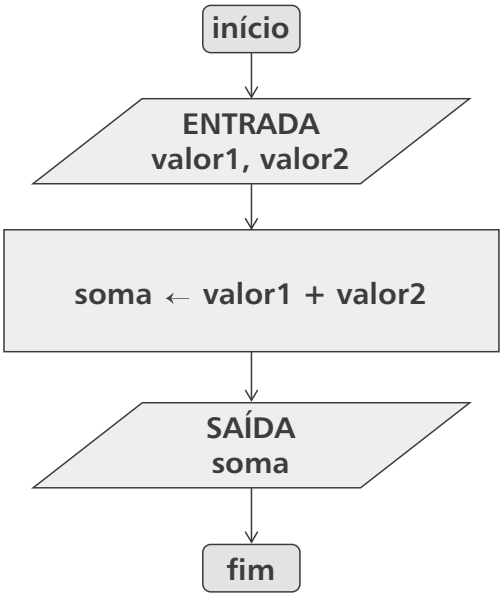


figura 1.5 Fluxograma da soma de dois números.

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.

Parte 3

Tabela Verdade e Lógica

Lógica e Cálculo Proposicional

4.1 INTRODUÇÃO

Muitos algoritmos e demonstrações usam expressões lógicas como:

"SE p ENTÃO q" ou "SE p_1 e p_2, ENTÃO q_1 OU q_2"

Logo, é necessário conhecer os casos nos quais essas expressões são VERDADEIRAS ou FALSAS, ou seja, saber o "valor verdade" de tais expressões. Discutimos essas questões neste capítulo.†

Também investigamos o valor verdade de afirmações quantificadas, as quais são expressões que empregam os quantificadores lógicos "para todo" e "existe".‡

4.2 PROPOSIÇÕES E SENTENÇAS COMPOSTAS

Uma *proposição* (ou *sentença*) é uma afirmação declarativa que é verdadeira ou falsa, mas não ambas. Considere, por exemplo, os seis itens a seguir:

(i) Gelo flutua na água. (iii) $2 + 2 = 4$ (v) Aonde você está indo?

(ii) A China é na Europa. (iv) $2 + 2 = 5$ (vi) Faça seu tema de casa.

Os quatro primeiros são proposições. Os dois últimos não. Além disso, (i) e (iii) são verdadeiras, mas (ii) e (iv) são falsas.

Proposições compostas

Muitas proposições são *compostas*, isto é, formadas por *subproposições* e vários conectivos discutidos a seguir. Tais sentanças são chamadas de *proposições compostas*. Uma proposição é denominada *primitiva* se não puder ser decomposta em proposições mais simples, ou seja, se não for composta.

Por exemplo, as proposições acima, de (i) a (iv), são primitivas. Por outro lado, as duas proposições a seguir são compostas:

"Rosas são vermelhas e violetas são azuis." e "John é esperto ou ele estuda todas as noites."

---

† N. de T.: É importante observar que os autores estão seguindo uma abordagem meramente intuitiva para o cálculo proposicional clássico. Do ponto de vista da lógica matemática, o objetivo do cálculo proposicional não é estabelecer o "valor verdade" de expressões.

‡ N. de T.: Normalmente o estudo de quantificadores se faz no cálculo de predicados de primeira ordem e não no cálculo proposicional.

A propriedade fundamental de uma proposição composta é que seu valor verdade é completamente determinado pelos valores verdade de suas subproposições, junto com a maneira como elas são conectadas para formar as proposições compostas. A seção a seguir explora alguns desses conectivos.

4.3 OPERAÇÕES LÓGICAS BÁSICAS

Esta seção discute as três operações lógicas básicas de conjunção, disjunção e negação, as quais correspondem, respectivamente, às palavras "e", "ou" e "não".

Conjunção, $p \wedge q$

Quaisquer duas proposições podem ser combinadas pela palavra "e" para formar uma proposição composta chamada de *conjunção* das proposições originais. Simbolicamente,

$$p \wedge q$$

que se lê "p e q", denota a conjunção de p e q. Como $p \wedge q$ é uma proposição, ela tem um valor verdade que depende apenas dos valores verdade de p e q. Especificamente:

Definição 4.1: Se p e q são verdadeiras, então $p \wedge q$ é verdadeira; caso contrário, $p \wedge q$ é falsa.

O valor verdade de $p \wedge q$ pode ser definido equivalentemente pela tabela na Fig. 4-1(a). Aqui a primeira linha é uma maneira abreviada de dizer que se p é verdadeira e q é verdadeira, então $p \wedge q$ é verdadeira. A segunda linha diz que se p é verdadeira e q é falsa, então $p \wedge q$ é falsa. E assim por diante. Observe que há quatro linhas correspondentes às quatro possíveis combinações de V e F para as duas subproposições p e q. Note que $p \wedge q$ é verdadeira apenas quando p e q são ambas verdadeiras.

p	q	$p \wedge q$
V	V	V
V	F	F
F	V	F
F	F	F

(a) "p e q"

p	q	$p \vee q$
V	V	V
V	F	V
F	V	V
F	F	F

(b) "p ou q"

p	$\neg p$
V	F
F	V

(c) "*não* p"

Figura 4-1

Exemplo 4.1 Considere as quatro proposições a seguir:

(i) Gelo flutua na água e $2 + 2 = 4$.
(ii) Gelo flutua na água e $2 + 2 = 5$.
(iii) China é na Europa e $2 + 2 = 4$.
(iv) China é na Europa e $2 + 2 = 5$.

Apenas a primeira é verdadeira. As outras são falsas, pois pelo menos uma de suas subproposições é falsa.

Disjunção, $p \vee q$

Duas proposições quaisquer podem ser combinadas pela palavra "ou" para formar uma proposição composta chamada de *disjunção* das proposições originais. Simbolicamente,

$$p \vee q$$

que se lê "p ou q", denota a disjunção de p e q. O valor verdade de $p \vee q$ depende apenas dos valores verdade de p e q como se segue.

Definição 4.2: Se p e q são falsas, então $p \vee q$ é falsa; caso contrário, $p \vee q$ é verdadeira.

O valor verdade de $p \vee q$ pode ser definido equivalentemente pela tabela na Fig. 4-1(b). Observe que $p \vee q$ é falsa apenas no quarto caso, quando p e q são ambas falsas.

Exemplo 4.2 Considere as quatro sentenças a seguir:

(i) Gelo flutua na água ou $2 + 2 = 4$.
(ii) Gelo flutua na água ou $2 + 2 = 5$.
(iii) China é na Europa ou $2 + 2 = 4$.
(iv) China é na Europa ou $2 + 2 = 5$.

Apenas a última sentença (iv) é falsa. As outras são verdadeiras, uma vez que pelo menos uma de suas subsentenças é verdadeira.

Observação: A palavra "ou" é comumente usada de duas maneiras distintas em português. Às vezes, é empregada no sentido de "p ou q, ou ambas", ou seja, pelo menos uma das duas alternativas acontece; e, às vezes, é usada no sentido de "p ou q, mas não ambas", isto é, somente uma das alternativas ocorre. Por exemplo, a afirmação "Ele irá para Harvard ou Yale" utiliza "ou" no último sentido, chamado eventualmente de *disjunção exclusiva*. A menos que seja estabelecido o contrário, "ou" deve ser empregado no primeiro sentido. Essa discussão aponta para a precisão conquistada em nossa linguagem simbólica: $p \vee q$ é definida por sua tabela verdade e *sempre* significa "p e/ou q".

Negação, $\neg p$

Dada qualquer sentença p, outra sentença, chamada de *negação* de p, pode ser formada escrevendo-se "Não é verdade que . . ." ou "É falso que . . ." antes de p ou, se possível, inserindo em p a palavra "não". Simbolicamente, a negação de p, que se lê "não p ", é denotada por

$$\neg p$$

O valor verdade de $\neg p$ depende do valor verdade de p como se segue:

Definição 4.3: Se p é verdadeira, então $\neg p$ é falsa; e se p é falsa, então $\neg p$ é verdadeira.

O valor verdade de $\neg p$ pode ser definido equivalentemente pela tabela da Fig. 4-1(c). Assim, o valor verdade da negação de p é sempre o oposto do valor verdade de p.

Exemplo 4.3 Considere as seis sentenças a seguir:

(a_1) Gelo flutua na água. (a_2) É falso que gelo flutua na água. (a_3) Gelo não flutua na água.
(b_1) $2 + 2 = 5$ (b_2) É falso que $2 + 2 = 5$. (b_3) $2 + 2 \neq 5$

Então, tanto (a_2) quanto (a_3) são a negação de (a_1); e tanto (b_2) quanto (b_3) são a negação de (b_1). Como (a_1) é verdadeira, (a_2) e (a_3) são falsas; e como (b_1) é falsa, (b_2) e (b_3) são verdadeiras.

Observação: A notação lógica para os conectivos "e", "ou" e "não" não é completamente padronizada. Por exemplo, alguns textos usam:

$$\begin{array}{rcl} p \,\&\, q,\ p \cdot q \text{ ou } pq & \text{para} & p \wedge q \\ p + q & \text{para} & p \vee q \\ p',\ \bar{p} \text{ ou } \sim p & \text{para} & \neg p \end{array}$$

4.4 PROPOSIÇÕES E TABELAS VERDADE

Seja $P(p, q, \ldots)$ uma expressão construída a partir de variáveis lógicas $p, q, \ldots$, que assumem o valor VERDADEIRO (V) OU FALSO (F), e a partir dos conectivos lógicos $\wedge$, $\vee$ e $\neg$ (bem como outros discutidos adiante). Tal expressão é chamada de *proposição*.

A principal propriedade de uma proposição $P(p, q, \ldots)$ é que seu valor verdade depende exclusivamente dos valores verdade de suas variáveis, ou seja, o valor verdade de uma proposição é determinado, uma vez que o valor

verdade de cada uma de suas variáveis seja conhecido. Uma maneira concisa e simples para mostrar essa relação é por meio de uma *tabela verdade*. Descrevemos abaixo um modo de obter tal tabela verdade.

Considere, por exemplo, a proposição $\neg(p \wedge \neg q)$. A Fig. 4-2(a) indica como a tabela verdade de $\neg(p \wedge \neg q)$ é construída. Observe que as primeiras colunas da tabela são para as variáveis p, q, . . e que há linhas suficientes para permitir todas as possíveis combinações de V e F para essas *variáveis*. (Para duas variáveis, como acima, quatro linhas são necessárias; para três variáveis, oito linhas são necessárias; e, no caso geral, para n variáveis, 2^n linhas são necessárias.) Existe, assim, uma coluna para cada estágio "elementar" da construção da proposição, sendo que o valor verdade em cada passo é determinado a partir dos estágios anteriores, pela definição dos conectivos $\wedge$, $\vee$ e $\neg$. Finalmente, obtemos o valor verdade da proposição, o qual aparece na última coluna.

A tabela verdade finalizada da proposição $\neg(p \wedge \neg q)$ é mostrada na Fig. 4-2(b). Ela consiste precisamente nas colunas da Fig. 4-2(a) que aparecem sob as variáveis e sob a proposição; as outras colunas foram meramente usadas na construção da tabela verdade.

p	q	$\neg q$	$p \wedge \neg q$	$\neg(p \wedge \neg q)$
V	V	F	F	V
V	F	V	V	F
F	V	F	F	V
F	F	V	F	V

(a)

p	q	$\neg(p \wedge \neg q)$
V	V	V
V	F	F
F	V	V
F	F	V

(b)

Figura 4-2

Observação: Para evitar um número excessivo de parênteses, às vezes adotamos uma ordem de precedência para os conectivos lógicos. Especificamente,

$\neg$ precede $\wedge$, o qual tem precedência sobre $\vee$

Por exemplo, $\neg\, p \wedge q$ significa $(\neg\, p) \wedge q$ e não $\neg(p \wedge q)$.

Método alternativo para construir uma tabela verdade

Outra maneira para construir a tabela verdade para $\neg(p \wedge \neg q)$ é a seguinte:

(a) Primeiro, construímos a tabela verdade mostrada na Fig. 4-3. Ou seja, primeiro listamos todas as variáveis e as combinações de seus valores verdade. Há também uma linha final rotulada "passo". Em seguida, a proposição é escrita na linha do topo e à direita de suas variáveis, com espaço suficiente de modo a existir uma coluna sob cada variável e sob cada operação lógica na proposição. Por último, (Passo 1), os valores verdade das variáveis são colocados na tabela sob as variáveis na proposição.

(b) Agora valores verdade adicionais são colocados na tabela verdade, coluna por coluna, sob cada operação lógica, como mostrado na Fig. 4-4. Também indicamos o passo no qual cada coluna de valores verdade é colocado na tabela.

A tabela verdade da proposição, portanto, consiste nas colunas originais sob as variáveis e do último passo, ou seja, a última coluna é colocada dentro da tabela.

p	q	$\neg$	$(p$	$\wedge$	$\neg$	$q)$
V	V		V			V
V	F		V			F
F	V		F			V
F	F		F			F
Passo						

Figura 4-3

p	q	$\neg$	$(p$	$\wedge$	$\neg$	$q)$
V	V		V		F	V
V	F		V		V	F
F	V		F		F	V
F	F		F		V	F
Passo			1		2	1

(*a*)

p	q	$\neg$	$(p$	$\wedge$	$\neg$	$q)$
V	V		V	F	F	V
V	F		V	V	V	F
F	V		F	F	F	V
F	F		F	F	V	F
Passo			1	3	2	1

(*b*)

p	q	$\neg$	$(p$	$\wedge$	$\neg$	$q)$
V	V	V	V	F	F	V
F	V	F	V	V	V	F
F	F	V	F	F	F	V
F	F	V	F	F	V	F
Passo	4	1	3	2	1	

(*c*)

Figura 4-4

4.5 TAUTOLOGIAS E CONTRADIÇÕES

Algumas proposições $P(p, q, \ldots)$ contêm apenas V na última coluna de suas tabelas verdade ou, em outras palavras, são verdadeiras para quaisquer valores verdade de suas variáveis. Tais proposições são chamadas de *tautologias*. Analogamente, uma proposição $P(p, q, \ldots)$ é dita uma *contradição* se tiver apenas F na última coluna de sua tabela verdade ou, em outras palavras, se for falsa para quaisquer valores verdade de suas variáveis. Por exemplo, a proposição "p ou não p ", isto é, $p \vee \neg p$, é uma tautologia, e a proposição "p e não p", isto é, $p \wedge \neg p$, é uma contradição. Isso é verificado, examinando suas tabelas verdade na Fig. 4-5. (As tabelas verdade têm somente duas linhas, uma vez que cada proposição conta apenas com uma variável p.)

p	$\neg p$	$p \vee \neg p$
V	F	V
F	V	V

(*a*) $p \vee \neg p$

p	$\neg p$	$p \wedge \neg p$
V	F	F
F	V	F

(*b*) $p \wedge \neg p$

Figura 4-5

Observe que a negação de uma tautologia é uma contradição, pois é sempre falsa. E a negação de uma contradição é uma tautologia, uma vez que é sempre verdadeira.

Agora seja $P(p, q, \ldots)$ uma tautologia, e sejam $P_1(p, q, \ldots)$, $P_2(p, q, \ldots)$, . . . proposições quaisquer. Como $P(p, q, \ldots)$ não depende dos valores verdade em particular de suas variáveis $p, q, \ldots$, podemos substituir P_1 por p, P_2 por q, . . . na tautologia $P(p, q, \ldots)$ e ainda teremos uma tautologia. Em outros termos:

Teorema 4.1 (Princípio de Substituição): Se $P(p, q, \ldots)$ é uma tautologia, então $P(P_1, P_2, \ldots)$ é uma tautologia para quaisquer proposições $P_1, P_2, \ldots$.

4.6 EQUIVALÊNCIA LÓGICA

Duas proposições $P(p, q, \ldots)$ e $Q(p, q, \ldots)$ são *logicamente equivalentes* ou, simplesmente, *equivalentes* ou *iguais*, e se escreve

$$P(p, q, \ldots) \equiv Q(p, q, \ldots)$$

se tiverem tabelas verdade idênticas. Considere, por exemplo, as tabelas verdade de $\neg(p \wedge q)$ e $\neg p \vee \neg q$ que aparecem na Fig. 4-6. Observe que ambas as tabelas verdade são a mesma, isto é, as duas proposições são falsas no primeiro caso e verdadeiras nos outros três casos. Consequentemente, podemos escrever

$$\neg(p \wedge q) \equiv \neg p \vee \neg q$$

Em outras palavras, as proposições são logicamente equivalentes.

Observação: Sejam p a sentença "Rosas são vermelhas" e q a sentença "Violetas são azuis". Seja S a declaração:

"Não é verdade que rosas são vermelhas e violetas são azuis."

Então S pode ser escrita na forma $\neg(p \wedge q)$. Contudo, como observado acima, $\neg(p \wedge q) \equiv \neg p \vee \neg q$. Consequentemente, S tem o mesmo significado da declaração:

"Rosas não são vermelhas ou violetas não são azuis."

p	q	$p \wedge q$	$\neg (p \wedge q)$
V	V	V	F
V	F	F	V
F	V	F	V
F	F	F	V

(*a*) $\neg (p \wedge q)$

p	q	$\neg p$	$\neg q$	$\neg p \vee \neg q$
V	V	F	F	F
V	F	F	V	V
F	V	V	F	V
F	F	V	V	V

(*b*) $\neg p \vee \neg q$

Figura 4-6

4.7 ÁLGEBRA DE PROPOSIÇÕES

Proposições satisfazem várias leis que são listadas na Tabela 4-1. (Nessa tabela, V e F são restritos aos valores verdade "Verdadeiro" e "Falso", respectivamente.) Estabelecemos esse resultado formalmente.

Teorema 4.2: Proposições satisfazem as leis da Tabela 4-1.

(Observe a semelhança entre a Tabela 4-1 e a Tabela 1-1 sobre conjuntos.)

Tabela 4-1 Leis da álgebra de proposições

Idempotência	(1a) $p \vee p \equiv p$	(1b) $p \wedge p \equiv p$
Associatividade	(2a) $(p \vee q) \vee r \equiv p \vee (q \vee r)$	(2b) $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
Comutatividade	(3a) $p \vee q \equiv q \vee p$	(3b) $p \wedge q \equiv q \wedge p$
Distributividade	(4a) $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$	(4b) $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
Identidade	(5a) $p \vee F \equiv p$ (6a) $p \vee V \equiv V$	(5b) $p \wedge V \equiv p$ (6b) $p \wedge F \equiv F$
Involução	(7) $\neg\neg p \equiv p$	
Complementaridade	(8a) $p \vee \neg p \equiv V$ (9a) $\neg V \equiv F$	(8b) $p \wedge \neg p \equiv V$ (9b) $\neg F \equiv V$
Leis de DeMorgan	(10a) $\neg(p \vee q) \equiv \neg p \wedge \neg q$	(10b) $\neg(p \wedge q) \equiv \neg p \vee \neg q$

4.8 SENTENÇAS CONDICIONAIS E BICONDICIONAIS

Muitas sentenças, especialmente em matemática, são da forma "Se p então q". Tais sentenças são chamadas de *condicionais* e são denotadas por

$$p \rightarrow q$$

A condicional $p \rightarrow q$ é frequentemente lida como "p implica q" ou "p somente se q".

Outra sentença comum é da forma "p se, e somente se, q". Tais sentenças são conhecidas como *bicondicionais* e são denotadas por

$$p \leftrightarrow q$$

Os valores verdade de $p \rightarrow q$ e $p \leftrightarrow q$ são definidos pelas tabelas na Fig. 4-7(*a*) e (*b*). Note que:

(a) A condicional $p \rightarrow q$ é falsa apenas quando a primeira parte p é verdadeira e a segunda parte q é falsa. Logo, quando p é falsa, a condicional $p \rightarrow q$ é verdadeira, independentemente do valor verdade de q.

(b) A bicondicional $p \leftrightarrow q$ é verdadeira sempre que p e q têm os mesmos valores verdade e falsa nos demais casos.

A tabela verdade de $\neg p \wedge q$ aparece na Fig. 4-7(c). Observe que a tabela verdade de $\neg p \vee q$ e $p \rightarrow q$ são idênticas, ou seja, são ambas falsas apenas no segundo caso. Consequentemente, $p \rightarrow q$ é logicamente equivalente a $\neg p \vee q$; ou seja,

$$p \rightarrow q \equiv \neg p \vee q$$

Em outras palavras, a sentença condicional "Se p então q" é logicamente equivalente à sentença "Não p ou q", a qual envolve apenas os conectivos $\vee$ e $\neg$ e, assim, já faz parte de nossa linguagem. Podemos considerar $p \rightarrow q$ como uma abreviação para uma sentença recorrente.

p	q	$p \rightarrow q$
V	V	V
V	F	F
F	V	V
F	F	V

(a) $p \rightarrow q$

p	q	$p \leftrightarrow q$
V	V	V
V	F	F
F	V	F
F	F	V

(b) $p \leftrightarrow q$

p	q	$\neg p$	$\neg p \vee q$
V	V	F	V
V	F	F	F
F	V	V	V
F	F	V	V

(c) $\neg p \vee q$

Figura 4-7

4.9 ARGUMENTOS

Um *argumento* (ou *inferência*) é uma afirmação na qual um dado conjunto de proposições $P_1, P_2, \ldots, P_n$, chamadas de *premissas*, implica (tem como consequência) outra proposição Q, chamada de *conclusão*. Tal argumento é denotado por

$$P_1, P_2, \ldots, P_n \vdash Q$$

A noção de "argumento lógico" ou "argumento válido" é formalizada como se segue:

Definição 4.4: Um argumento $P_1, P_2, \ldots, P_n \vdash Q$ é dito *válido* se Q é verdadeira se todas as premissas $P_1, P_2, \ldots, P_n$ são verdadeiras.

Um argumento que não é válido é chamado de *falácia*.

Exemplo 4.4

(a) O seguinte argumento é válido:

$$p, p \rightarrow q \vdash q \text{ (}\textit{Modus Ponens}\text{)}$$

A demonstração dessa regra segue da tabela verdade da Fig. 4-7(a). Especificamente, p e $p \rightarrow q$ são simultaneamente verdadeiras apenas no Caso (linha) 1 e, nesse caso, q é verdadeira.

(b) O argumento a seguir é uma falácia:

$$p \rightarrow q, q \vdash p$$

$p \rightarrow q$ e q são ambas verdadeiras no Caso (linha) 3 da tabela verdade da Fig. 4-7(a), mas, nesse caso, p é falsa.

Agora, as proposições $P_1, P_2, \ldots, P_n$ são simultaneamente verdadeiras se, e somente se, a proposição $P_1 \wedge P_2 \wedge \ldots P_n$ é verdadeira. Assim, o argumento $P_1, P_2, \ldots, P_n \vdash Q$ é válido se, e somente se, Q é verdadeira sempre que $P_1 \wedge P_2 \wedge \ldots \wedge P_n$ for verdadeira ou, equivalentemente, se a proposição $(P_1 \wedge P_2 \wedge \ldots \wedge P_n) \rightarrow Q$ for uma tautologia. Estabelecemos esse resultado formalmente.

Teorema 4.3: O argumento $P_1, P_2, \ldots, P_n \vdash Q$ é válido se, e somente se, a proposição $(P_1 \wedge P_2 \ldots \wedge P_n) \rightarrow Q$ é uma tautologia.

Aplicamos esse teorema no exemplo a seguir.

Exemplo 4.5 Um princípio fundamental de raciocínio lógico diz:

"Se p implica q e q implica r, então p implica r"

p	q	r	$[(p$	$\rightarrow$	$q)$	$\wedge$	$(q$	$\rightarrow$	$r)]$	$\rightarrow$	$(p$	$\rightarrow$	$r)$
V	V	V	V	V	V	V	V	V	V	V	V	V	V
V	V	F	V	V	V	F	V	F	F	V	V	F	F
V	F	V	V	F	F	F	F	V	V	V	V	V	V
V	F	F	V	F	F	F	F	V	F	V	V	F	F
F	V	V	F	V	V	V	V	V	V	V	F	V	V
F	V	F	F	V	V	F	V	F	F	V	F	V	F
F	F	V	F	V	F	V	F	V	V	V	F	V	V
F	F	F	F	V	F	V	F	V	F	V	F	V	F
Passo			1	2	1	3	1	2	1	4	1	2	1

Figura 4-8

Ou seja, o argumento a seguir é válido:

$$p \rightarrow q,\ q \rightarrow r \vdash p \rightarrow r \text{ (\textit{Lei do Silogismo})}$$

Esse fato é verificado pela tabela verdade na Fig. 4-8, a qual mostra que a seguinte proposição é uma tautologia:

$$[(p \rightarrow q) \wedge (q \rightarrow r)] \rightarrow (p \rightarrow r)$$

De forma equivalente, o argumento é válido, uma vez que as premissas $p \rightarrow q$ e $q \rightarrow r$ são simultaneamente verdadeiras apenas nos Casos (linhas) 1, 5, 7 e 8 e, nesses casos, a conclusão também é verdadeira. (Observe que a tabela verdade demandou $2^3 = 8$ linhas, pois há três variáveis p, q e r.)

Agora aplicamos a teoria acima sobre argumentos envolvendo sentenças específicas. Enfatizamos que a validade de um argumento não depende dos valores verdade nem do conteúdo das sentenças que surgem no argumento, mas da forma particular da inferência. Isso é ilustrado no exemplo a seguir.

Exemplo 4.6 Considere o seguinte argumento:

S_1: Se um homem é solteiro, ele é infeliz.
S_2: Se um homem é infeliz, ele morre cedo.

S: Solteiros morrem cedo.

Aqui a sentença S abaixo da linha denota a conclusão do argumento, e as sentenças S_1 e S_2 acima da linha correspondem às premissas. Afirmamos que o argumento $S_1, S_2, \vdash S$ é válido, pois ele é da forma

$$p \rightarrow q,\ q \rightarrow r \vdash p \rightarrow r$$

onde p é "Ele é solteiro", q é "Ele é infeliz" e r é "Ele morre cedo"; e pelo Exemplo 4.5 essa inferência (Lei do Silogismo) é válida.

4.10 FUNÇÕES PROPOSICIONAIS, QUANTIFICADORES

Seja A um dado conjunto. Uma *função proposicional* (ou *sentença aberta* ou *condição*) definida sobre A é uma expressão

$$p(x)$$

que tem a propriedade de que $p(a)$ é verdadeira ou falsa para cada $a \in A$. Ou seja, $p(x)$ se torna uma sentença (com um valor verdade) sempre que a variável x é substituída por qualquer elemento $a \in A$. O conjunto A é chamado de

domínio de $p(x)$, e o conjunto T_p de todos os elementos de A para os quais $p(a)$ é verdadeira é chamado de *conjunto verdade* de $p(x)$. Em outras palavras,

$$T_p = \{x \mid x \in A,\ p(x) \text{ é verdadeira}\} \quad \text{ou} \quad T_p = \{x \mid p(x)\}$$

Frequentemente, quando A é algum conjunto de números, a condição $p(x)$ tem a forma de uma equação ou desigualdade envolvendo a variável x.

Exemplo 4.7 Encontre o conjunto verdade para cada função proposicional $p(x)$ definida sobre o conjunto **N** dos inteiros positivos.

(a) Seja $p(x)$ a fórmula "$x + 2 > 7$". Seu conjunto verdade é $\{6, 7, 8, \ldots\}$, consistindo de todos os inteiros maiores do que 5.

(b) Seja $p(x)$ a fórmula "$x + 5 < 3$". Seu conjunto verdade é o conjunto vazio $\varnothing$. Isto é, $p(x)$ não é verdade para qualquer inteiro em **N**.

(c) Seja $p(x)$ a fórmula "$x + 5 > 1$". Seu conjunto verdade é **N**. Ou seja, $p(x)$ é verdadeira para todo elemento de **N**.

Observação: O exemplo acima mostra que se $p(x)$ é uma função proposicional definida sobre um conjunto A, então $p(x)$ poderia ser verdadeira para todo $x \in A$, para algum (ou alguns) $x \in A$, ou para nenhum $x \in A$. As duas subseções a seguir discutem quantificadores relacionados com tais funções proposicionais.

Quantificador universal

Seja $p(x)$ uma função proposicional definida sobre um conjunto A. Considere a expressão

$$(\forall x \in A)p(x) \quad \text{ou} \quad \forall x\, p(x) \tag{4.1}$$

a qual se lê "Para todo x em A, $p(x)$ é uma sentença verdadeira" ou, simplesmente, "Para todo x, $p(x)$". O símbolo

$$\forall$$

que se lê "para todo" é chamado de *quantificador universal*. A sentença (*4.1*) é equivalente à afirmação

$$T_p = \{x \mid x \in A, p(x)\} = A \tag{4.2}$$

ou seja, à afirmação de que o conjunto verdade de $p(x)$ é o conjunto todo A.

A expressão $p(x)$ em si é uma sentença aberta ou condição e, portanto, não tem valor verdade. Contudo, $\forall x\, p(x)$, isto é, $p(x)$ precedida pelo quantificador $\forall$, tem um valor verdade que segue da equivalência entre (*4.1*) e (*4.2*). Especificamente:

Q_1: Se $\{x \mid x \in A, p(x)\} = A$, então $\forall x\, p(x)$ é verdadeira; caso contrário, $\forall x\, p(x)$ é falsa.

Exemplo 4.8

(a) A proposição $(\forall n \in \mathbf{N})(n + 4 > 3)$ é verdadeira, pois $\{n \mid n + 4 > 3\} = \{1, 2, 3, \ldots\} = \mathbf{N}$.

(b) A proposição $(\forall n \in \mathbf{N})(n + 2 > 8)$ é falsa, pois $\{n \mid n + 2 > 8\} = \{7, 8, \ldots\} \neq \mathbf{N}$.

(c) O símbolo $\forall$ pode ser usado para definir a interseção de uma coleção indexada $\{A_i \mid i \in I\}$ de conjuntos A_i como se segue:

$$\cap(A_i \mid i \in I) = \{x \mid \forall_i \in I, x \in A_i\}$$

Quantificador existencial

Seja $p(x)$ uma função proposicional definida sobre um conjunto A. Considere a expressão

$$(\exists x \in A)p(x) \quad \text{ou} \quad \exists x, p(x) \tag{4.3}$$

que se lê "Existe um x em A tal que $p(x)$ é uma sentença verdadeira" ou, simplesmente, "Para algum x, $p(x)$". O símbolo

$$\exists$$

o qual se lê "existe" ou "para algum" ou "para pelo menos um" é chamado de *quantificador existencial*. A sentença (*4.3*) é equivalente à sentença

$$T_p = \{x \mid x \in A, p(x)\} \neq \varnothing \tag{4.4}$$

isto é, à afirmação de que o conjunto verdade de $p(x)$ não é vazio. Consequentemente, $\exists x\, p(x)$, ou seja, $p(x)$ precedida pelo quantificador $\exists$, tem um valor verdade. Especificamente:

Q_2: Se $\{x \mid p(x)\} \neq \varnothing$, então $\exists x\, p(x)$ é verdadeira; caso contrário, $\exists x\, p(x)$ é falsa.

Exemplo 4.9

(a) A proposição $(\exists n \in N)(n + 4 < 7)$ é verdadeira, pois $\{n \mid n + 4 < 7\} = \{1, 2\} \neq \varnothing$.

(b) A proposição $(\exists n \in N)(n + 6 < 4)$ é falsa, uma vez que $\{n \mid n + 6 < 4\} = \varnothing$.

(c) O símbolo $\exists$ pode ser usado para definir a união de uma coleção indexada $\{A_i \mid i \in I\}$ de conjuntos A_i, como se segue:

$$\cup(A_i \mid i \in I) = \{x \mid \exists i \in I, x \mid \in A_i\}$$

4.11 NEGAÇÃO DE SENTENÇAS QUANTIFICADAS

Considere a sentença: "Todos os bacharéis em matemática são homens". Sua negação se lê:

> "Não é o caso de que todos os bacharéis em matemática são homens" ou, equivalentemente, "Existe pelo menos um bacharel em matemática que é mulher (não é homem)"

Simbolicamente, usando M para denotar o conjunto de bacharéis de matemática, a afirmação acima pode ser escrita como

$$\neg(\forall x \in M)(x \text{ é homem}) \equiv (\exists\, x \in M)\ (x \text{ não é homem})$$

ou, quando $p(x)$ denota "x é homem",

$$\neg(\forall x \in M)\ p(x) \equiv (\exists\, x \in M)\ \neg p(x) \quad \text{ou} \quad \neg\forall x p(x) \equiv \exists x \neg p(x)$$

Isso é verdade para qualquer proposição $p(x)$. Ou seja:

Teorema 4.4 (DeMorgan): $\neg(\forall x \in A)p(x) \equiv (\exists\, x \in A)\neg p(x)$.

Em outras palavras, as duas sentenças a seguir são equivalentes:

(1) Não é verdade que para todo $a \in A$, $p(a)$ é verdadeira. (2) Existe um $a \in A$ tal que $p(a)$ é falsa.

Há um teorema análogo para a negação de uma proposição que contém o quantificador existencial.

Teorema 4.5 (DeMorgan): $\neg(\exists x \in A)p(x) \equiv (\forall x \in A)\neg p(x)$

Ou seja, as duas sentenças a seguir são equivalentes:

(1) Não é verdade que para algum $a \in A$, $p(a)$ é verdadeira. (2) Para todo $a \in A$, $p(a)$ é falsa.

Exemplo 4.10

(a) As sentenças a seguir são negações uma da outra:

"Para todos os inteiros positivos n temos $n + 2 > 8$"
"Existe um inteiro positivo n tal que $n + 2 \not> 8$"

(b) As sentenças a seguir também são negações uma da outra:

"Existe uma pessoa (viva) com 150 anos de idade"
"Toda pessoa viva não tem 150 anos de idade"

Observação: A expressão $\neg p(x)$ tem o significado óbvio:

"A sentença $\neg p(a)$ é verdadeira quando $p(a)$ é falsa, e vice-versa"

Anteriormente, $\neg$ foi usado como uma operação sobre sentenças; aqui $\neg$ é usado como uma operação sobre funções proposicionais. Analogamente, $p(x) \wedge q(x)$, que se lê "$p(x)$ e $q(x)$", é definida por:

"A sentença $p(a) \wedge q(a)$ é verdadeira quando $p(a)$ e $q(a)$ são verdadeiras"

Analogamente, $p(x) \vee q(x)$, que se lê "$p(x)$ e $q(x)$", é definida por:

"A sentença $p(a) \vee q(a)$ é verdadeira quando $p(a)$ ou $q(a)$ é verdadeira"

Assim, em termos de conjuntos verdade:

(i) $\neg p(x)$ é o complemento de $p(x)$.
(ii) $p(x) \wedge q(x)$ é a interseção entre $p(x)$ e $q(x)$.
(iii) $p(x) \vee q(x)$ é a união de $p(x)$ com $q(x)$.

Podemos também mostrar que as leis para proposições valem igualmente para funções proposicionais. Por exemplo, temos as Leis de DeMorgan:

$$\neg(p(x) \wedge q(x)) \equiv \neg p(x) \vee \neg q(x) \quad \text{e} \quad \neg(p(x) \vee q(x)) \equiv \neg p(x) \wedge \neg q(x)$$

Contraexemplo

O Teorema 4.6 nos diz que para mostrar que uma sentença $\forall x$, $p(x)$ é falsa, é equivalente mostrar que $\exists\, x \neg p(x)$ é verdadeira ou, em outros termos, que existe um elemento x_0 com a propriedade de que $p(x_0)$ é falsa. Tal elemento x_0 é chamado de *contraexemplo* da afirmação $\forall x$, $p(x)$.

Exemplo 4.11

(a) Considere a sentença $\forall x \in \mathbf{R}$, $|x| \neq 0$. Ela é falsa, pois 0 é um contraexemplo, ou seja, $|0| \neq 0$ não é verdadeira.
(b) Considere a sentença $\forall x \in \mathbf{R}$, $x^2 \geq x$. Ela não é verdadeira, uma vez que, por exemplo, $\frac{1}{2}$ é um contraexemplo. Especificamente, $(\frac{1}{2})^2 \geq \frac{1}{2}$ não é verdadeira, isto é, $(\frac{1}{2})^2 < \frac{1}{2}$.
(c) Considere a sentença $\forall x \in \mathbf{N}$, $x^2 \geq x$. Ela é verdadeira, onde $\mathbf{N}$ é o conjunto de inteiros positivos. Em outras palavras, não existe inteiro positivo n para o qual $n^2 < n$.

Funções proposicionais com mais de uma variável

Uma função proposicional (de n variáveis) definida sobre um produto cartesiano $A = A_1 \times \cdots \times A_n$ é uma expressão

$$p(x_1, x_2, \ldots, x_n)$$

que tem a propriedade de que $p(a_1, a_2, \ldots, a_n)$ é verdadeira ou falsa para qualquer n-upla $p(a_1, \ldots, a_n)$ de A. Por exemplo,

$$x + 2y + 3z < 18$$

é uma função proposicional sobre $\mathbf{N}^3 = \mathbf{N} \times \mathbf{N} \times \mathbf{N}$. Tal função proposicional não tem valor verdade. Contudo, temos o que se segue:

Princípio básico: Uma função proposicional precedida por um quantificador para cada variável, por exemplo,

$$\forall x \exists y, p(x, y) \quad \text{ou} \quad \exists x\, \forall y\, \exists z, p(x, y, z)$$

denota uma sentença e tem um valor verdade.

Exemplo 4.12 Seja $B = \{1, 2, 3, \ldots, 9\}$ e considere que $p(x, y)$ denota "$x + y = 10$". Então $p(x, y)$ é uma função proposicional sobre $A = B^2 = B \times B$.

(a) O que se segue é uma sentença, uma vez que há um quantificador para cada variável:

$$\forall x \exists y, p(x, y), \qquad \text{isto é,} \qquad \text{"Para todo } x\text{, existe } y \text{ tal que } x + y = 10\text{"}$$

Essa sentença é verdadeira. Por exemplo, se $x = 1$, faça $y = 9$; se $x = 2$, faça $y = 8$; e assim por diante.

(b) O que se segue também é uma sentença:

$$\exists y \forall x, p(x, y), \qquad \text{isto é,} \qquad \text{"Existe } y \text{ tal que, para todo } x\text{, temos } x + y = 10\text{"}$$

Não existe tal y; logo, essa sentença é falsa.

Observe que a única diferença entre (a) e (b) é a ordem dos quantificadores. Assim, uma ordenação diferente dos quantificadores pode produzir uma sentença diferente. Notamos que, quando traduzimos tais sentenças quantificadas para o português, a expressão "tal que" frequentemente é seguida por "existe".

Negando sentenças quantificadas com mais de uma variável

Sentenças quantificadas com mais de uma variável podem ser negadas aplicando sucessivamente os Teoremas 4.5 e 4.6. Assim, cada $\forall$ é mudado para $\exists$ e cada $\exists$ é mudado para $\forall$, à medida que o símbolo de negação $\neg$ passa pela sentença, da esquerda para a direita. Por exemplo,

$$\begin{aligned} \neg[\forall x \exists y \exists z, p(x, y, z)] &\equiv \exists x \neg[\exists y \exists z, p(x, y, z)] \equiv \neg \exists z \forall y [\exists z, p(x, y, z) \\ &\equiv \exists x \forall y \forall z, \neg p(x, y, z) \end{aligned}$$

Naturalmente, não colocamos todos os passos quando negamos tais sentenças quantificadas.

Exemplo 4.13

(a) Considere a sentença quantificada:

"Todo estudante cursa pelo menos uma disciplina na qual o docente é um professor assistente."

Sua negação é a sentença:

"Existe um estudante tal que, em toda disciplina cursada, o docente não é um professor assistente."

(b) A definição formal de que L é o limite de uma sequência $a_1, a_2, \ldots$ segue abaixo:

$$\forall \in > 0, \exists\, n_0 \in \mathbf{N}, \forall_n > n_0, \text{temos } | a_n - L| < \in$$

Assim, L não é o limite da sequência $a_1, a_2, \ldots$ quando:

$$\exists \in > 0, \forall n_0 \in \mathbf{N}, \exists_n > n_0 \text{ tal que } | a_n - L| \geq \in$$

Problemas Resolvidos

Proposições e tabelas verdade

4.1 Seja p a sentença "Está frio" e seja q "Está chovendo". Dê uma sentença verbal que descreva cada uma das sentenças a seguir: (*a*) $\neg p$; (*b*) $p \wedge q$; (*c*) $p \vee q$; (*d*) $q \vee \neg p$.

Em cada caso, traduza $\wedge$, $\vee$ e $\sim$ como "e", "ou" e "É falso que" ou "não", respectivamente, e então simplifique a sentença em português.

(*a*) Não está frio.

(*b*) Está frio e chovendo.

(*c*) Está frio ou está chovendo.

(*d*) Está chovendo ou não está frio.

4.2 Encontre a tabela verdade de $\neg p \wedge q$.

Construa a tabela verdade de $\neg p \wedge q$ como na Fig. 4-9(*a*).

p	q	$\neg p$	$\neg p \wedge q$
V	V	F	F
V	F	F	F
F	V	V	V
F	F	V	F

(*a*) $\neg p \wedge q$

p	q	$p \wedge q$	$\neg (p \wedge q)$	$p \vee \neg (p \wedge q)$
V	V	V	F	V
V	F	F	V	V
F	V	F	V	V
F	F	F	V	V

(*b*) $p \vee \neg (p \wedge q)$

Figura 4-9

4.3 Verifique que a proposição $p \vee \neg(p \wedge q)$ é uma tautologia.

Construa a tabela verdade de $p \vee \neg(p \wedge q)$ como mostrado na Fig. 4-9(*b*). Como o valor verdade de $p \vee \neg(p \wedge q)$ é V para todos os valores de p e q, a proposição é uma tautologia.

4.4 Mostre que as proposições $\neg(p \wedge q)$ e $\neg p \vee \neg q$ são logicamente equivalentes.

Construa as tabelas verdade para $\neg(p \wedge q)$ e $\neg p \vee \neg q$, como na Fig. 4-10. Como as tabelas verdade são as mesmas (ambas as proposições são falsas no primeiro caso e verdadeiras nos outros três casos), as proposições $\neg(p \wedge q)$ e $\neg p \vee \neg q$ são logicamente equivalentes e podemos escrever

$$\neg(p \wedge q) \equiv \neg p \vee \neg q$$

p	q	$p \wedge q$	$\neg (p \wedge q)$
V	V	V	F
V	F	F	V
F	V	F	V
F	F	F	V

(*a*) $\neg (p \wedge q)$

p	q	$\neg p$	$\neg q$	$\neg p \vee \neg q$
V	V	F	F	F
V	F	F	V	V
F	V	V	F	V
F	F	V	V	V

(*b*) $\neg p \vee \neg q$

Figura 4-10

Parte 4

Expressões e Atribuições (Operadores Aritméticos, Lógicos e Relacionais, Precedências de Operadores)

```
Constantes: <identificador> = <valor>
```

Exemplo de declaração de constantes:

```
Algoritmo NoIntervalo
{INFORMAR SE UM VALOR LIDO ESTÁ ENTRE DOIS LIMITES}
  Constantes:
    LIMITESUP = 10
    LIMITEINF = 3
```

Constantes são bastante utilizadas quando um mesmo valor aparece em diferentes pontos de um programa, como no exemplo anterior, em que se trabalha com um determinado intervalo de valores que será testado diversas vezes. Sem o uso de constantes, seria necessário que os mesmos valores fossem digitados em diferentes pontos do programa. Um erro de digitação do valor de um dos limites não impediria a compilação correta do programa, aumentando a possibilidade de erro de execução. Adicionalmente, caso fosse necessário alterar esse intervalo (por exemplo, aumentar o valor do limite superior), seria necessário que a alteração fosse feita em diferentes pontos do programa, o que poderia levar a erros. Com a utilização de uma constante, somente o valor definido na declaração precisa ser alterado sem que outra modificação no restante do programa se faça necessária.

Para identificar facilmente os lugares onde uma constante é utilizada, aconselha-se escrever seus nomes utilizando apenas letras maiúsculas.

2.3 → expressões

No exemplo simulado da Seção 1.1.1 foram realizadas algumas operações que envolvem cálculos de expressões aritméticas. Ao escrever o programa correspondente àquele exemplo, essas expressões deverão ser escritas de forma que sejam entendidas corretamente pelo compilador. Cada linguagem de programação define regras bem claras para escrever expressões aritméticas, lógicas e de *strings*.

2.3.1 expressões aritméticas

Expressões aritméticas são expressões cujos resultados são valores numéricos, inteiros ou fracionários. A sintaxe de uma expressão aritmética é a seguinte:

```
<operando>  <operador aritmético>  <operando>
```

Na pseudolinguagem utilizada neste livro, os operadores que podem ser usados em expressões aritméticas são os mesmos utilizados nas expressões aritméticas comuns. Mas, da mesma forma que nas linguagens de programação, o símbolo utilizado para a multiplicação é o asterisco, e o símbolo de divisão é a barra inclinada. A Tabela 2.1 mostra os operadores que podem ser utilizados em expressões aritméticas, na forma adotada pela pseudolinguagem.

tabela 2.1 Operadores aritméticos na pseudolinguagem

Operador	Significado	Observação
+	Soma	-
-	Subtração	-
*	Multiplicação	-
/	Divisão	-
**	Potência	-
Div	Divisão inteira	Operandos inteiros
Mod	Resto da divisão inteira	Operandos inteiros

Os operadores aritméticos têm diferentes precedências na execução das operações: primeiro são calculadas as potências, depois as multiplicações e as divisões e, no final, as somas e as subtrações. Expressões com operadores de mesma precedência justapostos são avaliadas da esquerda para a direita. Essa ordem de precedência pode ser alterada através do uso de parênteses.

Os seguintes tipos de operandos podem ser utilizados:

1. valores numéricos literais;
2. variáveis numéricas;
3. chamadas a funções[1] que devolvem um valor numérico;
4. expressões aritméticas, as quais podem incluir partes entre parênteses.

Se uma expressão aritmética incluir funções, essas terão precedência maior na execução.

Exemplos de expressões aritméticas:

```
a + 1
a * 2 + 7,32
( x / 2 )  / C - ( valor + 1 / 2 )
2 + cos(x)                              onde cos(x) é uma função
```

As expressões aritméticas devem ser escritas horizontalmente, em uma mesma linha, com eventuais valores fracionários expressos linearmente. Muitas vezes é necessário o emprego de parênteses para garantir a execução na ordem correta. A necessidade de linearização possibilita a uma expressão aritmética ter sua aparência inicial bastante modificada, como no caso da expressão a seguir:

$$a + \frac{(b-4)(\frac{a}{2}+z42)}{c+d}$$

A representação dessa expressão em pseudolinguagem fica:

```
a + ( ( b - 4 ) * ( a / 2  + 4 * z42) / ( c + d ) )
```

[1] Uma FUNÇÃO é um subprograma. Pode receber parâmetros (valores) para realizar sua tarefa e normalmente devolve um valor em seu nome, sendo o tipo do valor devolvido o próprio tipo da função. Mais detalhes sobre definição de funções são vistos no Capítulo 9.

Algumas funções básicas predefinidas já vêm embutidas nas linguagens de programação. Entre elas, funções matemáticas, como o cálculo do cosseno de um ângulo utilizado no exemplo anterior. Algumas dessas funções necessitam de alguma informação para calcular o que é pedido como, por exemplo, o valor do ângulo do qual se quer o cosseno. As informações requeridas são chamadas de parâmetros da função e são listadas logo após o nome da função, entre parênteses. Um parâmetro pode ser fornecido através de uma expressão cujo valor, depois de avaliado, será utilizado pela função.

Na Tabela 2.2 são listadas algumas funções que podem ser utilizadas na pseudolinguagem, definidas de forma idêntica ou similar àquela em que ocorrem na maioria das linguagens de programação.

tabela 2.2 Funções predefinidas na pseudolinguagem

Nome da função	Parâmetro	Significado
abs	valor	Valor absoluto do valor
sen	ângulo	Seno do ângulo
cos	ângulo	Cosseno do ângulo
tan	ângulo	Tangente do ângulo
arctan	valor	Arco cuja tangente tem o valor
sqrt	valor	Raiz quadrada do valor
sqr	valor	Quadrado do valor
pot	base, expoente	Base elevada ao expoente
ln	valor	Logaritmo neperiano
log	valor	Logaritmo na base 10

2.3.2 expressões lógicas

Expressões lógicas são aquelas que têm como resultado valores lógicos, ou seja, um dos dois valores `verdadeiro` ou `falso`.

Uma expressão lógica pode ter uma das seguintes formas:

```
<relação lógica>
<operando>  <operador lógico binário>  <operando>
<operador lógico unário>  <expressão lógica>
```

Uma relação lógica compara dois valores, numéricos ou alfanuméricos, resultando em um valor lógico `verdadeiro` ou `falso`. A sintaxe de uma relação lógica é a seguinte:

```
<expressão>  <operador relacional>  <expressão>
```

Os operadores relacionais utilizados na pseudolinguagem são listados na Tabela 2.3. Outros operadores serão vistos quando se analisar operações sobre *strings*. É importante lembrar que os dois operandos de uma relação devem ser do mesmo tipo para que possam ser comparados.

tabela 2.3 Operadores relacionais na pseudolinguagem

Operador relacional	Significado
=	Igual
≠	Diferente
>	Maior
<	Menor
≥	Maior ou igual
≤	Menor ou igual

Exemplos de relações:

`idade > 21`	onde `idade` é uma variável numérica
`nome = 'Ana Terra'`	onde `nome` é uma variável *string*
`a < (b + 2 * x)`	onde `a, b e x` são variáveis numéricas

Os operandos de expressões lógicas devem resultar em valores lógicos, que são então comparados através de um operador lógico. Podem ser:

- os valores lógicos literais `verdadeiro` e `falso`;
- variáveis declaradas como lógicas;
- relações lógicas;
- chamadas a funções que tenham resultado lógico;
- outras expressões lógicas.

O uso de parênteses é permitido, tanto para dar prioridade a algumas comparações, como simplesmente para tornar o entendimento das expressões mais claro.

Os operadores lógicos comparam valores lógicos, resultando em `verdadeiro` ou `falso`. Na Tabela 2.4 estão os operadores lógicos usualmente empregados – `e`, `ou`, `oux` (ou exclusivo) e `não` (negação) – identificando como é obtido o resultado da comparação. A Tabela 2.5 apresenta os resultados produzidos por cada operador lógico de acordo com os resultados das expressões lógicas A e B, representando "V" o valor lógico verdadeiro e "F" o falso.

tabela 2.4 Operadores lógicos na pseudolinguagem

Operador lógico	Tipo	Resultado
e	Binário	Verdadeiro somente se ambos os operandos são verdadeiros
ou	Binário	Verdadeiro se um dos operandos for verdadeiro
oux	Binário	Verdadeiro se somente um dos operandos for verdadeiro
não	Unário	Verdadeiro se o operando for falso, falso se o operando for verdadeiro

tabela 2.5 Tabela-verdade dos operadores lógicos

A	B	A e B	A ou B	A oux B	não A
V	V	V	V	F	F
V	F	F	V	V	F
F	V	F	V	V	V
F	F	F	F	F	V

A ordem de precedência na avaliação das operações incluídas em uma expressão lógica pode variar conforme a linguagem de programação utilizada. Na pseudolinguagem é adotada a seguinte ordem de precedência na avaliação das operações: primeiro são avaliadas as expressões aritméticas, depois as relações e, por último, as expressões lógicas. Nas expressões lógicas, primeiro são realizadas as negações e, depois, são aplicados os operadores lógicos, entre os quais o "e" tem maior prioridade, seguido pelos operadores "ou" e "oux". As ordens de precedência utilizadas nas linguagens Pascal e C serão mostradas nas seções específicas para essas linguagens, mais adiante neste capítulo.

Independentemente do conhecimento da ordem de precedência adotada na linguagem, o uso de parênteses é recomendado não só porque garante a correta avaliação das expressões, mas também porque facilita o entendimento do que está sendo executado.

Supondo:

a `i` uma variável inteira
b `r` uma variável real
c `c` uma variável do tipo caractere
d `achou` uma variável lógica

as expressões lógicas a seguir são válidas na pseudolinguagem:

```
( i ≠ 10 ) ou achou
( i mod 2 = 7 ) e ( r / 4 < 2 )
( r ≥ 0 ) e ( r + 1 < 10 ) ou achou
c = 'w' oux não achou
```

Parte 5

Comandos Básicos (Entrada, Saída, Atribuição e Declaração)

3.1 esquema básico dos algoritmos sequenciais

Os problemas puramente sequenciais geralmente incluem três atividades, que ocorrem normalmente na ordem indicada a seguir: entrada de dados, processamento realizado sobre esses dados (cálculos, comparações) e saída de dados ou apresentação dos resultados. Mesmo em problemas mais complexos, essas atividades constituem o esquema básico subjacente. O processamento e a saída de dados são elementos sempre presentes; já a entrada pode eventualmente não ocorrer, como nos casos em que o processamento se baseia em valores predefinidos e constantes.

Entradas são os dados fornecidos pelo usuário durante a execução do programa, sem os quais não é possível solucionar o problema. Existem casos em que alguns dados de entrada, pela sua constância e regularidade, podem ser utilizados como constantes em uma solução. No Exercício de Fixação 3.3 (discutido na Seção 3.6), em que um valor em reais é convertido para dólares, em um período de estabilidade da moeda americana a taxa de conversão para o dólar poderia ser colocada como um valor constante na expressão de cálculo. Mas caso fosse preciso alterar essa taxa, seria necessário alterar o código. Nesse, e em casos semelhantes, sugere-se optar pela solução mais genérica, em que mesmo os dados relativamente estáveis são sempre fornecidos como entradas.

As saídas de um problema são geralmente os elementos mais facilmente determináveis, uma vez que correspondem aos resultados esperados.

No problema apresentado no Algoritmo 1.1 (no Capítulo 1, Seção 1.1.4), em que é calculada a soma de dois valores, os dois valores a serem somados são as entradas que devem ser informadas; o processamento é o cálculo da soma; e a saída é a soma calculada.

Na sequência veremos os comandos que permitirão a leitura dos dados de entrada, a produção de resultados e sua apresentação em um meio externo.

3.2 → comandos de entrada e de saída

Os comandos de entrada e de saída de dados fazem a ligação entre o programa e o usuário. Toda a comunicação entre o mundo virtual e o mundo real é feita através desses comandos, sem os quais o usuário não ficaria ciente do que ocorre durante e ao término do processamento.

3.2.1 comando de entrada de dados

Através de um **comando de entrada de dados**, o programador solicita que um ou mais dados sejam obtidos (lidos) pelo computador a partir de um dispositivo de entrada como, por exemplo, o teclado. Os valores lidos devem ser armazenados em variáveis na memória para que essas possam depois ser utilizadas pelo programa. Para isso, o comando de entrada de dados deve, além de solicitar a operação de leitura, informar os nomes das variáveis que irão armazenar os valores lidos.

Na pseudolinguagem, um comando de entrada de dados é identificado pela palavra reservada `ler`, seguida da lista de variáveis que irão armazenar os valores lidos, as quais aparecem separadas por vírgulas e entre parênteses:

```
ler ( <lista de variáveis separadas por vírgulas> )
```

Por exemplo, o comando:

```
ler (nota)
```

pede que seja lido um valor no dispositivo de entrada de dados, dizendo ainda que o valor lido deve ser armazenado na variável denominada `nota`. Outro exemplo é o comando:

```
ler (a, b, c)
```

onde `a`, `b` e `c` são nomes de variáveis. Através desse comando serão lidos três valores de entrada, sendo o primeiro colocado na variável `a`, o segundo na `b` e o terceiro na `c`.

3.2.2 comando de saída de dados

Comandos de saída de dados são usados para transferir para fora do computador os resultados que foram solicitados a fim de que esses sejam vistos pelo usuário ou utilizados em futuro processamento. Um comando de saída inicia sempre pela palavra reservada `escrever`, seguida da lista de valores que deverão ser informados, os quais aparecem separados por vírgulas e entre parênteses:

```
escrever ( <lista de valores de saída separados por vírgulas> )
```

A lista de valores de saída pode conter:

- nomes das variáveis cujos conteúdos devem ser informados;
- expressões que serão avaliadas, sendo seu resultado informado na saída;
- *strings* formadas por cadeias de caracteres entre apóstrofos simples. As *strings* não são analisadas pelo computador, sendo simplesmente copiadas para o dispositivo de saída. Servem para explicar ao usuário o significado dos valores que são listados.

Por exemplo, o comando:

```
escrever (a, b)
```

vai transferir para a saída o valor contido na variável `a`, seguido do valor da variável `b`. Já o comando:

```
escrever (a * b + 1 / sin (c))
```

vai avaliar a expressão fornecida (`a * b + 1 / sin (c)`), transferindo para a saída somente seu resultado. Como exemplo da utilização de *strings* na saída, considere o comando:

```
escrever ('Maior valor: ', maior, ' Resultado: ' , soma)
```

Neste comando, `valor` e `soma` são nomes de variáveis. Supondo que o valor contido na variável `maior` seja 15 e que o valor de `soma` seja 20, a saída produzida seria a seguinte:

```
Maior valor: 15 Resultado: 20
```

3.2.3 formatação de entrada e saída

As linguagens de programação possibilitam definir como os dados deverão ser apresentados, ou seja, como as informações deverão ser fornecidas na entrada de dados e como os resultados deverão ser dispostos na saída. Ao escrever um algoritmo, deve-se analisar somente quais as ações que devem integrá-lo para que represente uma solução adequada e correta para o problema que se pretende solucionar. Não é aconselhável definir formatos de entrada e saída neste momento, deixando para fazê-lo na tradução do algoritmo para uma linguagem de programação específica. Por isso, a pseudolinguagem utilizada ao longo deste livro não define a formatação de entrada e de saída de dados, deixando para mostrá-la nas seções específicas das linguagens Pascal e C.

3.3 → comando de atribuição

No **comando de atribuição**, o resultado de uma expressão é atribuído a uma variável, ou seja, é colocado no espaço de memória reservado para essa variável. Se já existia algum valor armazenado na variável, ele é substituído pelo novo valor, e o valor anterior é perdido. Na pseudolinguagem, um comando de atribuição tem à esquerda o nome da variável que vai receber o valor, seguido de uma flecha direcionada para a esquerda, seguida à direita pela expressão cujo valor vai ser utilizado na atribuição. O sentido da flecha representa visualmente que o resultado da expressão é colocado na variável:

```
<variável> ← <expressão>
```

Somente um nome de variável pode ser colocado à esquerda em um comando de atribuição. A execução do comando inicia avaliando a expressão à direita, colocando depois seu resultado na variável à esquerda. Por exemplo, no comando:

```
a ← b + 1
```

o resultado da expressão `b + 1` é atribuído à variável `a`.

O tipo da variável que vai receber a atribuição deve ser compatível com o resultado da expressão à direita. Dependendo do tipo dessa variável, três tipos de atribuição são identificados, os quais são analisados a seguir.

3.3.1 atribuição numérica

Se a variável for numérica, o valor da expressão deve ser também um valor numérico. O valor a ser atribuído à variável pode ser (1) informado diretamente através de um número, (2) o valor armazenado em outra variável numérica ou (3) o resultado de uma expressão aritmética. Pode ser utilizada qualquer expressão aritmética, incluindo nomes de variáveis e chamadas a funções. O nome da variável à qual vai ser atribuído o valor pode também ser utilizado na expressão aritmética – nesse caso, a expressão é avaliada com o valor que a variável continha antes da execução do comando, sendo esse valor perdido ao ser feita a atribuição do novo valor.

O tipo do valor a ser atribuído à variável deve ser compatível com o tipo da variável que vai receber a atribuição. Assim, variáveis inteiras devem receber valores inteiros, e variáveis reais devem receber valores reais. Uma exceção a essa regra, entretanto, é a atribuição de valores inteiros a variáveis reais.

Vamos supor que foram declaradas as seguintes variáveis:

```
i, k (inteiro)
a, r (real)
```

Utilizando essas variáveis, os comandos de atribuição a seguir são válidos:

```
i ← 10            { i RECEBE O VALOR 10 }
i ← k             { i RECEBE O VALOR DA VARIÁVEL k }
i ← k - 2         { i RECEBE VALOR DA EXPRESSÃO ARITMÉTICA k - 2}
i ← i + k - 7     { VARIÁVEL PODE SER UTILIZADA NA EXPRESSÃO }
a ← 10            { VARIÁVEL REAL PODE RECEBER VALOR INTEIRO }
a ← 3,14 * sqr(r) { EXPRESSÃO π.r² UTILIZANDO CHAMADA À FUNÇÃO }
```

Os comandos de atribuição a seguir são inválidos:

```
i ← a             { VARIÁVEL INTEIRA NÃO PODE RECEBER VALOR REAL}
i ← i + a         { EXPRESSÃO TEM RESULTADO REAL POIS a É REAL }
i ← a > r         { EXPRESSÃO NÃO PODE SER LÓGICA }
a + b ← a         { NO LADO ESQUERDO NÃO PODE TER EXPRESSÃO }
a, b ← 0          { SOMENTE UMA VARIÁVEL NO LADO ESQUERDO }
```

Uma construção bastante comum é a variável que vai receber o valor da atribuição ser também utilizada na expressão à direita do sinal de atribuição. Nesses casos, a expressão é avaliada utilizando o valor contido na variável e, terminada a avaliação da expressão, seu resultado é colocado na variável, alterando então seu valor. Por exemplo, supondo que a variável `a` é inteira e contém o valor 5, o valor contido em `a`, após a execução do comando de atribuição a seguir, é 8:

```
a ← a + 3
```

3.3.2 atribuição lógica

Se a variável à esquerda do comando de atribuição for lógica, ela poderá receber somente um dos dois valores lógicos: `verdadeiro` ou `falso`. O valor lógico a ser atribuído pode (1) ser representado explicitamente através de uma dessas duas palavras reservadas, (2) ser o conteúdo de outra variável lógica ou (3) resultar da avaliação de uma expressão lógica.

Considerar as seguintes declarações:

```
i, k (inteiro)
x, y (lógico)
```

Os exemplos a seguir são comandos de atribuição lógica válidos:

```
x ← verdadeiro    { VALOR LÓGICO INFORMADO EXPLICITAMENTE }
x ← y             { x RECEBE VALOR LÓGICO DE y }
x ← i = k         { x verdadeiro SE i IGUAL A k  }
x ← i > 7 ou y    { x RECEBE RESULTADO DA EXPRESSÃO LÓGICA }
```

Os comandos a seguir são inválidos:

```
x ← i             { i É INTEIRO, NÃO É VALOR LÓGICO }
x ← x > 7         { ERRO NA EXPRESSÃO LÓGICA }
x ← k + 1         { EXPRESSÃO TEM VALOR INTEIRO, NÃO LÓGICO }
```

3.3.3 atribuição de caracteres

Se a variável for do tipo caractere, a expressão à direita deve resultar em um caractere; se a variável for do tipo `string`, a expressão deve resultar em uma *string*.

Considerar que foram declaradas as seguintes variáveis:

```
nome (string)
letra, letra2 (caractere)
```

Utilizando essas variáveis, os exemplos a seguir mostram comandos de atribuição válidos:

```
nome ← 'Ana Terra'  { STRING ATRIBUÍDA À VARIÁVEL nome }
letra ← 'Z'         { letra RECEBE CARACTERE 'Z' }
letra ← letra2      { letra RECEBE CARACTERE DE letra2 }
```

Ainda utilizando as mesmas variáveis, os comandos a seguir são inválidos:

```
nome ← 10           { STRING NÃO PODE RECEBER VALOR NUMÉRICO }
letra ← i > 2       { CARACTERE NÃO PODE RECEBER VALOR LÓGICO }
letra ← nome        { VARIÁVEL CARACTERE NÃO PODE RECEBER STRING }
letra ← letra + 10  { EXPRESSÃO À DIREITA ESTÁ INCORRETA}
```

Parte 6

Formas de Representação de Dados

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.

Parte 7

Representação de Algoritmos em Forma de Fluxograma

3.4 → fluxograma de programas sequenciais

No Capítulo 1 foram mostrados alguns blocos utilizados em fluxogramas, incluindo os que representam comandos de entrada, de saída e de atribuição (Figura 1.4). Diversas formas para representar entradas e saídas podem ser encontradas na literatura. Nos blocos adotados nesse livro é utilizado o mesmo bloco para ambas, identificando claramente a ação a ser executada (entrada ou saída). A Figura 3.1 mostra um fluxograma em que é feita uma entrada de dados que preenche a variável `valor`, sendo, em seguida, informado qual o valor lido.

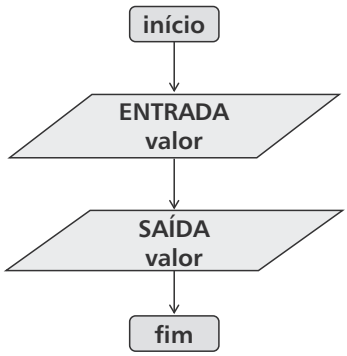


figura 3.1 Fluxograma com entrada e saída de dados.

Um programa pode ter vários comandos de entrada e de saída de dados em lugares diferentes. As formas dos blocos que representam esses comandos mostram visualmente, no fluxograma, os pontos de interação do programa com o usuário.

O comando de atribuição é representado através de um retângulo, dentro do qual é escrito o nome da variável, o símbolo que representa a atribuição (←) e a expressão, em sua forma matemática, ou seja, sem necessidade de representá-la em uma só linha. Como exemplo, a Figura 3.2 mostra o fluxograma que corresponde ao problema apresentado na Seção 1.2:

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

O primeiro passo corresponde a um comando de entrada de dados, em que são lidos dois valores. Para armazenar os valores lidos devem ser declaradas duas variáveis, `valor1` e `valor2`, de tipos compatíveis com os valores que serão fornecidos na entrada. No segundo passo, é

realizada a operação de soma dos valores contidos nas duas variáveis, sendo o resultado armazenado em outra variável chamada de `soma`. O terceiro passo corresponde a um comando de saída, através do qual o valor armazenado na variável `soma` é informado ao usuário. As setas indicam a sequência em que os comandos são executados.

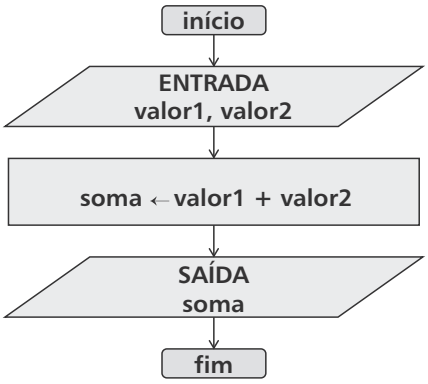


figura 3.2 Fluxograma da soma de dois valores.

3.5 estrutura de um algoritmo

Nesta seção será montado o primeiro algoritmo completo utilizando as declarações e os comandos vistos até aqui. Será utilizado o mesmo exemplo da seção anterior (soma de dois valores), para o qual já foi construído o fluxograma.

Um algoritmo deve sempre iniciar com um **cabeçalho**, no qual o objetivo do algoritmo deve ser claramente identificado. A primeira linha desse cabeçalho deve trazer o nome do algoritmo, o qual, por si só, deve dar uma indicação das ações a serem executadas pelo mesmo. No caso do exemplo, o algoritmo foi chamado de `Soma2`, pois vai efetuar a soma de dois valores. Na linha seguinte do cabeçalho, na forma de um comentário, deve ser explicado o objetivo do algoritmo. Essa explicação é útil principalmente nos casos em que o nome do algoritmo não é suficientemente autoexplicativo. Cabeçalho do exemplo utilizado:

```
Algoritmo Soma2
{INFORMA A SOMA DE 2 VALORES LIDOS}
```

Logo após o cabeçalho vem a seção das **declarações** de variáveis, de constantes e de tipos. Para facilitar o entendimento de um algoritmo, é importante identificar claramente as `variáveis de entrada` e `de saída`, pois elas fazem a interface do usuário com o programa. As demais variáveis utilizadas durante o processamento, denominadas `variáveis auxiliares`, são declaradas em uma linha especial. Essa separação desaparece ao se traduzir o algoritmo para uma linguagem de programação, mas é aconselhável que seja acrescentada ao programa na forma de um comentário.

Parte 8

Desenvolvimento de Algoritmos Sequenciais Através de Fluxogramas

3.4 fluxograma de programas sequenciais

No Capítulo 1 foram mostrados alguns blocos utilizados em fluxogramas, incluindo os que representam comandos de entrada, de saída e de atribuição (Figura 1.4). Diversas formas para representar entradas e saídas podem ser encontradas na literatura. Nos blocos adotados nesse livro é utilizado o mesmo bloco para ambas, identificando claramente a ação a ser executada (entrada ou saída). A Figura 3.1 mostra um fluxograma em que é feita uma entrada de dados que preenche a variável `valor`, sendo, em seguida, informado qual o valor lido.

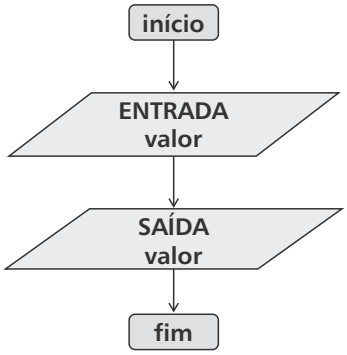


figura 3.1 Fluxograma com entrada e saída de dados.

Um programa pode ter vários comandos de entrada e de saída de dados em lugares diferentes. As formas dos blocos que representam esses comandos mostram visualmente, no fluxograma, os pontos de interação do programa com o usuário.

O comando de atribuição é representado através de um retângulo, dentro do qual é escrito o nome da variável, o símbolo que representa a atribuição (←) e a expressão, em sua forma matemática, ou seja, sem necessidade de representá-la em uma só linha. Como exemplo, a Figura 3.2 mostra o fluxograma que corresponde ao problema apresentado na Seção 1.2:

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

O primeiro passo corresponde a um comando de entrada de dados, em que são lidos dois valores. Para armazenar os valores lidos devem ser declaradas duas variáveis, `valor1` e `valor2`, de tipos compatíveis com os valores que serão fornecidos na entrada. No segundo passo, é

realizada a operação de soma dos valores contidos nas duas variáveis, sendo o resultado armazenado em outra variável chamada de `soma`. O terceiro passo corresponde a um comando de saída, através do qual o valor armazenado na variável `soma` é informado ao usuário. As setas indicam a sequência em que os comandos são executados.

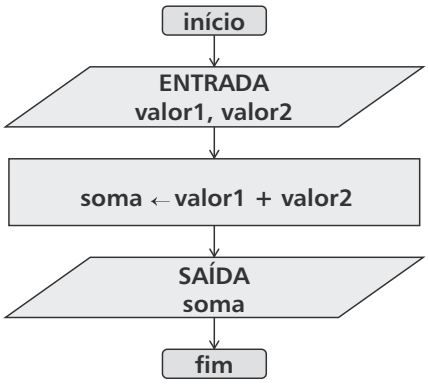


figura 3.2 Fluxograma da soma de dois valores.

3.5 estrutura de um algoritmo

Nesta seção será montado o primeiro algoritmo completo utilizando as declarações e os comandos vistos até aqui. Será utilizado o mesmo exemplo da seção anterior (soma de dois valores), para o qual já foi construído o fluxograma.

Um algoritmo deve sempre iniciar com um **cabeçalho**, no qual o objetivo do algoritmo deve ser claramente identificado. A primeira linha desse cabeçalho deve trazer o nome do algoritmo, o qual, por si só, deve dar uma indicação das ações a serem executadas pelo mesmo. No caso do exemplo, o algoritmo foi chamado de `Soma2`, pois vai efetuar a soma de dois valores. Na linha seguinte do cabeçalho, na forma de um comentário, deve ser explicado o objetivo do algoritmo. Essa explicação é útil principalmente nos casos em que o nome do algoritmo não é suficientemente autoexplicativo. Cabeçalho do exemplo utilizado:

```
Algoritmo Soma2
{INFORMA A SOMA DE 2 VALORES LIDOS}
```

Logo após o cabeçalho vem a seção das **declarações** de variáveis, de constantes e de tipos. Para facilitar o entendimento de um algoritmo, é importante identificar claramente as `variáveis de entrada` e `de saída`, pois elas fazem a interface do usuário com o programa. As demais variáveis utilizadas durante o processamento, denominadas `variáveis auxiliares`, são declaradas em uma linha especial. Essa separação desaparece ao se traduzir o algoritmo para uma linguagem de programação, mas é aconselhável que seja acrescentada ao programa na forma de um comentário.

A declaração de variáveis do Algoritmo Soma2 é a seguinte:

```
Entradas: valor1, valor2 (real)    {VALORES LIDOS}
Saídas: soma (real)
```

Os nomes escolhidos para as variáveis devem ser curtos e indicar qual a informação que elas irão armazenar. Caso isso não fique claro somente através do nome escolhido, é aconselhável escrever comentários explicando o significado de cada variável.

Após a seção de declarações, vem a área de **comandos**, delimitada pelas palavras reservadas início e fim. Cada comando deve ser escrito em uma linha separada. Ao contrário das linguagens de programação Pascal e C, a pseudolinguagem utilizada não emprega símbolo para separar comandos, sendo essa separação identificada somente pela posição de cada comando no algoritmo.

É importante utilizar **comentários** ao longo do algoritmo, indicando as ações que estão sendo executadas em cada passo. Isso auxilia muito os testes e a depuração do programa.

A estrutura básica de um algoritmo, com os elementos discutidos até o momento, é:

```
Algoritmo <nome do algoritmo>
{descrição do objetivo do algoritmo}
<declarações>
início
<comandos>
fim
```

Em declarações aparecem com frequência alguns ou todos os seguintes elementos:

```
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>
```

O algoritmo completo do exemplo da soma de dois valores é:

```
Algoritmo 3.1 – Soma2
{INFORMA A SOMA DE DOIS VALORES LIDOS}
  Entradas: valor1, valor2 (real){VALORES LIDOS}
  Saídas:   soma (real)
início
  ler (valor1, valor2)           {OBTÉM OS 2 VALORES}
  soma ← valor1 + valor2         {CALCULA A SOMA}
  escrever (soma)                {INFORMA A SOMA}
fim
```

Nos exercícios de fixação a seguir, recomenda-se definir inicialmente o(s) resultado(s) a produzir, a(s) entrada(s) a obter e, só então, tentar determinar um modo de solução. Procurar

identificar, nas soluções fornecidas, quais as linhas que correspondem, respectivamente, à entrada de dados, ao processamento e à apresentação dos resultados.

Observar que todos os problemas discutidos seguem o esquema básico destacado no início deste capítulo: entrada de dados, processamento e saída de dados.

3.6 exercícios de fixação

exercício 3.1 Fazer um programa que recebe três notas de alunos e fornece, como saídas, as três notas lidas, sua soma e a média aritmética entre elas.

A Figura 3.3 mostra o fluxograma deste programa. Inicialmente são lidas as três notas, que são também impressas para que o usuário possa verificar o que foi lido. Em seguida, é calculada e informada a soma. Finalmente, é efetuado o cálculo da média, que é também informado ao usuário. A utilização de diversos comandos de saída neste programa permite ao programador verificar quais os valores intermediários do processamento, auxiliando a depurar o programa.

O algoritmo desse programa acrescenta as declarações das variáveis utilizadas, que não aparecem no fluxograma. São incluídos também comentários para explicar os diferentes passos do algoritmo.

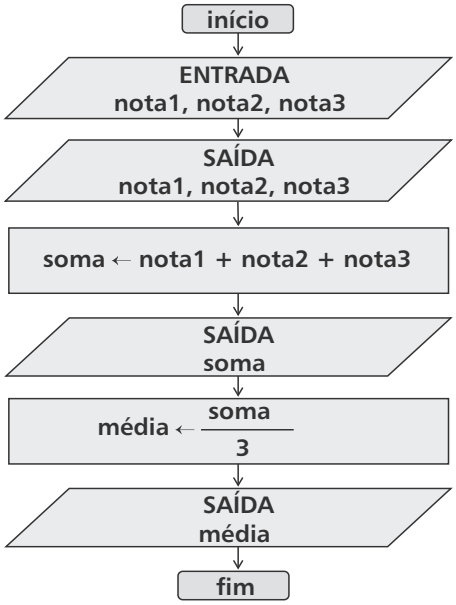


figura 3.3 Fluxograma do cálculo da média de três notas.

Parte 9

Representação de Algoritmos em Forma de Pseudocódigo

3.5 → estrutura de um algoritmo

Nesta seção será montado o primeiro algoritmo completo utilizando as declarações e os comandos vistos até aqui. Será utilizado o mesmo exemplo da seção anterior (soma de dois valores), para o qual já foi construído o fluxograma.

Um algoritmo deve sempre iniciar com um **cabeçalho**, no qual o objetivo do algoritmo deve ser claramente identificado. A primeira linha desse cabeçalho deve trazer o nome do algoritmo, o qual, por si só, deve dar uma indicação das ações a serem executadas pelo mesmo. No caso do exemplo, o algoritmo foi chamado de `Soma2`, pois vai efetuar a soma de dois valores. Na linha seguinte do cabeçalho, na forma de um comentário, deve ser explicado o objetivo do algoritmo. Essa explicação é útil principalmente nos casos em que o nome do algoritmo não é suficientemente autoexplicativo. Cabeçalho do exemplo utilizado:

```
Algoritmo Soma2
{INFORMA A SOMA DE 2 VALORES LIDOS}
```

Logo após o cabeçalho vem a seção das **declarações** de variáveis, de constantes e de tipos. Para facilitar o entendimento de um algoritmo, é importante identificar claramente as `variáveis de entrada` e `de saída`, pois elas fazem a interface do usuário com o programa. As demais variáveis utilizadas durante o processamento, denominadas `variáveis auxiliares`, são declaradas em uma linha especial. Essa separação desaparece ao se traduzir o algoritmo para uma linguagem de programação, mas é aconselhável que seja acrescentada ao programa na forma de um comentário.

A declaração de variáveis do `Algoritmo Soma2` é a seguinte:

```
Entradas: valor1, valor2 (real)    {VALORES LIDOS}
Saídas: soma (real)
```

Os nomes escolhidos para as variáveis devem ser curtos e indicar qual a informação que elas irão armazenar. Caso isso não fique claro somente através do nome escolhido, é aconselhável escrever comentários explicando o significado de cada variável.

Após a seção de declarações, vem a área de **comandos**, delimitada pelas palavras reservadas `início` e `fim`. Cada comando deve ser escrito em uma linha separada. Ao contrário das linguagens de programação Pascal e C, a pseudolinguagem utilizada não emprega símbolo para separar comandos, sendo essa separação identificada somente pela posição de cada comando no algoritmo.

É importante utilizar **comentários** ao longo do algoritmo, indicando as ações que estão sendo executadas em cada passo. Isso auxilia muito os testes e a depuração do programa.

A estrutura básica de um algoritmo, com os elementos discutidos até o momento, é:

```
Algoritmo <nome do algoritmo>
{descrição do objetivo do algoritmo}
<declarações>
início
<comandos>
fim
```

Em declarações aparecem com frequência alguns ou todos os seguintes elementos:

```
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>
```

O algoritmo completo do exemplo da soma de dois valores é:

```
Algoritmo 3.1 - Soma2
{INFORMA A SOMA DE DOIS VALORES LIDOS}
  Entradas: valor1, valor2 (real){VALORES LIDOS}
  Saídas:   soma (real)
início
  ler (valor1, valor2)           {OBTÉM OS 2 VALORES}
  soma ← valor1 + valor2         {CALCULA A SOMA}
  escrever (soma)                {INFORMA A SOMA}
fim
```

Nos exercícios de fixação a seguir, recomenda-se definir inicialmente o(s) resultado(s) a produzir, a(s) entrada(s) a obter e, só então, tentar determinar um modo de solução. Procurar

identificar, nas soluções fornecidas, quais as linhas que correspondem, respectivamente, à entrada de dados, ao processamento e à apresentação dos resultados.

Observar que todos os problemas discutidos seguem o esquema básico destacado no início deste capítulo: entrada de dados, processamento e saída de dados.

3.6 exercícios de fixação

exercício 3.1 Fazer um programa que recebe três notas de alunos e fornece, como saídas, as três notas lidas, sua soma e a média aritmética entre elas.

A Figura 3.3 mostra o fluxograma deste programa. Inicialmente são lidas as três notas, que são também impressas para que o usuário possa verificar o que foi lido. Em seguida, é calculada e informada a soma. Finalmente, é efetuado o cálculo da média, que é também informado ao usuário. A utilização de diversos comandos de saída neste programa permite ao programador verificar quais os valores intermediários do processamento, auxiliando a depurar o programa.

O algoritmo desse programa acrescenta as declarações das variáveis utilizadas, que não aparecem no fluxograma. São incluídos também comentários para explicar os diferentes passos do algoritmo.

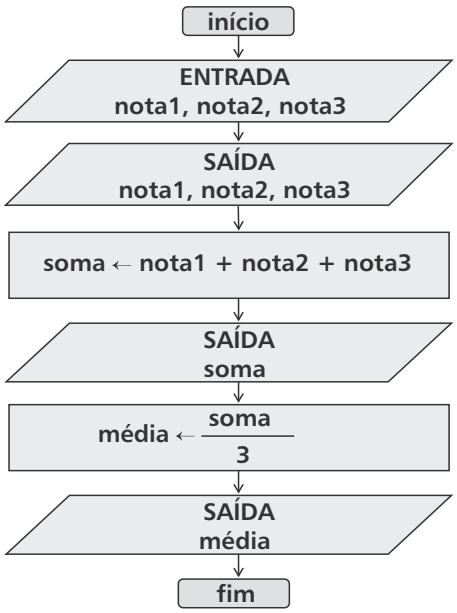


figura 3.3 Fluxograma do cálculo da média de três notas.

ANOTAÇÕES

CONTRIBUA COM A QUALIDADE DO SEU CURSO

Se você encontrar algum problema nesse material, entre em contato pelo email **eadproducao@unilasalle.edu.br**. Descreva o que você encontrou e indique a página.

Lembre-se: a boa educação se faz com a contribuição de todos!

UNIVERSIDADE LaSalle EAD

Av. Victor Barreto, 2288
Canoas - RS
CEP: 92010-000 | 0800 541 8500
ead@unilasalle.edu.br